Principal Typings for Interactive Ruby Programming

A Thesis Submitted to the
College of Graduate Studies and Research
in Partial Fulfillment of the Requirements
for the Degree of Master of Science
in the Department of Computer Science
University of Saskatchewan
Saskatoon, Saskatchewan, Canada

by

Andriy Hnativ

Permission To Use

In presenting this thesis in partial fulfillment of the requirements for a Postgraduate degree from the University of Saskatchewan, I agree that the Libraries of this University may make it freely available for inspection. I further agree that permission for copying of this thesis in any manner, in whole or in part, for scholarly purposes may be granted by the professor or professors who supervised my thesis work or, in their absence, by the Head of the Department or the Dean of the College in which my thesis work was done. It is understood that any copying or publication or use of this thesis or parts thereof for financial gain shall not be allowed without my written permission. It is also understood that due recognition shall be given to me and to the University of Saskatchewan in any scholarly use which may be made of any material in my thesis.

Requests for permission to copy or to make other use of material in this thesis in whole or part should be addressed to:

Head of the Department of Computer Science
176 Thorvaldson Building
110 Science Place
University of Saskatchewan
Saskatoon, Saskatchewan, Canada
S7N 5C9

Abstract

A novel and promising method of software development is the interactive style of development, where code is written and incrementally tested simultaneously. Interpreted dynamic languages such as Ruby, Python, and Lua support this interactive development style. However, because they lack semantic analysis as part of a compilation phase, they do not provide type-checking. The programmer is only informed of type errors when they are encountered in the execution of the program–far too late and often at a less-informative location in the code. We introduce a typing system for Ruby, where types will be determined before execution by inferring principal typings. This system overcomes the obstacles that interactive and dynamic program development imposes on type checking; yielding an effective type-checking facility for dynamic programming languages. Our development is embodied as an extension to `irb`, the Ruby interactive mode, allowing us to evaluate principal typings for interactive development.

Acknowledgements

I am sincerely thankful to my supervisor, Dr. Christopher Dutchyn, for his invaluable advices and ideas, and substantial support in many ways throughout my research.

Also, greatest thanks to my lab mates from Software Lab for being part of the great working environment.

Finally, biggest gratitude to all of those who helped me in any way during the completion of the project.

Table of Contents

List of Figures

List of Tables

List of Symbols

P	*set of primitives*
TV	*set of type variables*
FT	*set of fixed types*
UT	*set of unary types*
BT	*set of binary types*
PT	*set of proc/block types*
RC	*set of raw classes*
$\alpha, \beta, \gamma, \ldots$	*type variables*
$x \in S$	x *is a member of S*
$x \notin S$	*x is not a member of S*
$A < B$	*A is a subclass of B*
$\alpha * \beta * \gamma \rightarrow \chi$	*notation of a function expecting three arguments of types* α, β, *and* γ, *and returning a value of type* χ
$\alpha \triangleq \beta$	*type variable* α *is bound to another type variable β*
$\alpha := \mathtt{e}$	*type variable* α *is bound to final type/ expression* `e`
$\mathtt{A} \supset \mathtt{B}$	*class* `A` *contains constant* `B`
$\mathtt{a} \leftarrow \mathtt{b}$	*assign* `b` *to* `a` *in a substitution*

Chapter 1
Introduction

Historically, software has been developed using a top-down approach, where the software system design is divided into smaller pieces, each piece is implemented separately, and then combined into the final system [26, 48]. During the last decade, new ways to develop software have become popular. One reason is that the top-down approach has a number of drawbacks: inability to do integration testing early in the development process, inability to dynamically change the code, and a possible risk of rewriting large parts of the application due to changes in specification, are three of them [31, 44]. These drawbacks mean that when troubles are found, it is more expensive and difficult to discover their true causes and make changes to correct them. The system will require very frequent maintenance and replacement of important parts [9].

One of these new ways is called *incremental development,* where a system is built little by little incrementally until it is done. One little increment is implemented, inserted into the overall system, and tested, and this process is repeated until the system is complete [6, 20]. This means that each new little addition to the system is tested on every level of integration step by step, until they are tested together with a complete system [9].

One style of incremental development is *interactive programming* – a style of programming offering access to the code under development, interactive evaluation of the code under development, and access to intermediate execution states [12, p. 101]. Developers benefit by writing the pieces of the code without unnecessary analysis of the code as part of the end program, as the pieces of code are built independently. This provides a greater opportunity to experiment and to try out different ideas. Also programmers can combine interactive development with testing. They can write a procedure; then run and test it; then write another procedure that will use the completed one. They may tinker with different language constructs for writing a procedure at the same time, and when they are satisfied with the result, save the final version that will be integrated with the previously written and tested code. Thus using interactive programming they can make the maintenance phase of the software development more efficient as well.

Incremental development and interactive development used together offer many other advantages for programmers. The main ones are:

- focus on the code,
- early detection of errors,
- better program planning,
- and better control and understanding of problems.

First, interactive incremental style of development brings much convenience to programmers, as they keep their attention focused on coding and are not distracted by mechanical tasks such as saving files, updating source code, invoking a compiler, and reopening source files. To see this, consider the following example in Ruby. Imagine that a programmer writes a Ruby program to read data from a database, and later to process this data[1]. In this procedure the programmer connects to a MySQL database with the name `database1`, and using an SQL select query she extracts entries of two fields from the table named `employees` – names of the employees that are strings, and their salaries that are numbers. Later, she calculates the sum of all the employees' salaries. This process is shown in Figure 1.1.

```
def calcSum ()
   require 'mysql'  # connect MySQL library
   dbh    =    Mysql.real_connect("127.0.0.1",    "user",
                                  "password", "database1")
   sql = "SELECT name, salary FROM employeesSalaries"
   employees = dbh.query(sql)
   sum=0
   employees.each do |employee|
      sum+=employee[1]
   end
   return sum
end
```

Figure 1.1 Computing the overall payroll expense of a company

[1] An SQL query in Ruby returns an array with one entry for each row in the table. Each row's array is a further array containing fields in the order given in the `SELECT` query.

If the programmer is writing the code interactively, in order to test this function, she will just need to type the following short line in the interpreter[2]:

```
>> calcSum
```

This will output the result of the defined function, allowing the programmer to check the correctness of her function. If the programmer was not using interactive development techniques, in order to test the function she would need to write a complete program, compile it, and run it, that would take much more time and effort.

The second benefit of the incremental interactive development is the fact that it gives programmers additional early opportunities to find errors they committed. This is a benefit because usually the earlier an error is found, the easier it is for developers to correct it. Another manifestation of this benefit is that programmers are less likely to rely on or propagate broken code.

To see this, consider a slightly changed example from Figure 1.1. Imagine that the programmer made an error on the line 8 of Figure 1.2, where he, instead of adding salaries of employees that are located in the second field of the database, adds their names.

```
1: def calcSum ()
2:    require 'mysql'   # connect MySQL library
3:    dbh    =    Mysql.real_connect("127.0.0.1",    "user",
                                 "password", "database1")
4:    sql = "SELECT name, salary FROM employeesSalaries"
5:    employees = dbh.query(sql)
6:    sum=0
7:    employees.each do |employee|
8:       sum+=employee[0]  # Error – 0 instead of 1
9:    end
10:   return sum
11: end
```

Figure 1.2 Mistakenly adding names instead of salaries

[2] The initial >> prompt signifies the root level of the interaction session – the level where code can be executed and tested.

If the programmer follows an incremental interactive development style, she will test the defined function immediately, and she will get an error message from the interpreter that her code tries to add numbers to strings, that is not allowed. In this case, it will be a trivial task to correct the error. If the programmer did not follow this style, she might not be able to find this error immediately. By the time the error is discovered, she might have extended this function, for example, to calculate average, minimum, and maximum salaries as well, making the needed corrections more onerous.

The third benefit of the incremental interactive development is that it improves project planning: the code starts off working the way the programmers expect and stays working through the entire development process. Programmers need not wait until the whole program is written to find out if some parts of the program work. This allows subsets of code to be made available to end-users for testing and further requirements analysis, therefore saving time and cost for maintenance. These savings are significant; as Pressman noted, historically 60[3] percent of software life-cycle costs occur during the maintenance phase [29, p. 805]. To see this, consider again the example shown in Figure 1.2. If the programmer corrected the error at once, her function would be available for integration testing or use. Later, she will not need to come back to this code as she has tested it and ensured that it works correctly already.

Another important benefit of the incremental interactive development is that it helps to locate the actual source of many problems. The source of the error will typically be in the most recently changed or added code; that is usually small. A programmer will not need to look through thousands of lines of code to find the cause of the problem.

To show this benefit, consider a slightly abstract example of many mutually related functions. Imagine that the programmer spends a lot of time developing eight functions given in Figure 1.3.

Imagine that the programmer did not follow incremental interactive development and did not thoroughly test each function both alone and integrated with other defined functions. If she called the function `f8` after line 24 in Figure 1.3, she would get the following message from the interpreter:

[3] The oft-quoted 80% figure for Pressman's study is apparently a shibboleth, which he himself repaired with the 60% figure after a 1993 study by Hanna [14].

```
TypeError: String can't be coerced into Fixnum
        from (irb):2:in `+'
        from (irb):2:in `f1'
        from (irb):5:in `f2'
        from (irb):8:in `f3'
        from (irb):11:in `f4'
        from (irb):14:in `f5'
        from (irb):17:in `f6'
        from (irb):20:in `f7'
        from (irb):23:in `f8'
        from (irb):25
```

```
1:  def f1 a,b
2:      a+b
3:  end
4:  def f2 a,b
        # some code
5:      f1 (a,b)
6:  end
7:  def f3 a,b
8:      f2 (a,b)
        # some code
9:  end
10: def f4 a,b
11:     f3 (a,b)
        # some code
12: end
13: def f5 a
14:     f4 (5,a)  # The function f4 expects the first argument to be a number
        # some code
15: end
16: def f6 a
        # some code
17:     f5(a)
        # some code
18: end
19: def f7 a
        # some code
20:     f6(a)
21: end
22: def f8 a
        # some code
23:     f7(a)
        # some code
24: end
25: f8("string")
```

Figure 1.3 Many mutually related functions with an error hidden deep in the code

Here the error message consists of many lines, making it not easy to understand and find a root cause of the problem. In this example the real cause of the problem can be found in the body of the function `f5`, where a programmer constraints the argument of this function to be an integer, because `f4` forces it to have the same type as `5`. If instead of `5` the programmer wrote `"hello"`, the code would be correctly typed. But as it can be seen from the error message above, the function `f5` was mentioned by the interpreter only in the middle in the list of all the functions that were called. To understand the true source of the error, the programmer will need to examine all of functions that were used, as the original source of this problem is hidden deep in this body of code. This kind of message is difficult to understand as it involves many functions.

In general, in order to understand the meaning of the lengthy message, a programmer would need to keep many functions and relations between them in her head, a large cognitive burden. If the programmer followed the incremental interactive development style and tested each function right after their definitions, she would get a similar message after the definition of the function `f5`, thus she would be able to find and correct the source of the problem at once.

All the above-mentioned benefits of the incremental development—focus on the code, early detection of errors, better program planning, and better control and understanding of problems—make it an increasingly popular technique as different developers search for ways to use it. Because of the benefits that they provide, the languages that support interactive programming (eg. Python, Ruby and Lua) are becoming increasingly popular among developers.

Unfortunately, interactive facilities for these sorts of languages are far from perfect. These facilities have drawbacks that do not allow a programmer to realize the whole potential of these languages. One of these drawbacks is that many kinds of common errors will not be recognized by the language interpreter, therefore will not be signaled to a programmer until a late stage of the software development. In many cases, even with the interactive mode, it is impossible to run the code as soon as it is written, as the function under development may rely on some other functions not defined yet. Therefore programmers will continue writing new code while having not corrected old code. This leads to the same problems as the traditional development style.

Consider an example where the current interpreter fails to work in a convenient way for programmers. We will change the code from Figure 1 to contain a social insurance number (SIN) of employees instead of salaries in the database, and to display the SIN in a formatted way. The changed code is shown in Figure 1.4. Here, the programmer has provided arguments in the wrong order to the function `displayEmp` resulting in a runtime value error.

When writing this kind of code, a programmer will not gain the benefits of the incremental interactive development as only after line 27 can the first function be called and tested. Therefore, only at that point will the type error be reported by the Ruby interpreter. In Figure 1.4 we show that the function was called only at line 143 to show the fact that the programmer did not realize she made an error, as no error message was reported to him. If a programmer tries to call this function before the remaining procedures are written, she will get a message from a Ruby interpreter that tested procedure relies on pieces of code that are not yet defined[4].

When the programmer tests the `listEmployees` function at the line 143, the Ruby interpreter reports an error:

```
NoMethodError: undefined method `/' for "Harry Gerrard":String
        from (irb):33:in `printSIN'
        from (irb):21:in `displayEmp'
        from (irb):18:in `listEmployees'
        from (irb):143
```

Note that this kind of error can be observed much earlier, after the second function definition is finished at line 17. The error is at line 10, when the programmer provided arguments for the function `displayEmp` in the wrong order. Line 8 shows, that the second field of the database contains integer type by adding it to another integer value that is stored in the variable `sum`, while line 15 anticipates it (as it was passed as a second parameter) to be a string, as it will be used in the addition operation with another string. Further, we also believe that the error message received is not particularly informative; the programmer may spend a lot of time locating the cause of the error.

[4] It is possible to test with stub functions, but often those functions do not represent the future implementations of the corresponding functions correctly.

To summarize, the error message about the error committed at line 8 in Figure 1.4 was reported too late – at runtime – rather than as early as possible. Not reporting errors in a timely fashion exacerbates coding difficulty. It makes locating the root cause harder. Even worse, the error may get carried down into additional code, making the repair larger and more time consuming, similar to that seen in Figure 1.4.

```
1:  def calcSum
2:     require 'mysql'
3:     dbh = Mysql.real_connect("127.0.0.1", "user",
                                "password", "database1")
4:     sql = "SELECT name, salary FROM employeesSalaries"
5:     employees = dbh.query(sql)
6:     sum=0;
7:     employees.each do |employee|
8:        sum+=employee[1]
9:     end
10:    displayEmp (employee[0], employee[1])
11:    puts sum
12: end
13: def displayEmp (SIN, name)
14:      printSIN SIN     # nnn-nnnn-nnn
15:      printName “name: ”+name
16:  end

17:  def printSIN SIN
     # display SIN as nnn-nnnn-nnn
18:     a1= SIN%1000
19:     SIN /=1000
20:     a2= SIN %10000
21:     SIN /=10000
22:     a3= SIN %1000
23:     printf "#{a3}-#{a2}-#{a1}\n"
24:  end

25:  def printName name
26:      puts name
27:  end

# many lines of code elided

143: >> listEmployees
```

Figure 1.4 Function arguments out of order

Also, it is worthwhile remembering that a programmer often must write many complicated test cases to completely exercise a piece of code, and accordingly, to spend much time developing and running those test cases. Immediate checks whether the code contains consistent types can be done by the interpreter in many cases.

Clearly, one source of coding difficulty in the interactive languages like Ruby, Python, and Lua is the absence of a type system. Types [25] are restrictions put on the expressions that show how the results of expressions can be used. If the expression does not have restrictions, it can be used in all possible operations. Type errors indicate the situations where expression results will be used incorrectly. All the error examples shown in this chapter are instances of type errors, that can be recognized earlier than with current interpreters by using types.

We describe a typing system for the interactive language Ruby. Having typing in the language brings advantage, as the operations that are allowed for different types are known in advance, for example, we know that bitwise operations can be performed for integers and not for strings, and code can be checked *before the execution* to notify the programmer of inconsistencies. As a result, the development process will be enhanced, as a programmer will not need to search as deeply for errors when she learns that something is wrong with her code. This should lead programmers to correct code faster, and prevent them from relying on erroneous code as often. For example, for the code in Figure 1.3 and Figure 1.4, a type checker will report error messages right after the incorrect code is written, thus urging the programmers to correct them at once before proceeding with other coding.

At first glance, the easiest way to add a typing system to interactive languages is to look at the type systems for traditional, compiled, languages, and put the similar systems into interactive languages. For interactive programming languages, we believe there is a potentially more effective approach that supports the distinctiveness of interactive development. Code in traditional languages can be compiled only after the program is complete, thus a situation of using fragments of code (functions and variables in particular) before their definition is not possible for them. In contrast, this case can occur for interactive languages, as we saw previously in this chapter (Figure 1.4 in particular), and is in fact quite common for them. Thus interactive languages appear to require a different, more sophisticated, type system that can handle incomplete and evolving programs. A *principal typings* system provides this facility, by carrying incomplete type information until it can be resolved and checked.

We choose to explore this facility with the Ruby programming language [10], because

- Ruby is a widely-used programming platform, which offers the opportunity to evaluate the principal types inference on a large body of production code.
- Ruby contains most of the essential constructs found in other dynamic languages, such as Javascript, Python, Lua, and Scheme.

As an exemplar of dynamically-typed languages, success with Ruby will inform the development of interactive development environments for the entire range of dynamic languages. The challenges and techniques for typing Ruby described in this thesis will apply for other dynamic languages.

Thesis Statement

> Principal typings improve the interactive software development process for Ruby by supplying
>
> - specific and targeted
> - informational and error-reporting
>
> messages to the programmer at an earlier stage of development.

The rest of the thesis is organized as follows. Chapter 2 gives a general background of type inference and basic type inference algorithms (Section 2.1), the novelty of principal typings (Section 2.2), and some features of Ruby, including examples of dubious programming constructs (Section 2.3). In Chapter 3, we provide detailed information about typing Ruby, what approaches have been attempted, and what challenges Ruby presents. Chapter 4 discusses the implementation details of our system, Rubin; here we give information of tools used, data structures, and key algorithms used. Chapter 5 evaluates our system. A summary and a discussion of possible future work conclude this document in Chapter 6.

Chapter 2
Background

In the previous chapter, we explored interactive incremental development. We mentioned that there are some problems with this style that can be corrected by adding typing to the languages that support this style. Before we delve into typing and type inference systems for interactive languages, we lay the groundwork to understand these systems.

This chapter contains four topics. First, we will talk about some specifics of Ruby, language that we chose to do our research for. Next, we will discuss type inference, and the most popular algorithm of type inference, the Hindley-Milner-Damas algorithm. Third, we will explain the notion of principal typings, and shows, where they can be beneficial to use. Finally, we will turn our attention to some work related to ours that has been done for other interactive programming languages.

2.1 Ruby

Ruby is a dynamically-typed object-oriented language that supports interactive programming. Ruby has several benefits that assure the growth of the popularity of this language. In addition to being extendible, portable, and interactive, the key benefit of Ruby is that it is clean. It means that the language is concise, much more succinct than many other popular languages (including Java and C++). One main reason for this is that Ruby does not have type annotations for the variables and functions. As a result of this conciseness, many agree that the language is very productive [8, 30]. Many programmers who switched to Ruby from other languages (from Java or even C) noticed an increase in their coding productivity [30]. The explanation for this benefit is said to be that Ruby requires fewer lines of code to solve the same problem. Among other things, fewer lines may mean fewer bugs, less coding time, and less cost to build an application.

We will use Ruby as our prototypical dynamically-typed, interactively-developed programming language. During the remainder of the thesis, we will examine a number of examples of code, using standard Ruby constructs. Here, we provide a brief overview of each of those constructs. We begin with classes and functions, then briefly discuss different kinds of variables, and finish with control-flow constructs and other fundamental features of Ruby.

2.1.1 Classes and Modules

Ruby is a completely object-oriented language: everything in it is an object. Objects are categorized by classes, which are templates for similar data structures. Like Java or C#, a special class called `Object` is the root of all classes. All other classes inherit from it, thus all classes have some methods inherited immediately, for example, `to_s`, a method to provide a string representation of an object, is immediately available to all classes. Moreover, the Ruby interpreter starts the interaction session in the body of the `Object` class; hence, all instance variables defined at the root interaction level will be instance variables of the class `Object`. Ruby supports single inheritance and prohibits overloading; only a few built-in functions like `+`, `insert`, are overloaded.

Class declarations begin with the `class` statement giving the name of the class, and finish with a matching `end`. To create an instance of a class, programmers must use the `new` keyword. This method invokes the constructor of a class, a block of code, the purpose of which is a creation of a class's instance. Ruby provides default constructors for all classes. These default constructors do not expect any arguments supplied when they are called, and do nothing more than creation of an object with default field values. Ruby has a number of predefined classes: `Integer` – supports operations on integers, `String` – supports operations on strings, `Regexp` – supports operations on regular expressions, etc.

Ruby also has a concept of *modules* – constructs similar to classes, but without an ability of being instantiated and being inherited from. Modules however can be mixed into classes by the `include` construct. In this case, those classes will have all the functionality of the modules available inside them.

As a side note, modules are distinct from files that are textually included by the require keyword. The require keyword provides a way of modularizing code, so duplicated operations can be contained in a single shared file. Modules have a different purpose; that of mixing functionality into classes.

Classes may declare methods that can be accessed only after the class instantiation. The exception is a set of class methods (for modules: module functions), which can be called with the name of a class/module as a receiver. There are several ways to define a class/module function; the most common of them is to put a keyword self and a dot in front of a function's name in its declaration. In Figure 2.1 the function class_fun is a class function, and it can be called with the name of its class as the receiver. The function not_class_fun is an instance method, hence calling it with the name of its class as the receiver is not allowed: it must be called with the name of the class's instance as the receiver.

```
class A
  def self.class_fun
    print “I am a class function”
  end
  def not_class_fun
    print “I am not a class function”
  end
end
A.class_fun         # correct
A.not_class_fun     # not correct
```

Figure 2.1 Example of a class function

Ruby supports a notion of *singleton functions* – functions, that are present for usually one instance of a class. Singleton functions are defined in the same way as ordinary ones, but with the name of the necessary instance of a class in front of the function's name. In Figure 2.2 the programmer defines a singleton function identify for the variable c, which is an instance of the String class. Then he will be able to call this instance function for the variable c; for all other instances of the class String this function will not be available.

```
c=String.new
d=String.new
def c.identify
  print "I am the variable c"
end
c.identify #prints "I am the variable c"
d.identify #Error
```

Figure 2.2 Example of a singleton function

Classes and modules definitions create new scopes for variables. This means that methods of some inner class will not have access to the variables of the outer class. Also both classes and modules can be nested within others.

2.1.2 Functions

Ruby, as most other popular programming languages, allows programmers to reuse code by writing functions[5]. Ruby has a rich set of built-in libraries with a myriad of functions. For example, Ruby supports all popular arithmetic operations (+, -, *, /, % (modulo), div), comparison operators (==, >, <), etc. There are many libraries of functions developed by many programmers specifically for Ruby. For example, the MySQL library provides an interface for Ruby to work with MySQL databases.

Functions are defined using the def keyword. Unlike Java, Ruby supports default arguments, but they must come after all other arguments. There is also a way to specify that the function will accept an arbitrary number of arguments. This is done by replacing the last formal parameter with a * prefix. This argument will behave as an array containing all other arguments provided.

Figure 2.3 shows definition of the function fun that will accept an arbitrary number of arguments. Whenever this function is called, all the arguments provided may be accessed as elements of the array p.

[5] As Ruby is a pure object-oriented language, all Ruby functions are actually methods on objects.

```
def fun(*p)
  ...
end
fun                          # correct
fun(25)                      # correct
fun(25,"hello", 45, 67)      # correct
```

Figure 2.3 Example of a function with an arbitrary number of arguments

Ruby functions return either values of expressions specified after the `return` keywords, or expressions that were evaluated last in bodies of those functions. Function definitions introduce new scopes. That means, for example, that local variables, defined at the same level as a current function, will not be accessible from inside of another function’s definition (as in Figure 2.4 the variable `a` is not accessible from the body of the function `cannot_see_a`).

```
a=8
print a                      # correct, 'a' is accessible here
def cannot_see_a
  c=a                        # Error, 'a' is not accessible here
end
b=lambda {|d| d+a}           # correct, 'a' is accessible here
b.call(2)                    # call of a lambda-function
```

Figure 2.4 Example of function scoping

There are many possible operations with functions. For example, each function can be duplicated (using the `alias` keyword), or can be destroyed using the `undef` keyword. In Figure 2.5, the function `will_be_undefined` was created, duplicated into the function `remains`, and afterwards removed.

```
def will_be_undefined
  ...
end
alias remains will_be_undefined  # function ‘foo1’ created with the very
                                 # same functionality as foo
undef will_be_undefined
will_be_undefined                # Error
remains                          # Correct
```

Figure 2.5 Example of function aliasing and removal

Ruby also supports anonymous functions (so called *lambda functions* or *procs*). Unlike ordinary functions, anonymous ones do not introduce new scopes, thus they have access to local variables of the outer scope. They behave like usual values, so any variable can be an anonymous function. In order to call a function of this kind, programmers must apply the `call` method.

Figure 2.4 shows the scoping difference between ordinary functions and lambda functions. The ordinary function `cannot_see_a` in this example does not have an access to the variable `a` defined in the same scope, and the lambda function, that is stored in the variable `b`, can access the variable `a`.

2.1.3 Variables

All variables can be discriminated by two properties: *scope* and *extent*. The former defines where the variable is visible, where it may be accessed and changed. The latter describes the lifetime of that variable. Based on this information, Ruby contains four different kinds of variables: instance, class, global, and local variables.

- *instance variables*: These are fields forming an object which is the instance of a class. Instance variables are visible within each method of the class, and throughout the body of the class as well. They last for the lifetime of the object. All instance variables in Ruby are private. Ruby identifies instance variables with a `@` prefix.

Figure 2.6 gives an example of instance variables. There are two instance variables in the class `Rectangle`: `@width` and `@height`. As this example shows, they can be accessed from inside `Rectangle`'s methods (like from the method `perimeter`), but cannot be seen from outside of the class definition.

```
class Rectangle
  @width=0                # this is an instance variable
  @height=0               # this is an instance variable
  def perimeter
    2*@width+2*@height    # the only way to access instance variables –
                          # through class methods
  end
end
r = Rectangle.new
print r.@a                # error – cannot access instance variable this way
```

Figure 2.6 Example of instance variables

- *class variables*: These are similar to instance variables with the difference that class variables are associated with the class rather than any particular instance of the class, and are the same across all object instances (class variables in Ruby are similar to class static variables in Java or C++) [10]. Ruby class variables exist not only for classes in which they are defined, but are shared with all their descendant classes. Ruby identifies class variables with a `@@` prefix.

Figure 2.7 provides an example of class variables. As this example shows, whenever a programmer changes a value of a class variable for one instance of the class or its subclass, `instBase1` in the example, the other instances will change the value of the corresponding variable too.

- *global variables*: These are variables which are visible everywhere and which last from the time they are first created, throughout the remaining lifespan of the entire program. Ruby distinguishes these variables with a prefix `$`.

Figure 2.8 shows typical usage of global variables. This example shows that global variables can be accessed from a variety of scopes; in fact, they can be accessed from any scope.

```
class BaseClass
    @@var = 1                                 # this is a class variable
    def change_var
      @@var+=1
    end
    def var
      @@var
    end
end
class SubClass < BaseClass                    # 'SubClass' is a subclass of
     ...                                      # 'BaseClass'
end
instBase1 = BaseClass.new
instBase2 = BaseClass.new
instSub = SubClass.new
print instBase1.var                           # outputs 1
print instBase2.var                           # outputs 1
print instSub.var                             # outputs 1
instSub.change_var
print instBase1.var                           # outputs 2
print instBase2.var                           # outputs 2
print instSub.var                             # outputs 2
```

Figure 2.7 Example of class variables

```
$s                    # Error, the global variable $s wasn't initialized yet
$s=7
print $s              # prints "7"
def globVarVal
   $s
end
globVarVal            # returns 7
```

Figure 2.8 Example of global variables

- *local variables*: These are variables, which are visible only within one scope that is current when they are created. These variables do not have any prefixes. Function formals are local variables for the scope inside the function's body.

Figure 2.4 shows, how the local variables can be used; they can be accessed from the same scope where they were created (printing the value of the variable `a`), but the programmer's attempt to access a local variable from a different scope will fail (reference to the variable `a` from the body of the function `cannot_see_a`).

2.1.4 Assignments

Ruby assignments are written with the assignment operator, =. One interesting feature of Ruby assignment operator is that it supports multiple assignments, when several values are assigned to several targets in one assignment operator. For example, after only one line of code:

```
a, b, c = 1, "hello", [2.3, 4.5]
```

the variable a will contain 1, the variable b will contain string "hello", and the variable c will contain array with float values.

When multiple assignments are used, the Ruby interpreter matches elements from the list of targets (the list on the left side of the = operator) to the corresponding elements (elements with the same ordering number) from the list of values (the list on the right side of the = operator). If there are more values that targets, excessive values are ignored. If there are more targets than values, excessive targets are still created, but they will have a null value.

2.1.5 Iteration

Ruby has an extensive support of loops. Almost all common loops (for and while in particular) are present here. For all Ruby iteration constructs a new scope for the body of the iteration is not introduced. However, the variables created inside the body of an iteration statement will not be accessible after that statement (as with the variable a in Figure 2.9).

The execution of all iteration statements can end:

- *normally*: after condition to continue iteration is not true anymore
- *abruptly*: as a result of one of the following statements: break, next, or redo.

The break statement in Ruby is similar to break statements in other popular languages: if the Ruby interpreter encounters this statement during the execution of a code, it terminates the smallest enclosing loop immediately. The next statement is similar to `continue` in C++ or Pascal: if the Ruby interpreter encounters this statement during the execution of a code, it terminates the current iteration of the loop, and starts a new one. The `redo` statement is similar to the `next` statement, but it restarts the loop iteration again rather that continuing on to the next iteration. Usage of any of these three statements outside of a loop is not allowed.

Ruby has two kinds of iteration constructs: guarded iteration and bounded iteration.

2.1.5.1 Guarded Iteration:

Just as with most imperative languages, Ruby includes the standard guarded iteration constructs: `while` and `until`. They differ in what the guard tests for: in a `while` loop, the guard tests for continuation, an `until` loop guards for termination. In other words, `while` runs until its condition is true, while `until` runs until its condition is not true.

Figure 2.9 shows an example of the `while` loop usage. The loop's body contains a reference to the outer local variable `x`, which is allowed. However, an attempt to access the variable `a` created inside of the loop will not be successful.

```
x=10          # 'a' not defined
while (x>5)
    print x  # 'x' is accessible, and on the first iteration is equal to 10
    x=x-1
    a=x
end
print a              # Error – variable 'a' is not visible here
```

Figure 2.9 Example of guarded iteration

2.1.5.2 Bounded iteration:

Ruby supports non-guarded iteration loops: loops that run constructs in their bodies a certain specified number of times. Constructs of this kind are called with a *block*: a special piece of Ruby code that can accept arguments, and can be passed around. Blocks are similar to lambda-functions, but they cannot be explicitly called, unlike lambda functions are called with the `call` method). Bounded iteration uses blocks with a variable that represents the current iteration. This variable can be used, for example, to access specific elements in an array, for example, for in-order printing. We show this case in Figure 2.10.

One example of bounded iteration is the `each` method. It is used to iterate through all elements of some container. For example, in Figure 2.10 calling the `each` function to the array `a` starts iteration through each of the elements of this array, one at each iteration. In our example, execution of the `each` statement prints all the elements of the array `a`, each on a separate line.

```
a=[1,2.3]
a.each { |index| print a[index] + "\n" } #prints:
                                         #  1
                                         #  2.3
```

Figure 2.10 Example of bounded iteration

2.1.6 Conditionals

Ruby supports most common conditionals:

- `if`: evaluates a body expression if condition is true
- `unless`: evaluates a body expression if condition is false
- `case`: evaluates options when a matching condition is true

All of them also accept an `else` block, and `if` statement can accept `elsif` blocks as well. `then` statement is acceptable, but not necessary.

All of these three conditional statements do not introduce new scopes. Any variables created inside the body of the single branch that is evaluated will be accessible after the conditional.

Figure 2.11 shows the usage of `if` conditional statement in Ruby. In it, the value of the variable `d` after the conditional will be determined based on the value of the previously defined variable `a`.

```
a=7       # d and b do not exist
if a>6    # this condition is true, so this branch will be executed
  d=4
elsif a<6 # not executed
  d=5
  b=5
else      # a==b; not executed
  d=6
  b=6
end
puts d    # will print either 4
puts b    # Error – b was not created
```

Figure 2.11 Example of conditionals

2.1.7 Types

In the next chapter we will explore the complexity of Ruby types in detail. Here, we note that Ruby supports the normal range of simple values:

- integers
- floats
- strings – `String` keyword.

Ruby also supports more complicated values like:

<u>2.1.7.1</u> <u>Array</u> [36]

Ruby arrays are very similar to arrays in other popular languages. Ruby has a built-in class, `Array`, and programmers have several options how to create an array. They can either call a constructor of the `Array` class

```
a=Array.new
```

or they can use square brackets to specify that they want to create arrays.

```
a=[element1, element2]
```

As in many other languages to access a specific element of the array, Ruby programmers use square brackets `[ ]` with the index of the element.

```
a[0] # returns the first element in the array ‘a’
```

<u>2.1.7.2</u> <u>Range</u> [38]

Ruby Ranges represent intervals—sets of discrete values with a start and an end. Ranges may be constructed using the `s..e` and `s...e` literals (the latter does not include the last element inside the range), or using the `Range::new` construct. When used with a bounded iterator, ranges return each value in the sequence.

2.1.7.3 Hash [37]

Ruby hashes are associative arrays, similar to those of other modern languages. As for arrays and ranges, there are two ways to create a hash: construct them using `{}` literals (for example, {“a” => 1, “b” => 2}), or call the constructor `Hash.new`. To get a value for a specific key of the hash programmers must use square brackets `[]`, just like arrays.

2.1.7.4 Symbol [5, 39]

Symbol objects represent names and some strings inside the Ruby interpreter. They are generated by prefixing a colon with an identifier (`:name` or `:"string"`), and by `to_sym` methods present for many classes [39]. Symbols are similar to strings, but they are memory efficient [5]. Internally symbols are stored as integers, so the maximum space that one symbol takes in memory is never bigger than the space taken by an integer. Symbols are similar to interned strings in Java [13, 15]: the same identifier points to the same memory location.

All values have many built-in standard operators: strings, for example, can be concatenated using the `concat` method; one hash can be blended with another using the `update` method, and so on. For a complete list of all built-in methods for Ruby types, we refer the reader to [35].

2.1.8 Exceptions

Exceptions in Ruby are handled in a similar manner as Java, except that `try-catch-finally` block in Java is spelled as `begin-rescue-ensure` block in Ruby. Figure 2.12 shows an example of exception handling. The programmer tries to open a file. If there was any runtime error during this process, for example, if the file was not found, the `rescue` block will be triggered. The `ensure` block always is evaluated last, regardless of whether any exception was raised.

```
begin
 file = File.open(“1.txt”) #.. process
rescue
   #.. handle an exception
ensure
   #.. always runs
end
```

Figure 2.12 Example of exceptions

2.1.9 Constants

Constants usually indicate values that are not supposed to change their values or types. For example, Ruby classes are constants. Ruby distinguishes constants with an uppercase first letter in their names.

Ruby is a very flexible language regarding constants. Programmers can access constants from any scope, using `::` operator to change scopes. In Figure 2.13, a constant `C` from the class `B` is accessed from the root level.

```
class A
  class B
    C=3
  end
end
v=A::B::C  # v is equal to 3
```

Figure 2.13 Example of constants

2.1.10 Reflection

Reflection is a way to access and possibly modify the program directly at runtime. For example, a programmer may want to create an instance of a class depending upon the parameter passed to a function. This parameter could be the name of the class instance of which will be created.

Another example of Ruby’s reflection is the `eval` statement. It takes only one parameter that is a string, and treats it as if it were real program syntax to be evaluated.

There are many other things one can do with reflection, and for thorough information, refer to [42, 45]. In my experience, reflection is not used heavily for most Ruby projects; hence most Ruby programmers can avoid learning this part of the language.

2.1.11 Other

There are some other Ruby constructs worth mentioning here:

❖ *output methods*

Most common among them are:

- `puts`: output with the carriage return
- `print`: output without the carriage return
- `printf`: formatted output

❖ *input methods*

Probably the most common method is `gets` – it reads a user's input to a string that is its single parameter

❖ *comments*

Ruby allows programmers to write comments for their code by putting them after the # symbol.

The constructs described in this section are the most heavily used Ruby constructs. The next piece of background we explore is typing of programming languages, and a particular technique of typing called type inference.

2.2 Type Inference

Types are sets of allowable operations for values. They are used to put restrictions on values so programmers and compilers know how the values are permitted to be used. Each value is represented by a set of bits in memory; types inform programs and programmers how those sets of bits should be treated. [7, 18, 25]

Probably most common types are numbers (integers and floats) and strings. Arrow types constitute a particular subclass of types: types that represent functions. We will distinguish those types with → symbol, where elements mentioned on the left side of the arrow will indicate function parameters, and the single element on the right side that will indicate the return value. For example, the notation `a*b*c → r` describes a function expecting three arguments of types `a`, `b`, and `c`, and returning a value of type `r`.

The rules for forming judgments about types are formalized as a *type system*. The analyzer that keeps track of types and checks that the rules are obeyed is called a *type checker*. Type systems are usually conservative and terminating, as this way they can guarantee the type safety for the programs.

Some kinds of type systems are present in almost every programming language. Arguably, the most common type systems are *static type system*s, systems that ensure *type safety* (correctness in using types) of the code before runtime, that is, no type errors will be encountered at runtime [25]. Most systems of this sort enforce safety by requiring programmers to indicate and adhere to future restrictions for values using *static type annotations* – a set of keywords that identify types. For example, if a programmer wants to use a variable with the name `iCount` in Java, he needs to define it along with its type, before he will be able to use it.

```
int iCount;  // definition
iCount=0;    //  use
```

Often languages are more liberal and do not require type annotations to be annotated by the programmer. Still, many of them guarantee the type safety of accepted programs. The process that allows this to happen is called *type inference*—the process of finding types for expressions from the code itself without annotations [18].

Consider the Ruby code given in Figure 2.14. What is its type?

```
def reciprocal a
  return 1.div(a)
end
```

Figure 2.14 A function to compute reciprocal for a number

We know at first glance that the function `reciprocal` takes one argument and returns some value[6]. But using a type inference algorithm we can determine the type of the function to be

`Number → Number`

[6] This is an important observation as not all functions return values, and not all functions take arguments.

This type signature means that the function takes one argument, which must have a `Number` type, and returns a `Number` value. Indeed, in Ruby the predefined function `div` for integers (in the example the function is called with receiver equal to 1, which is an integer) accepts only a number, and returns a number as the result. We have inferred the type of `reciprocal`.

Type inference is built on two important concepts: constraints and type variables. To motivate the latter, consider another example:

```
def identity x
   return x
end
```

Figure 2.15 A polymorphic function

The function `identity` is *polymorphic,* meaning it accepts many types– the type of the return value is restricted to the same as the type of the argument. That type may be string, or integer, or any other type. This kind of code cannot be typed using simple types like integers or strings. Instead, *type variables*, variables that represent unknown types for expressions, provide us an ability to precisely type this kind of code, because they can represent the substitution of types before they are determined.

Initially, we might believe that the argument and result types for `identity` might be different, and so require two type variables. In our example there may be two type variables – one to represent the type of the single argument, and another one to represent the return type of the function. So the function `identity` will be of a type[7]:

$$\delta := \texttt{identity}$$

$$\delta := \alpha \rightarrow \beta$$

But, examining the function's body, it is clear that both of these type variables will have the same type – the type of the argument `x`. There is a constraint implicit in this code, that the return type of this function must be the same as the type of the argument.
We write this as

[7] We will adopt the convention that type variables will be denoted by lower-case Greek letters: $\alpha, \beta, \gamma, \ldots$

$$\delta := \alpha \rightarrow \beta$$

and

$$\alpha \triangleq \beta$$

This requirement for types in different positions to be the same is an example of a typing constraint. There are two fundamental kinds of constraints. The first one is illustrated in the Figure 2.15, when one type variable is connected to another, yielding the same type for both – those are represented with the symbol $\triangleq$. There is another kind of constraint, when a type variable is bound to an expression[8], with a notation `:=`[9]. For example, looking at the piece of code

```
a="start"
```

we observe that the variable a has a type string, so the constraint will be

$$\alpha := \texttt{String},$$

where α is a type variable that represents the variable a.

Type inference has been implemented for some compiled programming languages without type annotations (for example, Haskell [24], ML [27, 28], and F# [22]). The most common implementation of type inference is the Hindley-Milner-Damas (HM(X)) algorithm [7]. A modern presentation of HM(X) comprises two stages – constraint generation, and constraint solving.

2.2.1 Constraint Generation

Given a program, the type inferencer walks the entire abstract syntax of a program and emits constraints based on the expressions of the program. For example, if we have a piece of code a+b, the type of the expression b is the same as the type of the expression

[8] Sometimes the type expression is self-describing: if we have an expression 3, it is easy to see that it is of integer type.

[9] There are constraints coming from self-identifying types too, but they can be resolved at once.

a, as both those variables are involved in the addition operation (which usually assumes that both operands must have the same type[10] or that the types are compatible).

Two types are compatible if either they are the same, or one of them is a subtype of another. For example, type `Number` is compatible to a type `Integer` or a type `Float`, but the types `Integer` and `Float` are not compatible with each other.

Constraints are generated by assignments, statements, bindings, and primitives. Below we give a schematic example of each, and associated constraints. The algorithm for constraint generation is described in pseudo code in Figure 2.16.

In the pseudo-code given in Figure 2.16 we showed the most important classes of expressions. Others are generated similarly to those shown.

2.2.2 Solving Type Constraints

The general approach to solve type constraints is Robinson's unification resolution algorithm [33]. It takes all the constraints and either

a) generates substitution for these constraints.

or

b) shows discrepancies between the constraints. This case means that the programmer made a type error or errors in his code.

Robinson developed the unification resolution algorithm [33]. Our presentation closely follows Krishnamurthy [18]. He describes the unification algorithm as shown in Figure 2.17. It starts with an empty substitution. Then all constraints are pushed onto a stack. After that the algorithm pops constraints off the stack one by one, and creates substitutions for each of them. This process is repeated until the stack is empty; in this case the algorithm returns the substitution.

[10] The technique to convert expressions of an unacceptable type to an acceptable one before an operation is applied is called *coercion* [2].

```
For each expression e, recursively in the code,
    if e is a variable reference to variable v,
        if v already has a type variable $\alpha$ assigned
        then emit $\alpha$ := e
        else
            generate a fresh type variable $\beta$
            emit $\beta$ := e
        end
    elseif e is a constant
        compute type t for constant
        generate a fresh type variable $\alpha$
        emit $\alpha$ := t
        emit $\alpha$ := e
    elseif e is a primitive of type t
        generate a fresh type variable $\alpha$
        emit $\alpha$ := t
        emit $\alpha$ := e
    elseif e is a application of $e_1$.f($e_2$)
        generate four fresh type variables: $\alpha,\beta,\gamma,\delta$
        emit $\alpha$ := $e_1$
        emit $\beta$ := $e_2$
        emit $\delta$ := f
        emit f := $\alpha,\beta \rightarrow \gamma$
        emit $\gamma$ := e
    elseif e is an assignment of $e_1$=$e_2$
        generate two fresh type variables: $\alpha,\beta$
        emit $\alpha$ := $e_1$
        emit $\beta$ := $e_2$
        emit $\alpha \triangleq \beta$
        emit $\beta$ := e
    # other cases are handled similarly
    end
```

Figure 2.16 Constraint generation algorithm

Substitution binds identifiers to constants or to other identifiers. In the end it contains the solution of the constraints, by looking up an expression in the environment yielding its type.

```
1) for all constraints, $C_i$
     push them onto a stack

2) while the stack of constraints not empty

     a) pop the constraint between X and Y off the stack

     b) if X and Y are the same type variable then
          continue with the next constraint

     c) else if X is a type variable then
            $S[X \leftarrow Y]$ (extend the substitution to bind Y to X)
            $C_i = C_i[Y/X]$ (replace Y with X in all constraints)[11]

     d) else if Y is a type variable then
            $S[Y \leftarrow X]$
            $C_i = C_i[X/Y]$

     e) else if X is an expression then
            # Y must be a type
            $S[X \leftarrow Y]$
            $C_i = C_i[Y/X]$

     f) if X is of the form $X_1 * . . . * X_n \rightarrow X_r$, and
             Y is of the form $Y_1 * . . . * Y_n \rightarrow Y_r$ then
            push constraints $X_i=Y_i$ (for each $1 \le i \le n$) on stack
            push constraint $X_r=Y_r$ on stack

     g) else # X and Y do not unify
          report an error and halt[12]

3) return the substitution S
```

Figure 2.17 Unification resolution algorithm

Consider the example which demonstrates how the algorithm works. Imagine that for a piece of code below we want to find the type of the overall expression.

(“string” + y)

[11] *Replace X with Y* means replacement of all occurrences of X by Y both on the stack of constraints and in the substitution. Note, that creation of a new variable does not destroy variables from other scopes with the same name, it just shadows them.

[12] In practice many implementations do not halt and continue the unification resolution algorithm to report all inconsistencies. Note, that a decision must be made of what type to use for inconsistent statements, which may lead to different error messages later.

It contains three expressions:

- a reference to a variable, y
- literal string, “string”
- the entire sum expression (“string” + y)

The algorithm generates the following constraints:

```
α := ″string″
β := y
γ := ( ″string″ + y)
δ := +
δ := α*β → γ
```

There is a collection of predetermined constraints that were defined in advance. Among them we had a constraint about the primitive + that

```
+ := x*x → x
```

After constraint generation, the algorithm pushes all the constraints onto the stack, and performs a resolution. In Figure 2.18 we showed the succession of steps.

Step	Stack	Substitution

2e)	α := ″string″
	β := y
	γ := (″string″+y)
	δ := +
	δ := α*β → γ
	+ := x*x → x	

2e)	β := y
	γ := (″string″+y)
	δ := +
	δ := ″string″*β → γ
	+ := x*x → x
		α := ″string″

2e)	γ := (″string″+y)
	δ := +
	δ := ″string″*y → γ
	+ := x*x → x
		α := ″string″
		β := y

2e)	δ := +
	δ := ″string″*y → (″string″+y)
	+ := x*x → x
		α := ″string″
		β := y
		γ := (″string″+y)

2e)	+ := ″string″*y → (″string″+y)
	+ := x*x → x
		α := ″string″
		β := y
		γ := (″string″+y)
		δ := +

2f)	"string"*y → ("string" + y) := x*x → x	α := "string"
		β := y
		γ := ("string"+y)
		δ := +
		+ := "string"*y → ("string"+y)

2f)	"string" * y → ("string"+y)
	:="string"*"string" → "string"
		α := "string"
		β := y
		γ := ("string"+y)
		δ := +
		+ := "string"*y → ("string"+y)
		x := "string"

Figure 2.18 Example of constraints resolution

At this step it is seen that the type of the expression is `String`.

The algorithm described in this section is simple and suitable for most cases. Unfortunately, it is not sufficient for interactive development as we show in the next section.

2.3 Principal Typings

The Hindley-Milner-Damas system described in the previous section has a useful property called *principal types* [7, 18]. That is, every expression is assigned its most general type – type that any other type is more specific. For example the type of the function `identity` in Figure 2.15 will be ***α → α.*** This function has an interesting type – it is polymorphic. Its type can be either `String → String`, or `Integer → Integer`, in general

`Any type α → Same type α`

depending on the argument provided when `identity` is called. A programmer will not be able to assign a more general type for the expression than that derived by the type inference system [4, 7, 17, 18]. That is, any type that could be assigned must be a *subtyp*e of the inferred, most general type: a type that would be consistent with it.

The essence of principal types can be seen in the following logic judgment [7, 17]:

$$\Gamma \vdash t : \tau$$

This sequent states that given a set of type assignments in the environment Γ, the expression t has most general type τ. The environment Γ contains prior assumptions that help determine the type of t. That is, when typing some expression t, the system looks into Γ and based on the information there, determines the most general type for t or produces error messages. Hence, Γ is an input to the type inference algorithm, usually in the form of constraints or substitutions.

Consider Figure 2.14 on page 26. Among others, we have the following constraint in the environment:

```
δ := div
δ := Number*Number → Number
```

Based on the line 2 of the figure, the system tries to determine the type of the following expression

```
r=1.div(a)
```

By constraint resolution, it types both variables `r` and `a` as `Numbers`.

Principal types identify the most general type for expressions, but the need for Γ as input means that when checking t, all dependent code (code that t relies on) must be available to the type inferencer. Hence the type inferencer can either annotate program in sequence or as a whole block at once. Note that this condition applies to the compiled languages; hence HM (X) is sufficient for them[13].

As Ruby is an interactive language, we need a different type system, because not all of the code will be immediately available.

Interactive programming requires a type system that would be able to type code in any order as well as to produce types for well-structured code fragments. Such a type system exists; it is called *principal typings*.

[13] Separate compilation is not a problem, as type inferencer can store type information of different sources in special files, and apply this information whenever necessary.

Despite the similarity of the name to the one of the previous algorithm, principal typings is fundamentally different from principal types. The fundamental difference is the input/output behavior of the typing algorithm. Recall that principal types take an environment and an expression as inputs to yield a type for that expression, as shown in Figure 2.19.

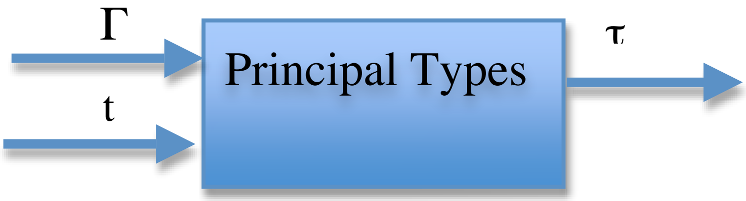

Figure 2.19. Behaviour of principal types system.

Principal typings only take the expression and gives an environment and the principal type of the expression [17].

$$t \vdash \Gamma, \tau$$

Figure 2.20 shows the behaviour of such systems.

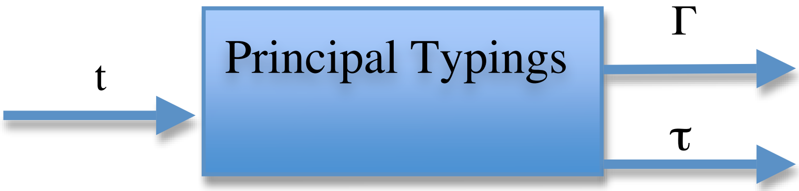

Figure 2.20. Behaviour of principal typings system

Typing of each expression creates a new environment containing restrictions and bounds that this expression requires from other code in order to be well-typed. As programs are interactively developed, these environments can be merged and compared to ensure that new code is compatible with old code. This allows a language with principal typings to infer types for the program code in any order. Also, it is not necessary to have the complete program, it is possible to type only well-formed fragments of the program and later to type-check their combination, emitting further restrictions or errors.

Consider an example of Ruby code where principal types property fails but principal typings infer correct types. The code in Figure 2.21 contains a function to compute factorials, and shows a use of that function later in the code.

```
1: def fact a
2:   if a==0
3:      1
4:   else
5:      a*fact(a-1)
6:   end
7: end

8: s=myToString (fact(7))
```

Figure 2.21. Use of a reference to a function still not written

After line 7, both principal types and principal typings will determine the type of the function `fact` to be `Integer` → `Integer`. But at line 8 systems that have only principal types will report an error that the function `myToString` is not defined yet (it is not present in the environment). A system with principal typings will just report that the expression `myToString` must have a type `Integer` → α. The principal typings system will generate the following environment:

Table 2.1 Environment generated at line 8 of Figure 2.21

Name	*type*
fact	: Integer → Integer
myToString	: Integer → α
s	: α

Later the system can check this environment against these constraints when `myToString` is actually defined.

```
9:  def myToString x
10:   x.to_s
11: end
```

Figure 2.22 Continuation of code in Figure 2.21

Now consider line 11 at Figure 2.22. During the function definition of `myToString`, principal typings will create another environment where `myToString` will be present.

Table 2.2 Environment generated at line 11 of Figure 2.22

Name	type
`myToString` :	β → `String`

Comparing this environment to the one in Table 1, there is no discrepancy:

$$\alpha := \texttt{Integer}$$

$$\beta := \texttt{String}$$

If there had been a discrepancy, then a type error describing the discrepancy would be reported.

This behaviour is what we need for Ruby because a programmer who uses interactive mode writes code such as function definitions that is not immediately executed and that uses some currently undefined expressions.

The principal typings algorithm can recognize and report precise type problems to a programmer. For example, consider that the programmer wrote the binary function definition in Figure 2.23 instead of that in Figure 2.22.

```
1: def myToString x,y
2:     x.to_s+y.to_s
3: end
```

Figure 2.23 Alternative implementation of `myToString` that needs two arguments

At this point, the programmer made an error: in Figure 2.18, the function `myToString` is called with only one argument, while the function definition expects two. A principal typings algorithm will create an environment for the second definition:

Table 2.3 Environment generated at line 11 of Figure 2.23

Name	type
`myToString` :	α*β→ `String`

In comparing environments, the principal typings algorithm detects this error, and can generate error message:

```
Warning: function myToStrong defined for 2 arguments,
previously called with 1.
```

The principal typing algorithm is also useful in other places. For example, separate compilation, where the whole system is divided into several modules that are compiled at different times, and each module may use external variables from other modules. Using principal typing we can infer the types of the program variables without forcing the programmer to specify the types of the external variables of the imported modules.

This algorithm is exactly what we need to develop the type checker for incremental development in Ruby. It is intended to work in such places where a principal types property would fail – in particular, for fragmented code.

Other researchers have considered typing for interactive languages. Their work will be shown in the next section.

2.4 Type Inference for Dynamic Languages

There were several projects trying to implement type inference for dynamic languages. Python is the most deeply studied, and we review the relevant work below[14]. It is important to note that none of the projects described below are focused on the interactive development approach supported by this thesis [3, 32, 41]. It is important to recognize that all projects listed in this subsection were developed primarily to improve performance. As a result, they do not need to type-check the entire language, in particular they do not need to cover complex cases, just common ones which offer greatest opportunity for performance enhancement. In our case we are considering interactive development, hence we need to cover all cases.

[14] We will defer discussion of the related work in Ruby – our target language – to the next section.

2.4.1 Psyco

Psyco [32], implemented by Armin Rigo, is a just-in-time compiler. It increases the performance of Python programs, up to a 40x depending on the particular application, almost to the speed of their C equivalent, but only for i386 processors. Psyco does this by finding locally-defined integers and strings that are unchanging from compile-time to runtime. Psyco substitutes the main eval loop of Python by its own, which can create several specialized versions of the machine code for different kinds of data: it is doing this by using the actual run-time data that the Python program manipulates. It works entirely at runtime, so it cannot be used for a static analysis.

2.4.2 Starkiller

Starkiller [41], by Salib, uses the Cartesian Product algorithm [1] to infer the types for Python source code. The type inference algorithm also handles data polymorphism in addition to parametric polymorphism, thus improving precision. Starkiller does almost complete type inference for Python avoiding only a few limitations like exception handling and reflection like the eval statement. However, in order to infer types, the complete modules must be provided, in contrast to our requirement to handle fragmented programs. One feature is an external type description language that enables extension programmers to document how foreign code interacts with Python. This enables Starkiller to analyze Python code that interacts with foreign code written in C, C++, or Fortran. Salib's primary aim was performance improvement, and numeric benchmarks show that Starkiller-compiled code performs almost as well as hand-coded C and substantially better than alternative Python compilers.

2.4.3 Brett Cannon's System

Brett Cannon's master's thesis [3] studies localized type-inference of atomic (our simple types) types in Python. Just as with our project, Cannon implemented a type inference algorithm without changing the semantics and syntax of the base language. He worked with Python, we work with Ruby. As with the other projects described here, he explored whether more type information at compile-time from type inference would improve Python execution performance. Unlike our goals, he was interested in using types for optimization purposes rather than to aid the programming process, so he did not consider interactive development and neglected complex type-checking cases. Unfortunately, he was unable to achieve a 5% performance improvement with his type inference.

2.5 Summary

In this chapter we gave a background of Ruby – a dynamic language that we use for our research. Next, as we presented in this chapter, types are an important concept in programming languages. Types and type checking ensure a better safety of applications. We described type inference, a way to type-check programs, and showed how principal typings can support interactive development. Also we discussed several projects that tried to add typing to dynamic languages. We next turn our attention to the challenges Ruby poses for type inference.

Chapter 3
Typing for Ruby

In the previous chapter we showed several benefits of Ruby language. Those benefits (cleanness, extendibility, portability and interactivity) ensure its popularity grows. But as we saw previously, the Ruby interpreter cannot assist in identifying coding issues in a timely way, so programmers require great discipline to benefit from incremental development style advantages described in the first chapter (focus on the code, early detection of errors, better program planning, and better control and understanding of problems). Some kinds of errors are reported only at runtime, as there is no type checking before code execution. To make incremental development effective, in other words, to get all the benefits that the incremental interactive development style offers, it is valuable to notify a programmer of type errors as early as possible. One approach to do it is to add typing to Ruby. Then type consistency will be checked much earlier, and errors may be reported earlier than those produced by the current interpreter. Moreover, the system with typing will be able to provide guidance and advices to programmers of how the types should be used.

There are several techniques to add typing to Ruby. One of them is to add optional static type annotations to the language[15]. For example, if a programmer wants to define a procedure that takes two integers as parameters, and returns their greatest common divisor, instead of the current version

```
def gcd (a, b)
```

he may need to write the code given next; she will have to indicate types for each argument (like in the example below type descriptions that follow **:**) as well as the return type for the function (type after the arrow symbol →)

[15] This approach was proposed for Python by the creator of that language Guido van Rossum [34].

```
def gcd (a: int, b: int) → int
```

Another possible syntax is the one shown below. In it, a programmer will have to declare the function (as in the example with the `decl` keyword) indicating the type of this function before actually defining it.

```
decl gcd: def (int,int) → int
def gcd (a, b)
```

Both these approaches have certain drawbacks. First, they take away the cleanness of the language – the code with annotations contains much more symbols. Second, these type annotations are optional; a programmer will not benefit from code lacking these annotations, for example, third-party packages and libraries. As a result, we believe that this approach is not suitable for our research.

3.1 Challenges for Typing Ruby

Type inference is another approach, but most efforts have concentrated on principal types rather than principal typings. For example, type-inferenced languages such as ML, F#, and Haskell, have compilers and cannot support interactive development fully. They do not require principal typings. On the other hand, Ruby has semantics that makes it much more difficult to apply type inference, and contains interactive features that do not allow one to infer the types for all valid code.

As the exemplar language for interactively-developed, dynamic languages, we concentrate on Ruby; but the techniques and challenges apply to other languages equally well.

Before providing a comprehensive list of situations of Ruby type-unsafe code, we provide four examples to demonstrate the impossibility to type-check all possible Ruby code. Some of these illustrate the concept of *slack* in a type system – code that executes correctly but violates the additional restrictions imposed by a type discipline.

3.1.1 Variables Shift Type

Ruby allows a programmer to make one variable store values of different types within one scope during different phases of development.

```
def foo(c=4)    #c is a number
    d=c
    c="hi"      #c is a string in the same scope
    ret=c
end
```

Figure 3.1. Example of slack

In Figure 3.1 the variable `c` is used in one scope to store different types: at the beginning of the function’s body it is an integer, while in the end it is a string. This complicates type inference[16] by not making it possible to know the type and sets of operations allowable for not only this variable, but also all other variables related to it (in our example, variables `d` and `ret`).

3.1.2 Branches of Control-flow Statements are not Required to be Type Consistent

Ruby permits inconsistent types for variety of things across different control paths. In Ruby, different branches of the control-flow statements may bind the same variable to values of different types.

```
if y>0
   x=2          #x is an integer
elsif y<0
   x="str"      #x is a string in the same scope
else
   x=2.3        #x is a float in the same scope
end
```

Figure 3.2 Example of slack for a conditional expression

The statement in Figure 3.2 is valid in Ruby, but it generates an unsolvable problem to the type inference system. After running this code, the variable `x` will have one of the

[16] There is a simple and clarifying technique to avoid this slack: fresh variables. Using newly-defined variable in place of `c` will avoid this dubious programming practice.

following types: Integer, Float or String. Without exact knowledge of the runtime value of the variable y it is impossible to determine the type of the variable x. Of course, any programmer relying on this polymorphism of x is setting himself up for failure as after the statement she will not be able to know how to use this variable, and whether her use of the variable is correct.

The problem arises not only with the conditional statements, but also with other control-flow statements. In the case of loops, a variable with some particular name may have different types in the different places of the loop. The unique issue is that before runtime it is usually not known how a loop will terminate if the body of this loop contains one of the following statements:

- break
- next
- redo

All of these statements change the control flow of the program. In the case of the break statement the loop is terminated in the place where this statement occurred, so after the body of the loop the variables will have the same types as they had at the point of the break statement. In the case of the next and redo statements the program skips all the code inside the body of the loop after that statement and starts a new iteration at the beginning of the loop. Unusual case is that if the next and redo statements occurred in the last iteration of the loop, there will be the same effect as with the break statement. Here is an example of the loop written in Ruby that shows these problems:

```
while i>0
  i-=1;
  c=4;                  # c is an integer
  if func1(c)>10
      break
  end
  if func2(c)>0:
       c="String"       # c is a string in the same scope
       next
  end
  c=[c,d]               # c is an array in the same scope
end
```

Figure 3.3 Example of slack for a loop

After this for loop that is present in Figure 3.3 the variable `c` can be either an array (if the loop terminates normally) or an integer (if the loop terminates by the `break` statement on the line 5) or a string (if the variable `d` was greater than zero in the last iteration of the loop and the `next` statement on the line 9 was executed). Again, having inconsistent results leads to potential future errors when type assumptions are violated: a dubious decision on the part of the programmer.

3.1.3 Ruby Exceptions Occur in Unpredictable Places

Exception handling is also an interesting case, as we do not know until the runtime which particular exception will be raised, as the body of the `try` statement can contain those statements that can raise more than one possible exception. Indeed, an exception may not be raised at all. Unlike Java, Ruby programmers do not indicate potential exceptions that methods might throw. Moreover, sometimes we do not know what exact statements of the `try` block can raise the particular exception, so we cannot draw the possible paths of the program execution. Consider this example shown in Figure 3.4.

```
begin
  eval string
  a=1              # a is a number
rescue SyntaxError, NameError => boom
  print "String doesn't compile: " + boom
  a="String"       # a is a string in the same scope
ensure
  print "Error running script: " + bang
  a=[]             # a is an array in the same scope
end
```

Figure 3.4 Example of slack for exceptions

It is uncertain which type the variable `a` will have after the execution of this block of code: it can be either an integer if no exception occurred, or either a string or an array depending on what particular exception was raised.

3.1.4 Reflection Constructs are Impossible to Type

In the previous chapter we briefly discussed a notion of reflection in Ruby, one example is a dynamic generation of code. As the code is generated dynamically, it is not available before runtime, thus it is impossible to type-check it before the code is run. Consider the example in Figure 3.5.

```
string1 = gets
x=eval string1
```

Figure 3.5 Example of Ruby's reflection

This example shows execution of code based on user input. Thus the code presents an impossible task for a type system to type-check it.

In summary, Ruby poses challenges for type inference and checking: many of the dubious constructs are confusing or error-prone. A discipline of type checking will help a programmer to write clearer and less problematic code, as well as next programmers who will be reusing the previous code.

Despite some difficulties that will not allow us to cover the complete language, we will be able to type many parts of Ruby and identifying type unsafe code, thus helping programmers in many cases. There are two basic cases possible for the code that is type unsafe:

- *errors* – cases where the programmer is not consistent
- *assumptions* – reliance on yet undefined code.

We will discuss both of these cases, showing possible situations, and explaining why these situations are erroneous or dangerous. We will start our discussion with errors.

3.2 Errors

Nine situations given below show a code that is problematic, but this will not become evident until runtime. Although in many of these cases a code is valid and acceptable by Ruby interpreter, each example is type-unsafe, and it is impossible to accurately identify the error.

3.2.1 Branches in control-flow Statements are not Type-consistent. As mentioned before, in order for programs to be type-safe, all branches of control-flow statements must be type-consistent. If they are not, the code is not type-safe. Here several cases are possible.

a. **Variable Created with Differing Types:** If a previously undefined variable is created with different types in different branches of a control-flow statement, then the programmer may not rely on the variable containing values of a known type after the conditional. In the example below, the variable `f` in the `then` branch will be an integer; in the `else` branch it will be a string. These are different types, leading to inaccurate programmer assumptions in the code depending on the result.

```
#  ‘f’ is not defined
if a>0
  f=9
else
  f="6"
end
#  ’f’ has ambiguous type
```

b. **Variable not Created in All Branches:** If a variable does not exist, and is created/assigned in only one branch, then an error occurs because the programmer cannot assume, after the conditional, that the variable exists and is initialized. Consider the following code, where the variable `d` is not yet created. The variable was created in one of the different branches of a control-flow statement, but not in the other, leading the programmer to unexpected results if they use `d`.

```
# neither ‘f’ or ‘d’ are defined
if a>0
  f=9
else
  d=6
end
# either ‘f’ or ‘d’ is created, not both
```

c. **Function Returning Different Types:** Functions can return from differ places in their body code, and hence may return values of different types. In these cases, the programmer cannot reliably depend on knowing the result type. This is illustrated in the example below. Based on the integer argument `arg`, the function diffReturn may return either an integer or a string, thus making this function not type safe.

```
def diffReturn arg
  if arg>0
    return 3  # integer return
  end
  return ""    # string return
end
# 'diffReturn' returns either an integer or a string
```

3.2.2 Local/Global/Instance Variable Changes Type: If a variable is already known to contain a value of a given type, and is later assigned a value of a differing type, the programmer may not know the variables type anymore. In the example below, the variable `b` was initialized with a string; later, in the body of the `while` statement, it is reassigned to contain an integer. As these are different types, this situation is unsafe.

```
b=""  # 'b' is a string
...
b=5   # 'b' is redefined as an integer
```

3.2.3 Number of Targets Does not Match Number of Values in Multiple Assignment: Ruby language does allow programmers to assign values to many variables at the same time by listing the necessary targets on the left of the = operator, and all the according values on the according positions on the right of the = operator. The number of targets should match the number of values except in unusual circumstances. If they don't, the statement may unintentionally assign wrong values to targets. In the example below the programmer assigns two values to three targets, and this situation appears to be unsound.

```
a,b,c=1,2  # type of 'c' is null
```

3.2.4 Inappropriate Use of Break, Redo, Next Statement: By Ruby specification, any of the following statements – `break`, `next`, `redo` – must strictly be used inside the loops. Any other usage of them is prohibited. In the example below the `break` statement is used outside any loop, which will cause an error when the function `funWithBadBreak` is run.

```
def funWithBadBreak
     break # illegal outside loop
end
```

3.2.5 Function Called with Wrong Parameters: Here several cases are possible: there may be an incorrect number of arguments, one or more arguments may be of the wrong type, or an expected block may be omitted.

a. First, we consider the case when a function is called with a wrong number of arguments. In the example below the built-in function `concat`, which expects one argument, but is called with two arguments.

```
“hi”.concat(“\n”,”.”) # wrong number of arguments
```

The same case applies to lambda procedure. In the example below the nullary lambda function stored in variable `a` is called with one argument instead of two, which is a type error.

```
a=lambda{2}
a.call(2) # wrong number of arguments
```

b. Second, a function may be called with arguments of the wrong types. In the example below, the built-in function `concat`, which expects one string argument, is called with one integer argument. In this case the function’s argument clashes with the function’s expected type, and this is an error.

```
“hi”.concat(3) # wrong type of the argument
```

c. Some functions expect to be applied to blocks, permitting co-routine execution via the `yield` statement. Neglecting to supply a block to one of these functions is an error. In the example below, the function `apply` must be provided with a block when called; the programmer omits the block, making her code fail.

```
def apply
  yield
end
apply  # needs block
```

3.2.6 Parameterized Types May Only Contain Values of a Single Type: Ruby supports parameterized types – containers of values of other types; for example, arrays, ranges (one value type) and hashes (two value types – key and value). In the example below, two hashes `h1` and `h2` used in the `update` operation clash: `h1` is a `Hash[Integer => String]`, and `h2` is a `Hash[String => Integer]`. This code is erroneous.

```
h1={1=>”one”}
h2={"two"=>2}
h1.update(h2)  # type clash
```

3.2.7 Classes, Modules, and Constants Redefined to Another: Once a name is bound to a class, it cannot be reassigned to a module, and vice versa. In the code below, a constant with the name `Aclass` was defined as a class. Later in the same body of code the programmer tries to redefine it as a module. Ruby does not permit this construction.

```
class AClass
...
end
...
module AClass  # redefine class as a module
...
end
```

3.2.8 Ordinary Functions Called as Class/Module Functions: Recall from the previous chapter that Ruby has a concept of class/module functions, ones that can be called with the name of their class/module as a receiver. Calling ordinary functions this way is not allowed. In the example below, the programmer calls the ordinary function `ordfun` as a class one. Ruby will report an error if the programmer tries to run the function `wrap`.

```
class A
  def ordfun
  end
end
def wrap
  A.ordfun  # ‘ordfun’ is not a class function
end
```

3.3 Situations containing assumptions

The situations below do not indicate that a programmer necessarily committed errors in her code or that she was inconsistent. However, they indicate that the code written so far is incomplete, i.e. in order to make the code be executable the programmer must correctly and consistently complete the remaining code. We provide five examples of such situations.

3.3.1 Use of Functions Before Declarations: Ruby allows programmers to reference functions before their declarations under the condition that before using the referencing code, the programmer will need to define the referenced function. In the following example, the function `gcd` is called inside the body of the function `lcm` before it is declared. If the programmer calls the function `lcm` right away, she will get an error; that is why this situation is potentially dangerous.

```
def lcm a,b
  a*b/gcd(a,b)  # ‘gcd’ not defined
end
```

Similar situations can occur if programmers use undefined functions for other purposes; for example, if they try to create duplicates using the `alias` keyword. In the example below, the programmer tries to make a duplicate of the function `euclid` that was not yet defined at that point. If she calls the function `duplicatingFun` she will get an error, as it is an error according to Ruby specifications to duplicate undefined functions.

```
# function ‘euclid’ not defined
def duplicatingFun
  alias gcd euclid
end
```

Recall that Ruby allows definitions to be forgotten. Un-defining a function that has not yet been declared is not allowed. In the example below, the programmer attempts exactly that – undefine the function `und` that was not defined at that point, and this situation is incorrect.

```
# function ‘und’ not defined
def undefiningFun
  undef und
end
```

3.3.2 A Global/Instance/Class Variable is Used Before Definition: Ruby allows programmers to use all kinds of variables except local ones before their declarations with the condition that before using the code that references these variables the programmer under need to define the missing variables. In the example below the programmer creates the function `geta` for the class `A` that relies on its instance variable, `@a`, which was not defined. Until the programmer defines this variable, she will not be able to run this function.

```
class A
  def geta
    @a  # ‘@a’ not defined[17]
  end
end
```

3.3.3 Reference to an Undefined Class/Constant: Ruby allows programmers to use in their code names of classes and constants that do not exist yet under the condition that before using that code the programmer will need to create them. In the example below, the function `callConst` relies on the constant `B` from the class `A`. If the class `A` is not visible, its constants will not be visible either.

[17] Recall that classes can be extended in other code sections: Ruby implements open classes, so `@a` may be defined later.

```
def callConst
  A::B # ‘A’ not visible
end
```

3.3.4 Functions are not Type-consistent: Principal types systems have constant environments, meaning that they do not allow programmers to change types of functions. Unlike them, principal typings systems allow programmers to change functions definitions and signatures. In the code below, the programmer changes the function `currentValue` to make it return string instead of integer. The problem is that the previously written code relies on the old definition of this function, so if the programmer tries to reference this code, she will fail.

```
def currentValue
3
end
...
currentValue +5 # ‘currentValue’ must return number
...
def currentValue
""
end
```

3.3.5 Definitions of functions with the Type Inconsistent to the One that it was Used Before: As we mentioned earlier, if a programmer wants to run a block of code that uses an undefined function, she needs to define that function, and this function must be able to accept the parameters that were provided, when it was called. In the example below, the parameters of the defined function `callLater` do not match those provided when the function was called, and this is a potential error.

```
def fun a
  callLater # ‘callLater’ must be without arguments
end
def callLater b # ‘callLater’ expects one argument
  b=""
end
```

3.4 Related Ruby-Typing Work

Several researchers investigated applying types to interactive languages, but none of them could solve the challenges tackled by our research – none of their systems were able to type fragmented code, or programs entered in independent order[18]. [11, 19, 23]

3.4.1 DRuby

Michael Furr et. al. [11] aimed to integrate static typing into Ruby. Their interpreter annotated Ruby code. In order to do that, they developed a new parser for the language. Also they created the Ruby Intermediate Language in order to translate the entire source language into this subset. To complete the system, researchers developed a type annotation language and a type inference system. Their system is called DRuby. They applied it to a suite of small benchmarks, and found that most of their benchmarks are statically typeable. Unlike our system, they altered the language syntax to support annotations; our work attempts to handle an unaltered syntax.

3.4.2 Kristensen's Master Thesis

Kristensen [19] accomplished another related work for his master's thesis in Aalborg University. The goal was to show that his Ecstatic tool can infer precise and accurate types for arbitrary Ruby programs. By implementing the Cartesian Product Algorithm he confirms that the algorithm can be retrofitted for a new language, as originally it was developed for the Self language. He was also able to devise a method for handling Ruby core and foreign code both implemented in C by utilizing RDoc—the embedded documentation generator for the Ruby programming language. Using Ecstatic, a number of experiments were performed that illuminated the degree of polymorphism employed in Ruby programs. The author also presents an approach for unit testing a type inference system. Again, unlike our work, his work focuses on precision and accuracy of complete programs, different from our focus on interactive coding.

[18] Here, i*ndependent order* means declarations given outside of a recursively-scoped complete module, or not in topological order of increasing dependency.

3.4.3 Duby

Charles Nutter developed a system called Duby [23], which uses type inference to help a Ruby compiler achieve better performance. The resulting system was nearly two orders of magnitude faster than the fastest JRuby production systems and at least an order of magnitude faster than the fastest incomplete, experimental implementations of Ruby. His aim was to investigate the performance issues of Ruby. As he says, the Duby benchmark result shows how fast a Ruby-like language can be. Nutter did not set a goal for his system to work for code fragments; our system must check fragmentary programs as they are interactively tested.

None of the projects mentioned above can be used to infer types in interactive mode; that is how this work differs from theirs.

3.4 Types in Ruby

Although Ruby is a dynamically typed language, it does not mean that it lacks a type system at all. Ruby still supports many types, but unlike those in Java or C, types in Ruby are determined at runtime. This section provides basic information of Ruby's latent types.

Ruby is a pure object-oriented language. That means that unlike many other popular languages, Ruby does not have a concept of primitive types – all Ruby types are based on classes. This, however, does not mean that Ruby does not support such popular in other languages types as integers or strings – variables, for example, can be of type integer or type string; but, in Ruby's case, they would be instances of built-in `Integer` class or `String` class respectively.

Figure 3.6 and Figure 3.7 contain a class hierarchy of Ruby basic built-in types. It was taken from [43]. In addition, there are function types that are not shown in this figure.

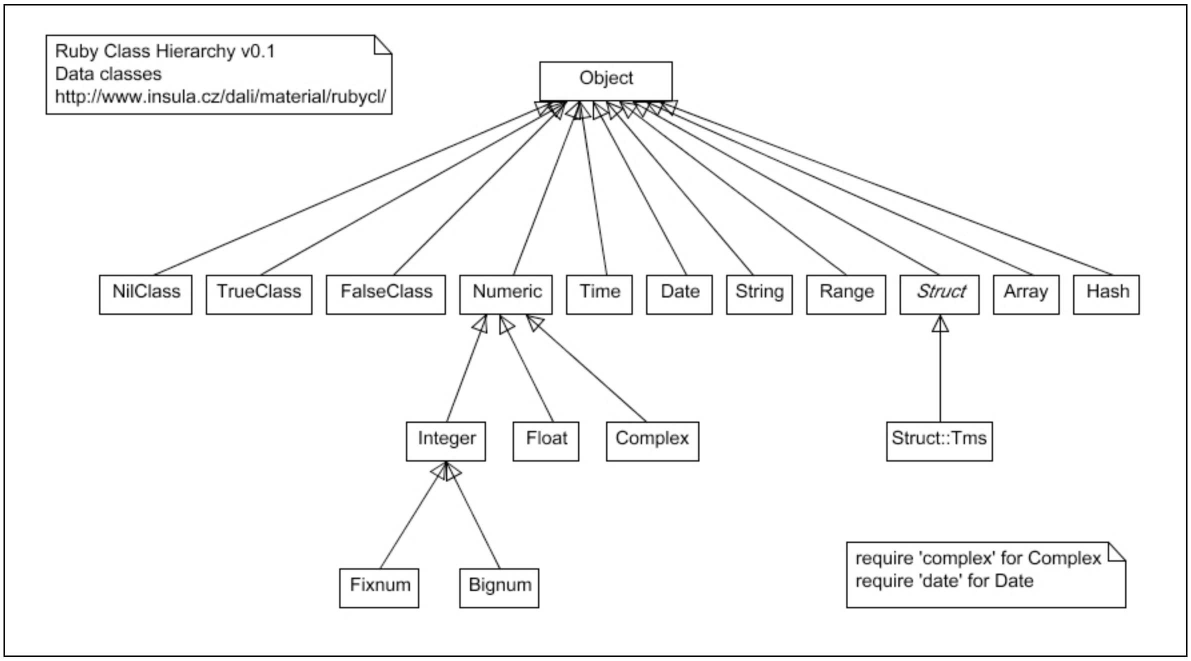

Figure 3.6 Ruby type hierarchy

These can be divided into three most important groups of types in Ruby. We called them *fixed types, unary types,* and *binary* types. In Figure 3.7 we provide the simplified diagram that focuses on the most interesting types for type inference. There are others (e.g., `Struct`), but they are not problematic for type inference because they are simple structures of types that we handle; we will concentrate on the types shown in Figure 3.7, as we believe they are most common Ruby platform types, most similar to popular types in other languages, and most suitable for our discussion. It is important to recognize that we will support user-defined classes, allowing us to manage third-party and directly developed code.

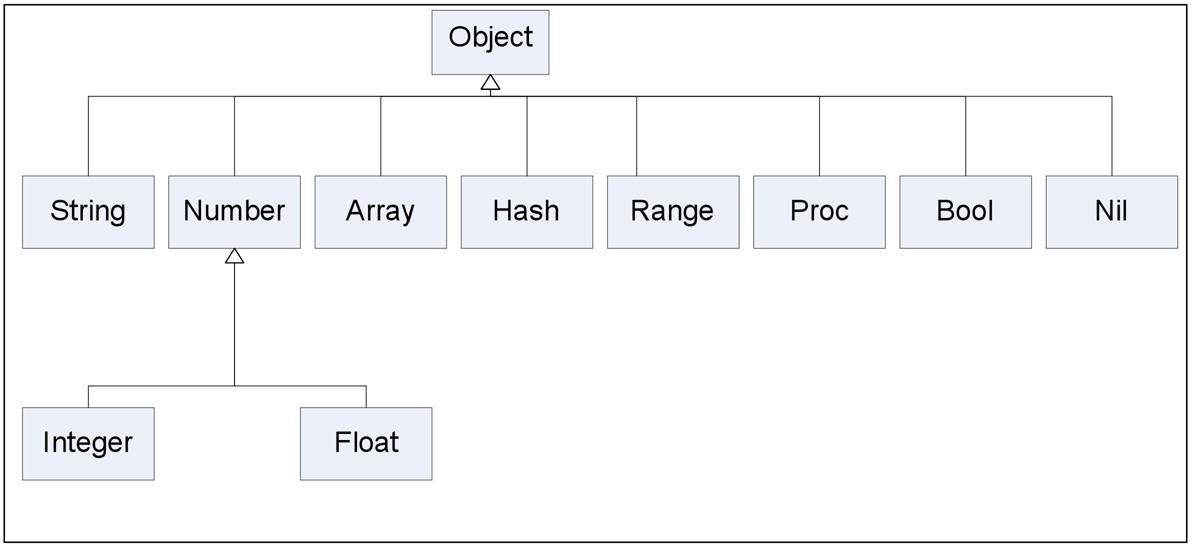

Figure 3.7 Simplified Ruby type hierarchy

3.4.1 Fixed Types

Fixed types are a subset of types that are primarily distinguished by the type of *one* component – basic value. For example, the difference between literals `“three”` and `3` is only value, thus both of these literals will have a fixed type. Most popular types – `String`, `Integer`, `Float` - are fixed types.

3.4.2 Container (parametric) Types

Those include unary and binary types. Unary types are a subset of types that are primarily distinguished by the type of one extra component – inner value – in addition to their basic type. Examples of them are `Array`s and `Range`s. For example, in order to be the same, two unary types must not only have the same basic value (for example, both be arrays), but also have the same inner values.

```
a=[1,2]
b=[“”]
```

Although both `a` and `b` are arrays, they are of different types because `a` is an array of integers, and `b` is an array of strings.

Binary types are a subset of types that are primarily distinguished by the type of two extra components – inner value and key value – in addition to their basic type. The most straightforward example of this kind of types are `Hash`es. For example, in order to be the same types, two binary types must have not only the same basic value (for example, both be hashes) and inner values (the condition that is sufficient for unary types), but also have the same key values. Consider this example

```
a={1=>2}
b=[“one” => 2]
```

Although both `a` and `b` are hashes and they both have the same inner types (integers), they are different types because `a` is a hash from integers to integers, and `b` is a hash from strings to integers.

3.5 Constraints for Ruby code

In the previous chapter we gave a general background of constraints. We explained their purpose in a general type-inference system in particular. In this subsection we give information about what constraints we needed specifically for Ruby. This gives a high-level description of the constraints arising from Ruby constructs. The details of our implementation are given in the next chapter.

There are basically two kinds of constraints arising in Ruby[19]:

- ones that bind two type variables together (represented by the symbol $\triangleq$)
- ones that bind expressions to their types/variables (symbol :=)

The first kind says that a type of the first variable must be the same as a type of the second one. This kind of constraints is generated, among other things, by the assignment operator “=”. An example of it is:

```
a=b
```

there is a constraint that the variable `a` must have the same type as the variable `b`. Hence the type variables for the expressions `a` and `b` must also be constrained to be equal.

The second kind of constraints are ones that bind expressions to their types. There are many possible Ruby expressions that can and must be classified as constraints, as

[19] Recall that there are constraints arising from self-identifying types as well, but as they can be resolved at once, they require no further discussion.

there are many possible kinds of expressions. We decided to divide constraints of the second kind further into the following three categories:

3.5.1. Function Constraints. *Constraints that represent calls of functions.*

In order to be solvable, constraints of this kind must contain the following information:

- name of the called function,
- number of arguments provided,
- types variables for the receiver and arguments[20], and
- type variable for the return value .

Consider the following example:

```
2+3
```

This piece of code creates one function constraint:

Name	"+"
Number of arguments	2
Type return	Tvret # *not determined*
Type - receiver	Tvrec `:=` int
Type - argument 1	Tvarg1 `:=` int

The constraint shown above was created for the method + with one receiver and one argument provided: values `2` and `3`.

3.5.2 Colon Node Constraints. *Constraints that represent colon nodes use.*

For example: `A::B::C::D` is the example of the colon node us.

Recall that colons represent nested scopes: each element with the name that is in the sequence must be an inner element for the element with name of the previous symbol in the sequence. In our example, the element with the name B (or to be more precise, a class with the name B) must be an inner class for the element (or the class) with the name A.

[20] Recall that all Ruby functions are actually methods.

Constraints of this kind will have a container that will store the sequence of called constants.

Container of constants	A ⊃ B ⊃ C ⊃ D

The constraint shown above was created for the method colon node with a vector of all constants provided, that are stored in the same order of their appearance in the code.

3.5.3 Creation of Singletons Constraints. *Constraints that represent creation of singleton functions*.

Remember that singleton functions are those that will exist only for one specific instance of a class. In the example below the programmer creates a singleton function `singFunc`.

```
a=String.new
…
def a.singFunc b
  3+b
end
```

After the code shown above, the variable `a`, that is a string, will have an access to the function `singFunc`. Other instances of the class `String` will not be able to see this function.

Constraints of this kind will have information about a defined function (`singFunc` in our example) and a variable that represents a receiver (`a`, that before the definition of `singFunc` was of the type `String`).

Receiver of singleton	Tvrec
Singleton function	`singFunc: Number → Number`

The singleton constraint shown above contains a type variable, that corresponds to the receiver variable `a`, and the function `singFunc` that will be accessible for the receiver.

3.6 Summary

Overall, types for Ruby are complex entities. They need to handle a variety of contentious cases, and identify correct and incorrect code precisely and clearly. Other’s have explored this topic, but not from the viewpoint of empowering interactive development. Hence, we have laid the groundwork in this chapter to explore how constraints and principal types can meet the challenges of typing Ruby code. In the next chapter we will talk about the implementation details of the system with principal typings for Ruby that is developed to meet the challenges of an interactive development.

Chapter 4
Implementation

In previous chapters we explored the benefits of type inference and reviewed a variety of challenges that Ruby poses for it. We identified an innovative type system, principal typings, that promises to support type inference for interactive Ruby programming. To demonstrate this facility, we implemented a type inference system for Ruby with principal typings. This chapter provides details of the developed system.

Our implemented system is called Rubin. It performs type-checking immediately after the abstract syntax tree of the Ruby code is constructed. The system extends the JRuby interpreter, jirb, version 1.1.2. It consists of 23 independent classes, which are located in the `org.jruby.ast` package (`rubin` subpackage), and 25 lines of Java code, which were inserted into two existing Java classes – `org.jruby.parser.ParserSupport` and `org.jruby.RubyNameError` – to provide the connection between jirb and Rubin.

The package, `org.jruby.ast`, which most of our implemented classes were inserted into, was designed to provide classes necessary for abstract syntax tree creation and analysis. This package contains descriptions of all possible nodes that the AST can contain. All nodes inherit from the basic abstract class `org.jruby.ast.Node`.

We chose to add our files into this package, as the purpose of them is the same as the purpose of the files in the package: they perform abstract tree analysis. Our files fit appropriately into the `org.jruby.ast` package, because they have the same functional requirements as those inside this package.

Our system contains 11 data types for type inference and 67 methods. Overall, Rubin takes approximately 10,000 lines of Java code[21]. This chapter summarizes the fundamental classes and operations of Rubin.

[21] Lines of code are measured using `wc`(1), and so include blank lines and comments.

The general diagram of how Rubin works is shown in Figure 4.1. Rubin steps in immediately after an abstract syntax tree is created and type-checks it.

1. Rubin performs constraint generation for the new code, creating a new table of constraints as an output.
2. Then Rubin starts a constraint resolution process, trying to resolve constraints from the new table with those imposed by tables from old code constructs. As conflicting constraints are recognized, Rubin reports error messages and warnings as necessary.
3. Last, Rubin returns to the normal interpretation stage.

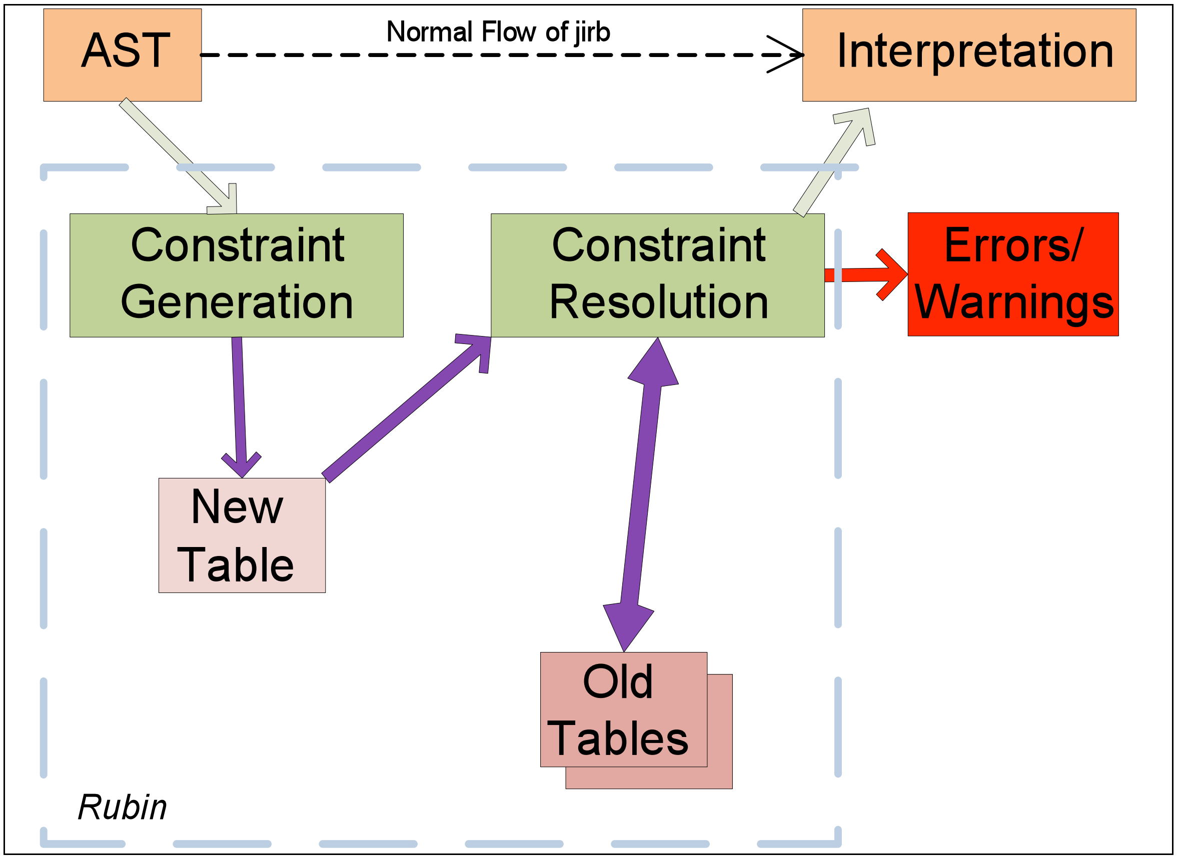

Figure 4.1 The diagram of how Rubin works

4.1 Data Types

We will start our description with the details of the data structures that implement Rubin. Most of these (except Constraint[22]) represent different Ruby types.

In this section we will discuss the purposes and implementation details of each of these. We will start our discussion with Type – the root type for all other type structures; then we will describe each of the nine subclasses of Type, each corresponding to a Rubin supported type. These are shown in Figure 4.2 using UML notation.

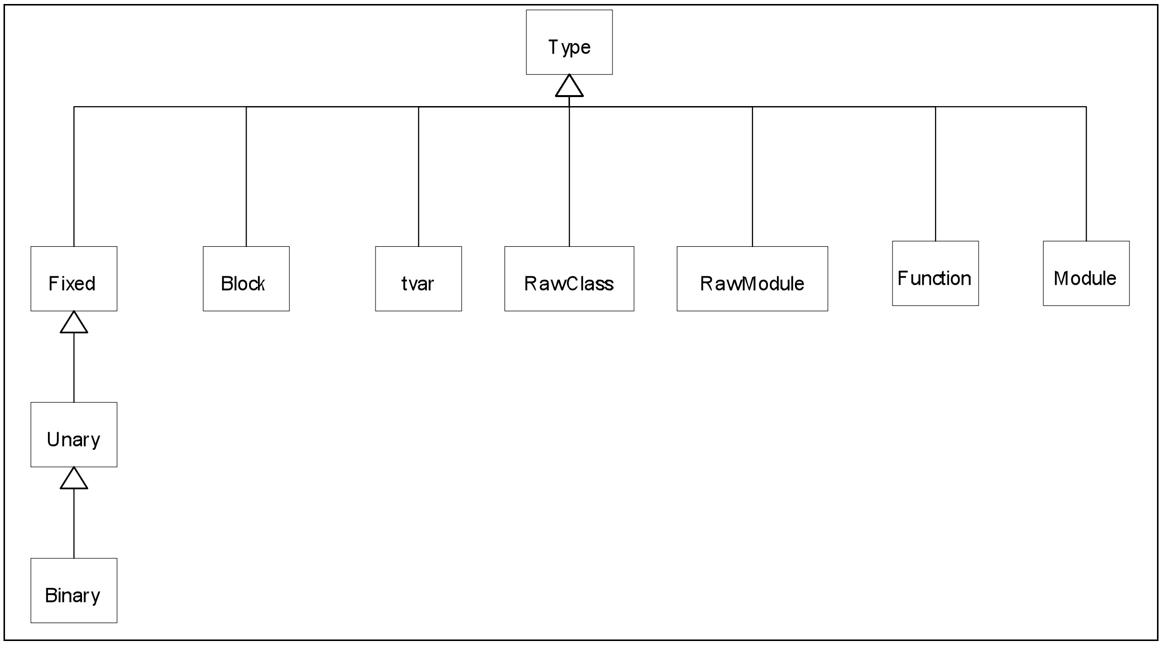

Figure 4.2 Hierarchy of Rubin's supported types

Additionally, we will describe the Constraint data structure, a special structure to represent the constraints required for type inference. This data structure does not represent any Ruby type, hence it is not shown in Figure 4.2. We will finish our discussion with the MainRubin class that connects all other data structures together and provides some additional utilities for them.

[22] We will use the Monaco font for Java code, reserving Courier fonts for Ruby code, just as we did earlier.

4.1.1 Type (org.jruby.ast.Type)

Every object that represents a type (either a type variable, or any kind of determined type – a final type (any determined type: predefined classes like Integer, String, Array, are automatically available), a function type, or an anonymous function type) is an instance of the abstract class Type, which defines general operations supported by all types. Among these operations are assign, the procedure for unifying one type to another type, and getType, the procedure to return a current type for final types, or the last type in a chain of bound types for type variables.

4.1.2 Fixed Type (org.jruby.ast.FixedType)

The goal of a type inferencer is to infer as many final types as possible. FixedType (FT) is a data type for describing final types such as String and Integer classes in Ruby, including those classes defined by Ruby programmers. In essence they represent values of classes.

Each fixed type has a name, that is stored in the field name and can be accessed with the method getName, and a super class, that is stored in the field superClass. This variable will contain Ruby's Object class if a Ruby class type does not have an explicit super class identified.

Each fixed type has several hash tables from Strings to Types. The table instVars stores all instance variables of FT, classVars stores all class variables of it, and constants stores all its constants. Each class can also have a scope of inner classes, classes, that are defined in the body of a current FT – they are stored in the table innerClasses. Similar structures for inner modules are stored in the table innerModules. The table methods is used to store all the methods of a given fixed type. There is an additional table to store singleton functions that is called singletonMethods.

Each of the hash tables supports the standard operation has that determines whether an essence with some name is present in the table, and the standard method get that returns a value for a given key from a table. As there may be several functions with one name for one class, the table methods has an additional method, numberOf, that returns a number of occurrences of functions with a provided name in that table.

Finally, each fixed type has a field, type, that gives information of the kind of type it represents. For example, the kind of type of 3 is FixedType (FT) (as 3 is an instance of the class Integer, which is a fixed type); the type of [3] is UnaryType (UT) and not FixedType (despite the fact that UnaryType is a subtype of FT); analogically, the type of {3 => "three"} is BinaryType (BT), and not UT or FT.

4.1.3 Unary Type (org.jruby.ast.UnaryType)

UnaryType (UT) is a data type for describing parametric types: types, that are distinguished not only by names but also by a single type parameter. The kind of type comprises such types as Range and Array, including those defined by programmers. Unary types are subclasses of fixed types that have the additional field, value, for describing that single parameter of parametric types, mentioned in the first sentence of this subsection. For example, Array [Integer] is not the same type as Array [String]: the field value of the former will contain a type Integer, while the same field of the latter will contain the type String.

4.1.4 Binary Type (org.jruby.ast.BinaryType)

BinaryType (BT) is a data type for describing binary parametric types: types, that are distinguished by names and *two* type parameters. This kind of type comprises such types as Hash, including those defined by programmers. Binary types are subclasses of unary types that have an additional field, key, for describing types of keys, those additional parameters that distinguish binary types from unary types. For example, Hash [Integer => String] is not the same type as Hash[Integer => Integer]: the field key of the former will contain a type String, while the same field of the latter, the type Integer .

4.1.5 Module (org.jruby.ast.ModuleType)

Module is a data type for representing modules defined by programmers in type inference constraints. This data structure is similar to FixedType, except that it does not have a field superClass. However, we chose Module not to be a superclass of FixedType, because they are logically different (modules, for example, can be mixed in inside classes, while classes cannot).

4.1.6 Raw Module (org.jruby.ast.RawModule)

Ruby raw modules describes modules as themselves, in contrast to instances of a Ruby module, that are represented by Module. Each raw class has a field, instanceModule, that describes the type for instance of that raw module, and a method getModule that returns that Java class.

4.1.7 Raw Class (org.jruby.ast.RawClass)

Raw Class is a data type for describing raw classes, or representations of Ruby classes as themselves, in contrast to instances of a Ruby class, that are represented by FIxedType. This is necessary to differentiate a piece of code like: A.foo from a piece of code like A.new.foo. In both cases A is the name of a class, but for the first case a receiver of the function foo is a Ruby class A, while for the second it is an object of the class A. Each raw class has a field, instanceType, that describes the type for instance of that raw class, and a method getFT that returns that Java class.

4.1.7 TypeVariable (org.jruby.ast.TypeVar)

The primary data structure for implementing principle typings is the TypeVariable (tvar) type structure. It contains the necessary information to enable Rubin to have initially unspecified types, and complete them at a later time. The completing type is stored in the inner field, which can be queried using the getInnerType method and updated with the setInnerType method. Updating is the key operation. If a type variable already has an assigned inner type, assignment is delegated to that inner type. Otherwise inner will be `null` before the assignment, and become set to the assigned type. The inner types for a chain of tvars may terminate at a type that is not a tvar. In this case, assignment is considered an error unless the assigned type and the inner type are compatible; as we mentioned earlier, two types are compatible if either they are the same, or one of them is a subtype of another. Otherwise the chain will terminate at another type variable. The last type in the chain of tvars that the current one is bound to, is called the *ultimate inner type*.

This data structure contains an additional method, `getType`, that returns the ultimate inner type for a type variable. It will be either another tvar when the type for this type variable is not determined, or a final type when the type for this type variable is known.

Type variables are fundamental data structures for type inference, as the overall process of type inference is nothing more than checking and determining tvars' type assignments.

4.1.8 Function (`org.jruby.ast.FunctionType`)

`Function` is a construct to represent function types, they can appear as right-hand side values of constraints. A name of a function is stored in the field `name`, and can be accessed by the method `getName`. The number of arguments is represented by the variable `NumOfArgs`. Each function has a vector vcTvars of type variables and a vector of constraints that are filled when the system evaluates the body of a function and generates both type variables and constraints. The information about any expected argument block is stored in `argblock`. By default this field is equal to `null` if a function does not expect any block. The boolean `isPrivate` indicates whether a function is set as `private`; the boolean `isModule`, in turn, indicates whether a function is defined as a module function, so it can be called with a module's name as a receiver. The field `isClass` provides a corresponding functionality for class functions – the functions that can be called with the name of their class as a receiver.

4.1.9 Block (org.jruby.ast.BlockType)

Block is a Java class for describing properties of blocks, including lambda-functions and Ruby procs. This data structure resembles Function, but, unlike functions, blocks cannot have another block provided when it is called, so Block does not contain an argblock field. Also blocks do not have the fields isModule, isClass, isPrivate, as blocks, unlike functions, cannot be class blocks, module blocks, or private blocks respectively. Other fields and methods are essentially similar to those of Function. Again like for Module and FixedType data structures, Block is not a superclass for Function, as they have substantial behavioural differences (blocks, for example, have access to the variables of the outer scope, while functions do not), and Ruby programmers do not consider blocks as superclasses of functions.

4.2 Managing Constraints

Now we are prepared to discuss a class that does not represent inner Ruby type. This fundamental data structure implemented for Rubin is the one to represent constraints, appropriately named Constraint (org.jruby.ast.Constraint).

Recall that there are the two kinds of constraints:

- ones that bind two type variables ($\triangleq$), and
- ones that bind expressions to tvars or final types (:=).

We never construct any constraints of the first kind, because we unify the two variables immediately. For this purpose we used the function unify; the exact process of unification is shown and described later in this chapter. Hence, our constraint structure needs to represent the second kind of constraint only.

Consider the second type of constraints. As we mentioned previously, there are many possible kinds of expressions that must be bound into constraints. For each of the different kinds of expressions, the Constraint data structure has matching fields and methods that provide necessary support for them.

Three special cases require careful presentation.

a) Function constraints

`Constraint` stores all type variables that represent types involved in it (a type of receiver, represented by a special type variable, `rec`, types of arguments, and return type) in a vector `vcTvars`. A type variable `rec` represents a receiver. `Constraint` also has a field, `NumOfArgs`, that stores the number of arguments provided for a function when this constraint was generated. This number gives the expected arity of the function.

We say that a function constraint is *explicit* if it was created for a function call with an explicit receiver Any function constraint that is not explicit is called *implicit*. For example:

```
a.foo      # explicit    rec == class of 'a'
foo        # implicit    rec == current class
```

b) Colon node constraints

If a `Constraint` deals with colon nodes, i.e. nested classes, it stores names of classes in an instance vector `vcConstants`. It relies on the `findClass` function from the `MainRubin` class (section 4.3.1) to search for a class or constant with a specific name in a given scope.

c) Singleton

When a method is defined for specific instances rather than an entire class, a special constraint must be constructed. It relies on the `setSingleton` function from the `MainRubin` class to change a receiver of a constraint, so it will still be an instance of a particular class, but with an access to more functions than other instances of the same class. A function to be added to the receiver class is stored in the `fun` field of `Constraint`.

Each constraint has a field isCol that if set to true indicates that this is a colon node constraint, isSingleton indicates that this is a singleton constraint. If both of these are set to false (the default), a constraint is a function constraint. Also Constraint data type has the field isSolvable, that is set to false only if Rubin determines that a constraint cannot be solved. For example, if the programmer called a non-existing function at the root level.

The Constraint structure is a necessary data type for a type inference: the type system keeps information in this structure of all the assumptions for a piece of Ruby code.

4.2.1 MainRubin class (org.jruby.ast.MainRubin)

MainRubin class is the class that binds all other Rubin classes together. It has access to many Rubin tables (environments) that store type information for the code. There are different tables for different scopes of the code, as well as different tables for different kinds of variables and functions: the method lookup provides a functionality to look into a current table, table for the scope of the code being currently evaluated, and all other accessible tables from the current scope.

Rubin relies on the class MainRubin to get many utilities necessary for type inference. The most important of the utility methods in this class are:

- lookup (String → piece) – searches for a class/function/variable with a given name in a current environment (table),
- putToTable (FT/function/variable → boolean) – puts a given class/function/variable into a current environment,
- removeFromTable (FT/function/variable → boolean) – removes a given class/function/variable from a current environment,
- setSingleton (Tvar, Function → boolean) – extends the type of a given type variables with a singleton function,
- setAsModuleFun (Function → boolean) – establishes a given function as a module function,

- `setAsClassFun` (`Function → boolean`) – establishes a given function as a class function.

As we mentioned before, 25 additional lines of Java code were inserted into two existing Java classes of jirb: method `switchModes()` to `org.jruby.RubyNameError`, and method `createRootTable()` to `org.jruby.parser.ParserSupport`. The first one turns Rubin on and off, and switches the mode, while the second one creates an initial table with predefined functions.

4.3 Algorithms

Next we describe the essential algorithms comprising Rubin. In particular, we will elaborate on the following algorithms:

1. an algorithm to generate constraints,
2. an algorithm to unify two types in general,
3. an algorithm to unify two final types,
4. an algorithm to solve constraints.

4.3.1 Generate constraints

We will start with the first fundamental procedure for the type inference, and it is the procedure of constraints generation. In order to generate constraints, the system analyzes each node in the abstract syntax tree, and creates constraints for each of them. These constraints will be solved later; with every new node at the root interaction level, the system tries to solve the newly-generated and previously-unsolved constraints.

The system behaves differently for each different node of an abstract syntax tree. If a node is an assignment node, the system binds the types of left and right hand expressions of that assignment. For conditionals and loops the system generates constraints for all nodes and generates type tables for all possible branches, and finally compares those tables. If it finds discrepancies, it emits an informative message to the programmer describing the problem. When a new entity, variable, function, class, module, is created, the system puts it into a corresponding scope. For example, if a function `innerFun` is created inside of the body of the function `outerFun`, then `innerFun` will be put into the scope of the function `outerFun`. When entities are referenced, the system searches them in the corresponding scopes, and if they are not there, or their types are not compatible, the system generates a message to a programmer.

The constraints are generated for function calls, singleton functions, and constant calls, whether names of a class/module, or ordinary constants that contain primitive types, etc.

The function `GenCns` generates constraints and inserts them into a list of constraints (`loc`), and returns the type of the checked node. It uses the function `Lookup` to search for a function/ class/module/variable in all accessible tables.

Below, in Figure 4.3 and Figure 4.4, is pseudo-code showing how the system generates constraints and merges tables.

```
GenCns (AST_NODE node): type
     bodytype = type of structure the node is located in[23]
     IF node ∈ ReturnNode THEN
          IF bodytype ∈ function THEN
              Unify (bodytype.return, GenCns (node.value))
              RETURN bodytype.return
          ELSE
              Report an error # 'return' cannot be called for class/module
              RETURN null[24]
          ENDIF
     ELSIF node ∈ Assignment THEN
          t = generate fresh tvar
          Unify (t, GenCns (node.value))
          Unify (t, GenCns (node.left)) # immediate constraint solve
          IF Lookup (node.left.name)!=null THEN
               Unify (Lookup (node.left.name), t)
          ELSE
               PutToTable (node.left.name → t)
          ENDIF
          RETURN Lookup (node.left.name)
     ELSIF node ∈ VarNode THEN
          RETURN Lookup (node.name)
     ELSIF node ∈ PrimitiveNode[25] THEN
          RETURN Lookup (node.name)
     ELSIF node ∈ AliasNode THEN
          IF Lookup (node.second.name)!=null THEN
               PutToTable (node.first.name →
                                   Duplicate(node.second))
```

Figure 4.3 Constraints Generation Algorithm (continued on the next page)

[23] This kind is either a function or a class/module.

[24] `null` is an instance of the fixed type.

[25] `PrimitiveNode` includes any of the primitive types – `Integer`, `String`, etc.

```
          ELSE
               Report a warning # the aliased function not defined
          ENDIF
          RETURN null
     ELSIF node ∈ UndefNode THEN
          IF Lookup (node.name)!=null THEN
               RemoveFromTable (node.name)
          ELSE
               report a warning # referenced function not defined
          ENDIF
          RETURN null
     ELSIF node ∈ Condition || node ∈ Loop THEN
          Vector tablesToCompare;
          FOR each branch
            FOR each node n ∈ branch
              GenCns (n)
            ENDFOR
            tablesToCompare.add (current table)
          ENDFOR
          table.put(CompareTables (tablesToCompare))
     ELSIF node ∈ FunctionCall || node ∈ ColonNode
      || node ∈ SingletonNode THEN
          loc.put (GenCns (node))
     ELSIF node ∈ Declaration[26] THEN
          FOR each node n in the declaration
            GenCns (n)
          ENDFOR
          putToTable (node)
     ENDIF
END
```

Figure 4.3 Constraints Generation Algorithm (continued from the prev. page)

[26] Declarations can be of functions, classes, or modules.

```
CompareTables (Vector tables): Table
  table1= tables[0]
  FOR each table2 in tables
     FOR each c $\in$ table1
        IF c $\notin$ table2
            Report an error  # branches not compatible
        ENDIF
     ENDFOR
     FOR each c $\in$ table2
        IF c $\notin$ table1
            Report an error  # branches not compatible
            IF !table1.contains(c.name)
               table1+=c
            ENDIF
        ENDIF
     ENDFOR
    table1.merge(table2)  # all elements from table2 not present in table1
                            will be added to table 1
  ENDFOR
  RETURN table1
END
```

Figure 4.4 Compare type tables of different branches of control-flow statements

4.3.2 Unify Two Types

Unification is the algorithm for type-checking and inference. Below we describe the unification of two instances of `Type`. The purpose of the algorithm is to make sure that the two types are compatible. This algorithm generates substitution, described in Chapter 3. Substitution is handled automatically, because if unification was correct and did not encounter any errors, both type variables involved in the unification will point at the same type.

In Figure 4.5, we give pseudo-code for unifying two types – `A` and `B`, that are given as parameters. If both `A` and `B` are final types, then the procedure `unifyFinalTypes`, that deals with the unification of two fixed types, is called. It checks whether these two types are compatible, as described later.

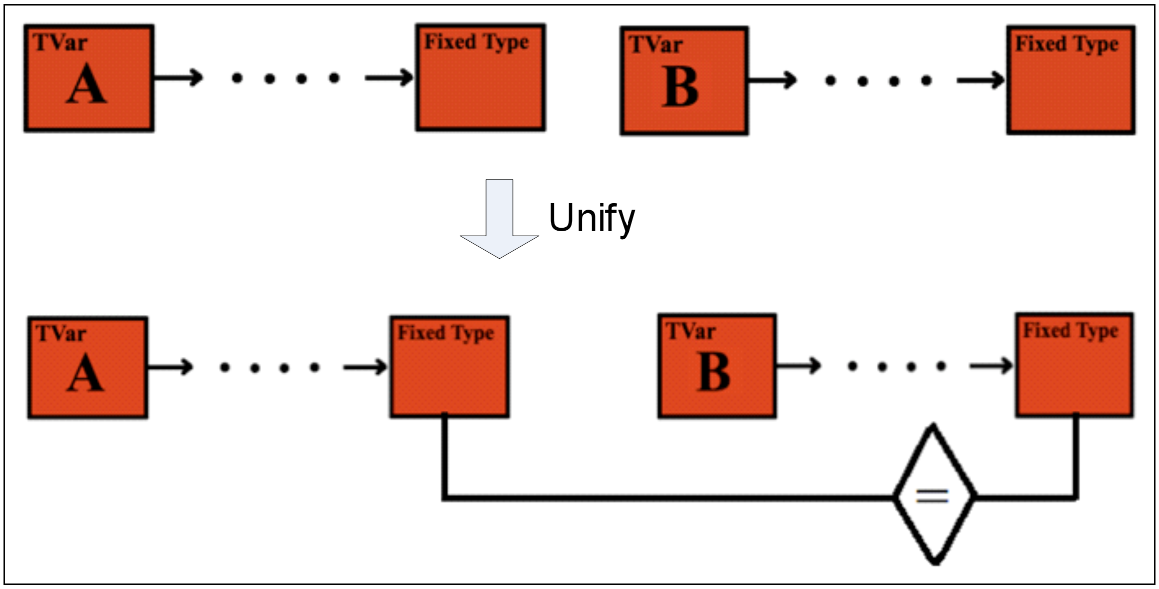

Figure 4.5 Unification of two final types

If one of the inputs' (`A` or `B`) ultimate inner types is a final type, and the other's is a type variable, then the former will become the inner type for the latter. This case is shown in Figure 4.6.

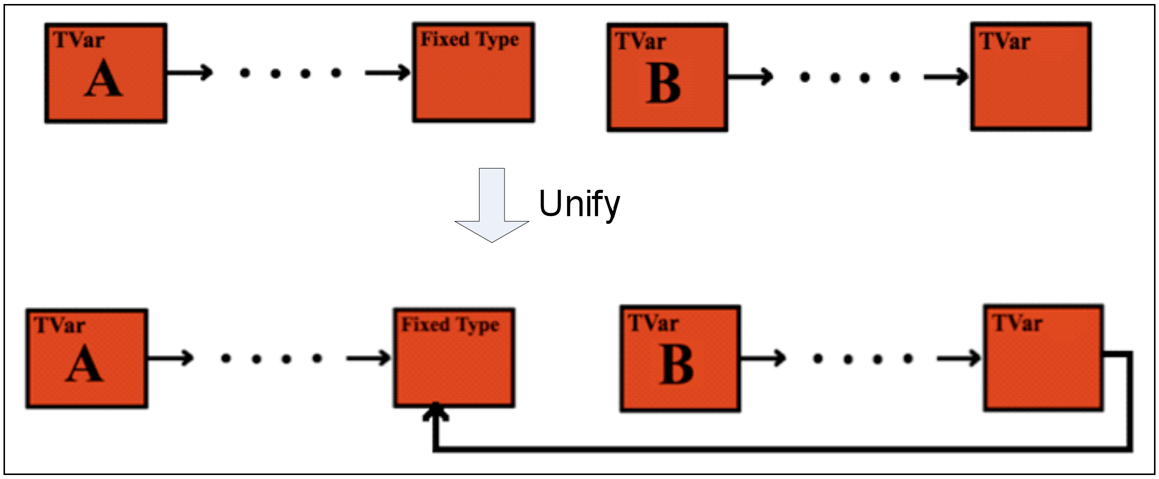

Figure 4.6 Unification of final type and a type variable

If the ultimate inner types of both parameters are type variables, then one will become an inner type for another, as it shown in Figure 4.7.

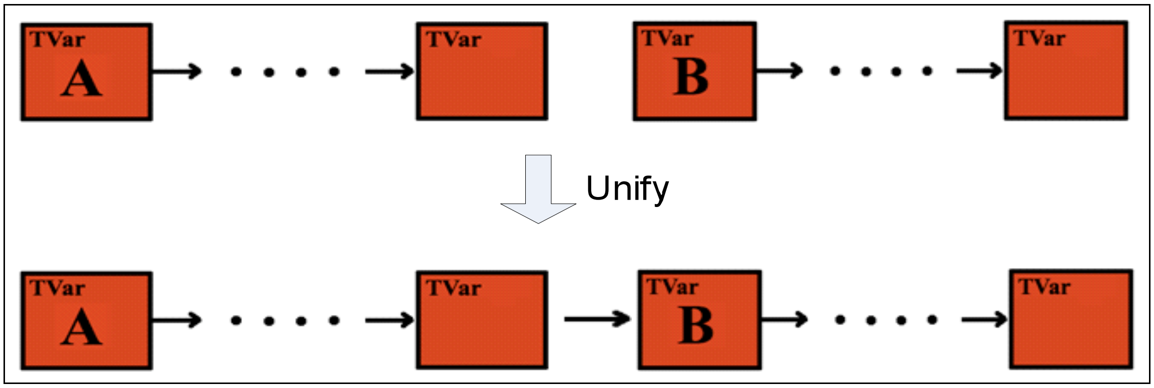

Figure 4.7 Unification two type variables

In pseudo-code the overall unification algorithm is given in Figure 4.8.

```
Unify (Type A, Type B): boolean
     IF A ∈ TV THEN
        IF A.inner == null THEN
           A.inner = B
           RETURN true
        ELSE
           C = A.getType()
           IF C.inner == null THEN
               C.inner = B  #(put B on the place of C, so A.inner == B
           ELSE
               RETURN Unify (C, B)
           ENDIF
        ENDIF
     ELSE  # (A is a final type) - A ∈FT
        IF B ∈ TV THEN
          RETURN Unify (B, A)
        ELSE
          RETURN UnifyFixedTypes (A, B)
        ENDIF
     ENDIF
END
```

Figure 4.8 Two types unification algorithm

4.3.3 Unify Two Fixed Types

Sometimes it is necessary to unify two final (already determined) types. For example, if two expressions have determined types, and they are used in an operation that requires them to have equal types, those final types must be unified. The problem is that, as we discussed earlier, final types can be different – fixed, unary, and binary – and each of them requires a different unification procedure.

The procedure `UnifyFinalTypes` attempts to unify two fixed types.

- If those two types are of different kinds, e.g. one of them is fixed, and another one is unary type, then the types are not compatible and the system returns an error.
- If both types are fixed, then if both types have the same name, then they are the same and therefore compatible; otherwise an error is returned.
- If the names are different, the types still can be compatible in the case when the first provided type is a subtype of the second. If it is not, then the types are not compatible.

When unifying two unary types, the sequence of actions is the same, but as unary types are additionally characterized by their inner types, those inner types must also be compatible. Comparison of two binary types poses another restriction: both key types must have compatible types. These algorithms are displayed in Figures 4.9-4.12.

```
UnifyFixedTypes  (Type A, Type B): boolean
  IF A.type != B.type THEN
     Report an error # Error, types not compatible
     RETURN FALSE
  ELSIF A.type ∈ FT THEN # are primitive types
     RETURN SolveFixedType (A, B)
  ELSIF A.type ∈ UT THEN
     RETURN SolveUnaryType (A, B)
  ELSIF A.type ∈ BT THEN
     RETURN SolveBinaryType (A, B)
  ENDIF
END
```

Figure 4.9 Two fixed types unification

```
SolveFixedType (FixedType A, FixedType B):boolean
 RETURN ((A.name==B.name) || (A.type is subtype of B.type))
END
```

Figure 4.10 Two primitive types unification

```
SolveUnaryType (UnaryType A, UnaryType B):boolean
  RETURN (SolveFixedType(A,B) && Unify(A.value,B.value))
END
```

Figure 4.11 Two unary types unification

```
SolveBinaryType (BinaryType A, BinaryType B):boolean
  RETURN (SolveUnaryType(A,B) && Unify(A.key, B.key))
END
```

Figure 4.12 Two binary types unification

4.3.4 Solve Constraints

After each new AST node is created at the root level, Rubin tries to solve both new constraints generated for the new node, and re-examine the old ones that were not solved previously. All constraints are stored in the list of constraints. During each stage of constraint resolution, Rubin examines constraints in this list, one by one. If any of the constraints can be solved, Rubin solves it, deletes it from the list, and starts new examinations of constraints from the list from the beginning, as solving one constraint could provide a new information necessary to solve some other, previously unsolvable, constraints, that could not be solved before. If Rubin determines that the constraint is never solvable, it reports a typing-error message, deletes this constraint from the list, and continues the current iteration of walking through the list. This process is repeated, until:

- no constraint was solved, or
- the list of constraints is empty.

In the first case the list is retained for the future analysis; the second case implies that all created constraints are solved.

In Figure 4.13, we give pseudo-code for the described procedure.

```
SolveConstraints (<Constraints> list): void
  FOR each constraint ∈ list
    IF SolveConstraint (constraint)
      delete constraint from the list
      restart the loop
    ELSIF !constraint.isSolvable
      report a typing error
      delete constraint from the list
    ENDIF
  ENDFOR
END
```

Figure 4.13 Solve constraints algorithm

Recall from section 4.2.1 that this constraint is either recognized as unsolvable if it was created at the root level, or the system defers resolution of this constraint until more information becomes available by adding it to a queue of the defined constraints. The function `FalseConstr` given in Figure 4.17 is provides this functionality by setting the flag to `false`.

4.3.5 Solve a Constraint

Our next algorithm will allow us to solve one constraint. A procedure to solve a single constraint is one of the fundamental parts of the type inference process.

As there are three kinds of constraint, the system first determines the kind of constraint to be solved.

If it is a *colon node constraint*, Rubin searches for all constants mentioned during the call of the colon node. If Rubin finds them all in corresponding scopes, it gets the type of the last one in the list, and unifies this type to the left-hand side type variable of the constraint. If Rubin cannot find at least one of the given constants, it cannot solve that constraint.

If the constraint is a *singleton constraint* for an already determined type, Rubin creates a new type for its receiver, that becomes a subtype of the previous one, by adding the newly defined function to the table of that class. If the type of the receiver is not determined, Rubin postpones the resolution of this constraint by continuing on to the next constraint.

Function constraints are the most complicated kind to solve. Before starting a process of a function constraint solution, Rubin examines the receiver of the constraint. If the receiver of the constraint is implicit, then initially the system searches for a function with the matching name up through nested scopes until it reaches the scope of the enclosing class of a current scope, or it reaches the root scope (`Object`). If the final type for the receiver is not determined yet, i.e. the ultimate inner type for the receiver is a type variable, then this constraint cannot be solved yet, and its solution is postponed until later. If the system has found the function with that name, it tries to determine whether the constraint is valid (whether the usage implied by the constraint is compatible with that function).

If the name of the constraint (that corresponds to the name of the function that this constraint represents) is equal to `new`, then this node is a creation of the instance for a class (a constructor): an initialization constraint. The receiver must be a raw class (for example `A::B.new`, not `a=A::B; a.new`). If Rubin shows that the receiver is not a raw class, it returns an error that the function `new` was called with the wrong receiver. Otherwise Rubin gets the matching class in the inner field of the raw class.

At this point the actual process of solving a constraint begins. Each class may have only one function of a given name, except for built-in functions. So the system checks for the function with the desired name in the tables of the receiver. If the function is not present anywhere, then the constraint cannot be solved and the solution is deferred. Alternatively Rubin reports a typing error if this constraint was created at the root level. If the function is present, and there is only one possibility, then the system tries to unify the matching types of the constraint and the function. If it cannot, then an error has been found; the constraint is not satisfiable.

If there is more than one possible function, then it is a built-in primitive. In this case Rubin needs to find the suitable function by sieving the possibilities based on the information given in the constraint:

- whether a block is necessary and whether it was provided,
- number of arguments, and
- types of each argument.

After each round of sieving, the system checks the number of possibilities left. If zero possibilities remain, none of the possibilities is suitable, and the constraint cannot be solved. If only one possibility is left, then the system tries to unify the constraint to that sole function. If more than one remains, then sieving continues. If in the end more that one possibility is left, the information given for the constraint resolution system was not sufficient to determine which possibility to use. In this case that system tries to find common types from all possible functions remaining. For example, consider the function + with the receiver that is an integer the two following possibilities remained:

```
+: Integer*Integer → Integer
+: Integer*Float → Float
```

In this case, the system is able to recognize that the argument must be of the type `Number`, and the return type also must be of the `Number` type, as both float and integer are numbers. So it unifies the matching type variable in the constraint to the common type. The system is always able to find a common type, as in Ruby every type is an instance of `Object`. Rubin is able to determine this type, but being an object does not provide any additional typing information.

In Figure 4.14 we give pseudo-code for this algorithm.

```
SolveConstraint (Constraint C):boolean
   IF C is colon node THEN
     RETURN SolveColonNode (C)
   ELSIF C is singleton THEN
     RETURN SolveSingleton (C)
   ELSIF C is implicit THEN
     RETURN SolveImplicit (C)
   ENDIF
   # The constraint is a function constraint
   A = C.rec.getType
   IF A ∈tvar THEN  # final type of receiver not determined yet
      RETURN FALSE  # Constraint cannot be solved yet
   ENDIF
   IF C ∈InitConstraint THEN  #  the name of the function is new
      RETURN SolveInitConstraint (C,A)
   ENDIF
   IF A.funs.has (C.name) THEN
      ResTable = SievePossibleFuns (C, A)
      n = ResTable.funs.numberOf (C.name)
      IF n > 1 THEN
        RETURN SearchCommonTypes (C,ResTable)
      ELSIF n == 0 THEN
        RETURN FalseConstr (C)
      ELSE
        RETURN UnifyCnstrToFun (C, A.funs.get (C.name))
      ENDIF
   ELSE
      RETURN FalseConstr (C)
   ENDIF
END
```

Figure 4.14 Constraint resolution algorithm

```
SolveColonNode (Constraint c): boolean
  FOR each name in C.vcConstants
    Find the necessary scope
    IF (Lookup (name)==null) THEN
       RETURN FalseConstr (C)
     ELSE
       RETURN true
     ENDIF
  ENDFOR
END
```
Figure 4.15 Colon node constraint resolution algorithm

```
SolveSingleton(Constraint C): boolean
  IF (Lookup (C.rec.name)==null) THEN
    RETURN FalseConstr (C)
  ELSE
    setSingleton (C.rec, C.fun)
    RETURN true
  ENDIF
END
```
Figure 4.16 Singleton constraint resolution algorithm

```
FalseConstr (Constraint C): boolean
  Report an error # one of the typing errors
  IF (C created at the root level) THEN
     C.isSolvable = false
  ENDIF
  RETURN false
END
```
Figure 4.17 Algorithm handle unsolvable constraint

```
SolveImplicit (Constraint C):Boolean
 F = table.get (C.NAME)
 IF F != null THEN
   RETURN UnifyCnstrToFun (C, F)
 ELSE
   RETURN FalseConstr (C) # Constraint cannot be solved yet
 ENDIF
END
```

Figure 4.18 Solve function constraint with an implicit receiver

```
SolveInitConstraint (Constraint C,tvar A):boolean
   IF A ∈RC THEN
      A = A.getFT
      RETURN true
   ELSE
      RETURN FalseConstr (C)
   ENDIF
END
```

Figure 4.19 Solve initialization constraint

```
SearchCommonTypes (Constraint C, Table table): Type
  FOR i = 1 to C.NumOfTvars
    Type t = table.ElementAt(0).ElementAt(i)
    FOR each possible function pf in table
      t = FindCommonTypeTwoTypes(t,pf.ElementAt(i))
    ENDFOR
    Unify (C.tvars(i), t)
  ENDFOR
  RETURN t
END
```

Figure 4.20 Search common type for the two types

```
SievePossibleFuns (Constraint C, Name A):
  table = A.funs.subtable (C.name)
  FOR all elements in table
    IF element.numberOfArgs != C.numberOfArgs THEN
        remove element from table
    ENDIF
    FOR each tvars ∈ C
      IF !Unify (tvar, matching[27] tvar in C) THEN
        remove element from table
      ENDIF
    ENDFOR
    IF !(element.argblock compatible with C.argblock) THEN
        remove element from table
    ENDIF
  ENDFOR
  RETURN table
END
```

Figure 4.21 Algorithm to sieve functions

At this point another important algorithm remains to be examined: the algorithm that unifies according types of a function and a constraint. When the system determines which function was called and finds this function in the table, Rubin must ensure that the use of this function is compatible with its description: the number of arguments is as specified, the used types are correct, and so on. This procedure works in the following way: a function and a constraint (or two blocks) given as arguments are considered compatible, if:

- their numbers of arguments are equal
- each argument is compatible with the matching function parameter’s type
- their argument blocks are not compatible

[27] Specifically, the return type of a function must be compatible with the return type of a constraint; the same for the receiver and all arguments.

The pseudo-code for this function is given in Figure 4.22.

```
UnifyCnstrToFun (Constraint C, Function F)
  IF F.numberOfArgs != C.numberOfArgs THEN
          RETURN FALSE
  ENDIF
     FOR each tvar ∈ C
          IF !Unify (tvar, matching tvar in C) THEN
               RETURN FALSE
          ENDIF
     ENDFOR
  ENDFOR
  IF !(element.argblock compatible with C.argblock) THEN
          RETURN FALSE
  ENDIF
  RETURN TRUE
END
```

Figure 4.22 Constraints and functions unification algorithm

The function `UnifyCnstrToFun` works for duplicates of the unified function and constraint, that is why the situation of mistakenly unifying a few tvars before realizing that the function does not correspond the constraint is not possible.

4.4 Summary

This chapter detailed our implementation of Rubin, our system that implements type inference for Ruby. In particular, we described data structures that are supported by Rubin, including

- Type,
- Fixed Type,
- Unary Type,
- Binary Type,
- Module,

- Type Variable,
- Function Type,
- Block, and
- Raw Class/Module.

Also we provided the descriptions of the fundamental algorithms of our system with principal typings. The most important of these are constraint generation and constraint resolution.

Next we turn our attention to validating our system, and showing its abilities. We will show how the system works with the potentially problematic Ruby constructs previously identified, and how it is able to improve the software development process. Also we will discuss several examples of the actual Ruby projects, and how Rubin could help programmers in developing those projects.

Chapter 5
Evaluation

The system was designed in order to improve Ruby interpretation mode described previously in this thesis. Here we provide a brief summary of the system's tasks:

- Provide error messages *earlier* in the development process than jirb does
- Provide error messages *at a better localized, more expected place* than jirb does
- Provide *more meaningful messages* than jirb does

This chapter evaluates the extent to which these goals are met.

There are several ways we can validate the system:

- Show that the subset of Ruby situations, for which error messages will be reported earlier by Rubin, is bigger than it was for jirb.
- Show that the system can improve developing of actual Ruby projects.
- Conduct a user study

As a result, we decided to perform several types of evaluation because it will better demonstrate the utility of the system from different angles.

First, we will show benefits provided by the system – we will examine all of the challenges described earlier, and show how Rubin reduces the difficulties. At this stage of evaluation we will also discuss how Rubin improves interactive/incremental Ruby coding. We will show what our system with principal typings can do that others lacking this property cannot.

Second, we will show several examples of buggy code in actual Ruby projects, and how the system like ours could prevent these bugs from remaining in the project. We will provide three examples. The purpose of that subsection is to show that the system like ours can be useful for real world applications.

5.1 Challenges

We show here the way the system works with different Ruby constructs and different coding situations where potential type problems may occur without warnings from jirb. We give the messages Rubin produces, and compare these messages to those of jirb.

As we discussed earlier, problems may show up for specific Ruby constructs, given in Chapter 2. Below, we list these constructs, and show how Rubin deals with each of them.

In each of the cases, Rubin will report a potentially unsafe situation to the programmer:

- **immediately after the construct is written**, if the code is being written on the root level of the interaction window, or
- **immediately after the programmer gets back to the root level**, if the code is not being written on the root level of the interaction window.

In addition, Rubin recognizes a number of different type-clashes, which can lead to later programmer confusion. Some of these are recognized as incorrect, and an error message is emitted; others are problematic, and an informational warning message is emitted. In the latter case, the system describes constraints that the programmer is expected to adhere to in the future. In both cases, the code can still be run – Rubin does not prohibit the code it considers to be flawed from being executed.

Because the system performs this way, the programmer will be able to correct the identified problem as soon after the error is committed. In this case, the possibility that the programmer will be relying on the potentially buggy code in the future is significantly reduced.

We will examine both errors and inconsistent use of code. We will start with errors.

5.1.1 Errors:

The following are situations that are recognized as errors by Rubin. Any occurrence of these results in the system reporting an appropriate error messages to programmers that indicate that they wrote erroneous code. If a programmer receives any of the messages that are described below, he probably will need to go back in his program and repair some blocks of code, as otherwise his program will fail when executed. The situations here correspond to those shown in Chapter 3. It is important to note that for all situations except 5.1.1.7, jirb will not report any error message at the time of code writing; it will crash later when the erroneous code is executed.

Here are the nine situations that we identified Rubin recognizes as errors. Descriptions of each of them we start with short reminder of a problem (additional information can be found in Chapter 3, section 3.2), and then show Rubin in action: a message that Rubin produces for a problem.

5.1.1.1 Branches in Control-flow Statements are not Type-consistent

Recall that type inconsistent branches may lead to future type errors, thus they are considered as errors by Rubin. Below we show Rubin's messages for each case possible for them.

a. **Variable Created with Differing Types:** If a previously undefined variable is created with different types in different branches of a control-flow statement, then the programmer may not rely on the variable containing values of a known type. As this is a dangerous situation, Rubin names these variables with ambiguous types. For this and other similar cases of control-flow statements for future analysis Rubin remember the results from the first evaluated branch – the branch that was written first (before other branches) by the developer (the branch that is the highest in the code). The system must rely on some information in order to carry on type checking. The potential problem was already reported to the programmer, so Rubin assumes that he either corrected the problem, or he controls the situation – in either case the task of the system was accomplished. The same reasoning applies for all other ambiguous situations.

```
1:   # 'f' is not defined
2:   if a>0
3:     f=9
4:   else
5:     f="6"
6:   end
7:   # 'f' has ambiguous type
```

Rubin's Message:

```
ERROR!!! The local variable 'f' was defined in THEN and ELSE bodies
of the IF statement on line 2 with different types.
   Object::Integer
   Object::String
For the future analysis we will assume that its type will be
Object::Integer
```

As in the previous case, this analysis applies to all inner-blocks of control flow statements, including `if`, `case`, `for`, `while`, `switch`, `until`, and `unless.`

b. **Variable not Created in All Branches:** Recall that if a variable does not exist, and is created/assigned in only one branch, then an error condition occurs because the programmer cannot assume, after the conditional, that the variable exists and is initialized. As a result, an error is reported by Rubin, and the message identifies the variable(s) that are known to not exist in advance and differ after.

```
1:   # neither 'f' or 'd' are defined
2:   if a>0
3:     f=9
4:   else
5:     d=6
6:   end
7:   # either 'f' or 'd' is created, not both
```

Rubin’s Message:

```
ERROR!!! The local variable 'f' was defined in THEN body, but was
not defined in ELSE body: IF statement on line 2
For the future analysis we will assume that the local variable ‘f’ exists
ERROR!!! The local variable 'd' was defined in ELSE body, but was
not defined in THEN body: IF statement on line 2
For the future analysis we will assume that the local variable ‘d’ exists
```

c. **Function Returning Different Types:** Recall that functions usually must return one type, and the cases where it is not true must be reported to the programmer. Rubin recognizes those situations as soon as the function definition is completed, and reports an error message indicating the name of the function and all the potential return types.

```
1:  def diffReturn arg
2:    if arg>0
3:      return 3  # integer return
4:    end
5:    return ""   # string return
6:  end
7:  # ‘diffReturn’ returns either an integer or a string
```

Rubin’s Message:

```
ERROR!!!  The function 'diffReturn' defined on line 1 may return
different types:
   Object::Integer
   Object::String
For the future analysis we will assume that the function will be
returning Object::Integer
```

5.1.1.2 Local/Global/Instance Variable Changes Type: As was shown previously, if a variable is already known to contain a value of a given type, and is later assigned a value of a differing type, the programmer may not know the variables type anymore. This is a dangerous situation, and that is why Rubin reports the variable(s), their original and their clashing new type. For future analysis Rubin retains the old type restrictions.

```
1:   b=""  # ‘b’ is a string
...
4:   b=5   # ‘b’ is redefined as an integer
```

Rubin's Message:

ERROR!!! The local variable 'b' changes its type on line 4. It was a Object::String and attempts to become an Object::Integer
For the future analysis we will assume that the local variable 'b' retains its first type - Object::String

5.1.1.3 Number of Targets Does not Match Number of Values in Multiple Assignment: As we said previously, the situation when a programmer uses different numbers of targets and values in the multiple assignment expression, appears to be incorrect. That is the reason why Rubin emits the error message. For the example below, Rubin reports that the programmer fails to assign one of the variables, namely `c`.

```
9:   a,b,c=1,2  # type of 'c' is null
```

Rubin's Message:

ERROR!!! The number of arguments (3) for a multiple assignment on line 9 is not equal to the number of values (2)

5.1.1.4. Inappropriate Use of `Break, Redo, Next` Statement: As was mentioned, `break, next,` and `redo` statements are not allowed outside of loops. Rubin can recognize and report on these situations.

```
11:  def funWithBadBreak
12:    break  # illegal outside loop
13:  end
```

Rubin's Message:

ERROR!!! BREAK statement is used outside a loop on line 12

5.1.1.5 Function Called with Wrong Parameters: Recall that functions calls must correspond with matching functions declarations; functions cannot be called with wrong parameters. Here we dwell on three potential dangerous situations for this case, and show how Rubin works with all of them.

a. A function called with the wrong number of arguments. Rubin can report on problem. For the example below, Rubin reports that the function `concat` cannot be called with two arguments.

```
5: “hi”.concat(“\n”,”.”) # wrong number of arguments
```

Rubin’s Message:

```
ERROR!!!! Cannot call the function ‘concat’ of the class Object::String
with 2 arguments: line 5
```

The same is true for lambda procedures. Rubin will notice and report that the lambda is called with the wrong number of arguments.

```
30:   a=lambda{2}
31:   a.call(2) # wrong number of arguments
```

Rubin’s Message:

```
ERROR!!! The proc is called with the wrong number of arguments: 1
instead of 0: line 31
```

b. The case when a function may be called with arguments of the wrong types is recognized by Rubin too.

```
12: “hi”.concat(3) # wrong type of the argument
```

Rubin’s Message:

```
ERROR!!! Argument 1 of String.concat must be Object::String;
Object::Integer was provided: line 12
```

c. Rubin is also capable of working with the yield and block problem: if a function expects a block, but is not given one, Rubin reports to the programmer that a block must be supplied.

```
3:   def apply
4:     yield
5:   end
6:   apply  # needs block
```

Rubin’s Message:

ERROR!!! A block was required while calling 'apply' but was not supplied: line 6
It expected 0 arguments

As mentioned already in this thesis, Rubin does not support functions with arbitrary number of arguments. If Rubin sees the function like that, it ignores the last formal.

5.1.1.6 Parameterized Types May Only Contain Values of a Single Type: Rubin also recognizes cases where parameterized types may only contain values of the same type. If the programmer-provided parameterized types with value types that are not compatible, the system will report that that the two hashes have incompatible value types.

```
6:   h1={1=>”one”}
7:   h2={"two"=>2}
8:   h1.update(h2)  # type clash
```

Rubin’s Message:

ERROR!!! Incompatible value types of binary types Object::Hash and Object::Hash: line 8

We must mention that Rubin assumes that parametric types always must have compatible value types: Rubin cannot work for the code that contains legitimate parametric types with incompatible value types: for example, the predefined function `divmod` returns an array with two values: an integer and a float. Although that is the behaviour designed by Ruby developers, Rubin will mark this situation as a type error, and report a corresponding message.

5.1.1.7 Class Modules, and Constants Redefined to Another: As we mentioned before, once a name is bound to a class, it cannot be reassigned to a module, and vice versa. Rubin is able to notice this problem, and report as meaningful error message as it can derive.

```
1:    class AClass
...
13:   end
...
17:   module AClass  # redefine class as a module
...
23:   end
```

a) Message of Ruby Interpreter:

“TypeError: AClass is not a module”

b) Rubin’s Message:

ERROR!!! The class AClass is redefined as a module: line 17

Although, as can be seen from the report above, jirb recognizes this situation and reports an error message, it can also be seen that this message is not descriptive: one of the critical pieces of lacking information in jirb’s message is the old type of the redefined Ruby essence (class/module/function). We believe, that this information may be useful for the programmers, that is why Rubin reports a similar message, but with the old type.

5.1.1.8 Ordinary Functions Called as Class/module Functions: Recall that calling an ordinary function with the name of its class is not allowed; Rubin recognizes such situations, and reports a corresponding message.

```
1:    class A
2:      def ordfun
3:      end
4:    end
5:    def wrap
6:      A.ordfun  # Error, ‘ordfun’ is not a class function
7:    end
```

Rubin’s Message:

ERROR!!! A function ‘ordfun’ exists, but it must be a module function: line 6

5.1.2 Informational Messages:

In this subsection we show the situations Rubin considers to be potentially dangerous, but not necessarily erroneous. All of them can be resolved by adding a necessary piece (an undefined function, for example) to the code. Therefore, Rubin does not report error messages, but rather informational messages in such situations. The jirb interpreter reports no message at all. Implicit in these messages is an obligation that the programmer does not execute some block of code before all other structures that it uses are defined. In the jirb interpreter, the program will crash if the programmer neglects to satisfy the obligation. Rubin highlights these obligations for programmers, enumerating and reporting necessary types for the structures, in order to reduce the number of runtime crashes.

5.1.2.1 Use of Functions Before Declarations: Recall that Ruby programmers may call undefined functions provided that they will define them before the calling blocks of code that reference those functions are executed. Rubin is able to see these cases, and report them to the programmer supplying the derived type for the called function also.

```
12:   def lcm a,b
13:     a*b/gcd(a,b)  # 'gcd' not defined
14:   end
```

Rubin's Message:

```
WARNING!!! The function 'gcd' is not defined for a class Object: line
13
If you want to use the function 'lcm' you need to define 'gcd'
It must have 2 arguments
Argument 1: any type
Argument 2: any type
It may return any type
The receiver must be Object
```

The system works for `aliases` and `undefs` as well. If a programmer tries to duplicate or undefine an undefined function, Rubin reports that the function that to be duplicated is not defined yet.

```
1:    # function ‘euclid’ not defined
2:    def duplicatingFun
3:      alias gcd euclid
4:    end
```

Rubin’s Message:

> WARNING!!! The function to be aliased – euclid – is not defined: line 3

```
1:    # function ‘und’ not defined
2:    def undefiningFun
3:      undef und
4:    end
```

Rubin’s Message:

> WARNING!!! The function ‘und’ might not be undefined at this point as currently it is not defined: line 3

In order to avoid the future problem, the programmer must define the lacking functions (`gcd`, `euclid`, and `und` respectively) before he calls the ones that rely on them (`lcm`, `duplicatingFun`, and `undefiningFun`).

5.1.2.2 A Global/Instance/Class Variable is Used Before Definition: The cases when a programmer uses yet undeclared variables, are supported by our system. Rubin produces an informational message, which tells the programmer all the information it could infer about the variable.

```
1:    class A
2:      def geta
3:        @a  # ‘@a’ not defined[28]
4:      end
5:    end
```

[28] Recall that classes can be extended in other code sections: Ruby implements open classes, so `@a` may be defined later.

Rubin's Message:

> WARNING!!! Before calling function 'geta' you need to define the instance variable @a: line 3
> The instance variable @a was used, but isn't defined: line 3

In order to avoid the future problem, the programmer must define the instance variable `@a` before he calls the method `geta`.

5.1.2.3 Reference to an Undefined Class/Constant. Using Rubin, programmers also will be able to get feedback about cases when they reference to an undefined class or a constant. Rubin reports a corresponding informational message to the programmer.

```
1:   def callConst
2:      A::B  # 'A' not visible
3:   end
```

Rubin's Message:

> WARNING!!! You are trying to access a constant B from an undefined class A: line 2

In order to avoid the future problem, the programmer must define a class A, and the constant `B` for it before he calls the method `callConst`.

5.1.2.4 Functions are not Type-consistent. Recall that the principal typings system must be able to allow programmers to change their function declarations. Rubin is flexible enough to be able to change its environment to support function redeclaration. Still, if a programmer changes the type of an existing function, Rubin reports a corresponding warning message to the programmer.

```
1:     def currentValue
2:        3
3:     end
4:    currentValue +5  # 'currentValue' must return number
5:    def currentValue
6:       ""
7:    end
```

Rubin’s Message:

```
WARNING!!! The function being defined on the line 5 - 'currentValue' -
existed before, inside the class Object
Return types are inconsistent! Previously the function returned
'Object::Integer' and now it will be returning 'Object::String'
```

5.1.2.5 Definitions of Functions with the Type Inconsistent to the One that It Was Used Before. Recall that the principal typings system must be capable of type-checking the code, that has references to the undefined pieces. Rubin provides such capability; it also provides appropriate messages for programmers in cases when he defined the function with a different type than the one expected.

```
1:    def fun a
2:      callLater  # ‘callLater’ must be without arguments
3:    end
4:    def callLater b  # ‘callLater’ expects one argument
5:      b=""
6:    end
```

Rubin’s Message:

```
WARNING!!! You are inconsistent! Previously you used the function
‘callLater’ with the minimum number of accepted arguments was '0'
and now it will be 1: line 4
```

In the code above the function callLater was called without arguments, but later the programmer defines it with one formal: if the programmer runs the function fun, his code will fail. In order to avoid such situations, the programmer must try to be consistent throughout entire code.

5.1.3 Comparison of Rubin to Ruby Interpreter and Systems with Principal Types.

In this subsection, we showed a host of Ruby constructs that the system was able to type correctly, and messages returned by it. Below we give a short summary of general Ruby cases that can cause type problems, and whether they are supported by systems with principal types and systems with principal typings, i.e. whether they are reported to the programmer immediately after they were committed. The red square mean that the system that the current column represents does not provide a feedback to a programmer if he committed the error described in the same row.

Table 5.1 Comparison of Error Message Reporting

Description of Problem	jirb Interpreter	Principal Types only	Principal Typings (Rubin)
Branches in Control-Flow Statements are Type-Inconsistent	🟥	🟥 🟩	🟩
Local/Global/Instance Variable Changes Type	🟥	🟥 🟩	🟩
Target Count Mismatches Value Count in Multiple Assignment	🟥	🟩	🟩
Ordinary Functions Called as Class/module Functions	🟥	🟩	🟩
Functions are Type-Inconsistent	🟥	🟩	🟩
Definitions of Function Type-Inconsistent with Prior Calls	🟥	🟥	🟩
Inappropriate Use of `Break, Redo, Next` Statement	🟥	🟩	🟩
Function Called with Wrong Parameters	🟥	🟥 🟩	🟩
Parameterized Types May Only Contain Values of a Single Type	🟥	🟥 🟩	🟩
Class/Module/Constant Redefined to Another Kind	🟩	🟩	🟩
Use of Function Before Declaration	🟥	🟥	🟩
Global/Instance/Class Variable Used Before Definition	🟥	🟥	🟩
Reference to Undefined Class/Constant	🟥	🟥	🟩

Not Supported	🟥
Supported	🟩

In Table 5.1, a divided cell, indicating both supported and not supported, shows that the system works for this problem in all situations except those that involve function redefinition.

Table 5.1 shows:

- that almost none problems described in Chapter 3 are recognized by **jirb**; the only exception is the case of class/module/constant redefinitions, in which the error message is reported, but can be improved,
- that many of the described problems cannot be recognized by **systems with principal types** property only,
- that all the described errors are recognized and reported by our **principal typings system**.

As can be observed, Rubin extends the set of possible Ruby cases, for which error messages will be reported to programmers. There are still some difficulties (parametric types with different value types, functions with an arbitrary number of arguments), that remain for the future work. Still, Rubin, the system with principal typings, was able to enhance Ruby code safety by supporting more cases than other previously existing systems did.

5.2 Application to third party systems

This subsection evaluates whether the problematic constructs described in the previous section are actually used by Ruby programmers, thus there exists a danger that the real Ruby code can contain kinds of bugs described in this thesis, not discovered by the developers. Below we give three popular Ruby projects: two, out of three, are in the top 100 of most popular Ruby projects by downloads from `http://rubyforge.org`. All of the three had releases, which contained such bugs that the system with principal typings could catch and report do the developers. Each of the bug descriptions below indicates that programmers that downloaded and used these releases encountered these hidden bugs, and that these bugs hindered their further development process by not allowing them to use the features of the buggy application they needed. By providing follow-ups of developers we show that the developers themselves admitted the errors, and corrected them.

5.2.1 Project name: Mechanize

The Mechanize library is used for automating interaction with websites. Mechanize automatically stores and sends cookies, follows redirects, can follow links, and submit forms. Form fields can be populated and submitted. Mechanize also keeps track of the sites that users have visited as a history [21].

At the beginning the project was named `WWW::Mechanize`. The first tracked version, 0.4.0, was released in 2006-03-22. The current version's number is 0.9.0, released on 2008-12-23. Currently the project is 77th most popular Ruby project to download with 16852 downloads [40].

Bug report:

[#15049] Mechanize 6.10 is broken for rails 1.8.2+

Description of the bug[29]

The developers provided the arguments for the alias function in a wrong order.

Follow-up:

Message

```
Date: 2007-10-30 12:37 Sender: Aaron Patterson fixed in
changeset:445
```

URL: http://rubyforge.org/tracker/?group_id=1453&atid=5709&func=detail&aid=15049

It took developers 5 days to fix the bug after it was reported. The buggy version was released July 26th, 2007, while next one, with the bug corrected was released December 4th, 2007, so the buggy version was unpatched for 131 days.

[29] Here, and in two other places we provide simplified versions. The exact descriptions of bugs are given in an appendix

Relevant code:

```
module WWW
  # :stopdoc:
 …
    class Page
     …
      if RUBY_VERSION > '1.8.2'
        alias :inspect  :pretty_inspect
      end
    end

    class Link
     …
      if RUBY_VERSION > '1.8.2'
        alias :inspect  :pretty inspect
      end
    end
end
```

Rubin in action:

The programmer who wrote this code provided arguments for the built-in function `alias` in a wrong order. If condition is not met (`RUBY_VERSION > '1.8.2'`) the Ruby interpreter does not evaluate the erroneous parts of the code, therefore it does not return any message for developers: this is apparently what happened. If the code was run through Rubin, a programmer would get the following message, that corresponds to the one described in subsection 5.1.2.1:

```
The function that is being tried to be aliased - pretty_inspect - was not defined
```

As can be seen, the programmer would have the information similar to the one that he received from the bug report. Thus he would be prevented from releasing the buggy code; he would be able to correct the error, and release a correct version at once.

5.2.2 Project name: TMail

TMail is an email handler library for Ruby. TMail can extract data from mail, and write data to mail following the relevant RFCs on the subject [46]. The current released version's number is 1.2.3.1. It was released on 2008-04-11.

Description of the bug

The programmer made a typo, writing 'Regep' instead of 'Regexp'.

Follow-up:

Message

Date: 2008-01-10 10:21 Sender: Mikel Lindsaar

Thanks for this, trunk REV 178 handles this bug... Mikel

URL:
http://rubyforge.org/tracker/?group_id=4512&atid=17370&func=detail&aid=16899

It took developers three days to fix the bug after it was reported. The buggy version was released on December 2nd , 2007, while next one, with the bug corrected, was released on January 11th, 2008, so the buggy version was unpatched for 40 days.

Rubin in action:

A programmer made a typing error – he typed 'Regep' instead of 'Regexp'. The Ruby interpreter does not return any message in this case. If the code was run through Rubin, a programmer would get the message:

Constant 'Regep' is not defined

As for the previous case, the programmer would find out of the error immediately after running the code through Rubin. Thus he would have an opportunity to correct the bug before releasing his application.

Relevant code:

```
module TMail

  class HeaderField
     …
   def new_from_port( port, name, conf = DEFAULT_CONFIG )
        re = Regep.new('\A(' + Regexp.quote(name) + '):',
'i')
        str = nil
        port.ropen {|f|
            f.each do |line|
              if m = re.match(line) then
                   str = m.post_match.strip
              elsif str and /\A[\t ]/ === line then
                   str << ' ' << line.strip
              elsif /\A-*\s*\z/ === line then
                   break
              elsif str then
                   break
              end
            end
        }
        new(name, str, Config.to_config(conf))
      end
    …
end
```

5.2.3 Project name: webgen

webgen is a free (GPL-licensed) command line application for generating static websites [47].

The first version 0.1.0 was released on 2004-07-08. The current released version's number is 0.5.10. It was released on 2009-08-10**.** Currently the project is 80th most popular Ruby project to download with 16790 downloads [40].

Description of the bug.

Calling an ordinary, non-module function with the name of the module (instead of the name of the instance) as the receiver.

Follow-up:

Message

Date: 2005-12-12 04:48 Sender: Thomas Leitner Fixed!

URL: http://rubyforge.org/tracker/?group_id=296&atid=1207&func=detail&aid=2991

Relevant code:

```
…
module FileUtils
     …
      def ask_before_delete( ask, func, list, options = {})
          …
      end
   …
end
…
FileUtils.ask_before_delete( @ask, :rm, file, :force =>
true )
…
```

It took developers three days to fix the bug after it was reported. The buggy version was released on November 27th, 2005, while next one, with the bug corrected, was released on December 29th, 2005, so the buggy version was unpatched for 32 days.

Rubin in action:

A programmer defined a function '`ask_before_delete`' for the module '`FileUtils`' and did not set it as a module function. Later a programmer calls this function with a receiver of the name of the module – this is allowed only for module functions. If the code was checked via Rubin, a programmer would get the message:

```
ERROR!!! A function 'ask_before_delete' exists, but it must be a module function
```

As for the two previous cases, the programmer will receive the error report at once, and he would be able to correct the problem before releasing his program. Otherwise, the interpreter did not notify the programmer of the error, as the code where the error happened apparently was not executed.

5.2.4 Summary of the System's Application to the Real-Life Development

In this section we showed how Rubin can improve coding actual or production Ruby projects. The fact that we showed popular Ruby projects containing bugs that Rubin can find, shows that the system may be useful not only for detecting bugs in small applications, but also for large applications, with thousands of lines of code, that are used by tens of thousands of users.

For the last two out of three examples Rubin generated many other messages. This happens because Rubin does not support the entire Ruby platform (this relates primarily to built-in classes: among big omissions we can mention `File`, `Thread`, and others). After their examination we believe that if Rubin supported every predefined procedure, those false positives would not happen.

5.3 Summary

In this chapter we presented two approaches of how we evaluated our work, and how those two approaches validate our work. Our evaluation suggests that the goals of our research, discussed in first two chapters and briefly mentioned again at the beginning of the current chapter, were achieved.

The developed system has proven to work for many Ruby constructs that are neither supported by jirb, nor by the systems with principal types. As was shown in the first section of this chapter, for most of them Rubin reports error messages earlier than jirb does.

All of the Rubin's messages were reported either immediately after the error was committed, if it was on the root level of interaction session, or immediately after the Ruby program returned to the root level, if it was not there. This not only shows that messages are reported earlier than those of jirb, but also proves a better localization of errors. Reporting error messages Rubin guarantees that the errors were committed in the most recent chunk of code[30], thus facilitating programmers' efforts to find these errors. In comparison, when jirb reports messages, it does not guarantee that the actual source of the problem lies in the most recent chunk of code; in this thesis we showed many examples when the actual source of the problem was located much earlier than corresponding jirb's message.

By providing error messages we showed that they clearly specify the reason of type inconsistencies, their precise location, and, whenever possible, report possible problems in the future if the code is not corrected. In contrast, jirb messages are way too general, and do not specify any of the information shown above to the Rubin's degree. Thus we can claim, that Rubin's messages provide more meaningful messages than jirb does.

Rubin was able to detect unnoticed errors in large, popular Ruby projects, thus showing the importance of this field of research.

[30] By *chunk of code* we mean those code constructs that are located between two adjacent root-level constructs.

A novice user used Rubin for interactive web/mobile development weekly for five weeks, checking 400 lines of Ruby code. The user reported that Rubin helped him identify an unexpected bug in his code, where he forgot some branches of the control flow. The error message from Rubin highlighted this omission and enabled him to correct his code. In his report the user stated that Rubin's messages were clear and understandable, pinpointing the precise location of potential problems. Based on his report, false positive messages, which can be generated by Rubin for unsupported Ruby parts, were not a problem: none were generated. The user reported that Rubin enhanced interactive development for him, and stated that he would use Rubin in future projects.

Based on all these results we believe that our system with principal typings can improve a software development process in many ways. In our next chapter, "Summary and Future work", we conclude our work, give some insights on how our research can be continued, and what new benefits this continuation can give to programmers.

Chapter 6
Conclusions and Future work

In this thesis we have shown that some popular interactive languages (Ruby in particular) do not allow programmers to incrementally develop programs to the fullest extent possible. One reason is that these interactive languages lack types for them, which highlight common semantic errors early. This sets limits for programmers. Using interactive languages could be a very good way to do incremental development, as a programmer can test different procedures immediately after their definitions, as well as refactor code on-the-fly.

One way to solve this problem is to add typing to the language. This is the way that we chose, using Ruby as our exemplar of interactively-developed, dynamic languages. Among the possible ways to add typing to Ruby we chose type inference. This has a primary benefit of retaining compatibility with existing bodies of code since Ruby syntax is left unchanged. Adding typing for Ruby is not a trivial task, as semantics of this language allows one to write constructs for which the types are impossible to determine until run-time. But these slack cases are often confusing or error-prone. The interactive development in turn puts its own challenges to the type inference system.

This project proposes a system that adds principal typings to Ruby helping to improve the process of checking and creating robust code incrementally. As the system works with principal typings (in contrast to the majority of other type inference systems that focus on principal types) it is able to preserve incremental development capabilities of Ruby, thus not impeding Ruby programmers in any way.

The evaluation of the system demonstrates its ability to improve the development process for programmers. The system is capable of finding many type errors in a Ruby code earlier in the development process that the current Ruby interpreter does. Also, in some cases it makes error messages look more meaningful for programmers than messages of the current interpreter. Finally, our system is able to place error messages to a better-localized code section, enhancing Ruby for incremental interactive development.

In the future among other things the system can be incorporated into an integrated development environment (IDE) and give programmers many useful features they do not get from current Ruby environments, like a decent code completion mechanism, type consistence checking, etc.

We believe that the investigation of the approach shown in this thesis and the system that was purposefully developed to do that is an important step in making programming in dynamic languages better.

Although the system already works for a big share of possible Ruby code, it does not cover every possible case. All the features of Ruby that the system supports are mentioned before. Ruby is a very big language and there are many other features not supported (like functions with arbitrary amount of arguments, data types like `Thread`, `File`, and others). We did not find these features interesting from a research perspective. It is achievable to implement them, but also it is time consuming. Especially, it did not appear to be worthwhile since Ruby acquires (and loses too) new features every day. But, if in the future someone decides to release the system for programmers and claim the completeness of it, it will be necessary to implement all of the Ruby features.

We are planning to make the system open-source as soon as this thesis is defended. We believe that some developers may find the system interesting and would like to continue working on it thus making the system better (one of the potential problems programmers may be working on is a completion of all Ruby features – the issue discussed in the previous paragraph).

Programmers may be particularly interested in further development of the system because of the fact that if developed properly it may be used for all Ruby programmers, not only for those who use IRB. Almost every Ruby programmer uses some sort of an integrated development environment (IDE) at some level of the development process, but Ruby IDEs are still very undeveloped comparing to IDEs of popular languages (for example, Visual Studio .NET for C# or Eclipse for Java). One of the most important things that current Ruby IDEs lack is an effective code completion mechanism.

Code completion is a technique that is used heavily by programmers of other languages. A well-designed code completion mechanism can greatly save time for programmers. Ruby developers, in contrast, must deal with a much weaker code completion mechanism. The issue, discussed heavily in this thesis – an absence of a type system for Ruby – is one of the important reasons for the fact, that type completion systems for Ruby are so underdeveloped comparing to those of other languages. The absence of a type system means that Ruby IDEs cannot offer a nice code completion mechanism for programmers with types for functions, variables etc. Basic type inference systems are implemented in some Ruby IDEs, RubyMine [16] is probably the best of them. But the information these systems give during code completion is not sufficient, and with the system like Rubin the process can be greatly improved.

IDE with this system integrated can potentially provide some other useful facilities. We briefly dwell on just two of them: function prototypes and class skeletons.

As was mentioned earlier in this thesis, programmers may use functions before the definition of them. In this case IDE with the system can infer types of these functions, and provide a function prototype later. In this case programmers will not have to worry about finding all the places where such things (use of a function before its definition) happened and define the function manually. Also type information generated by the system can be very useful for better understanding of the function's purpose: sometimes it may be hard for programmers to remember the purpose of some function previously used, especially in the case when they referenced to this function much earlier in the coding process.

This is true not only for functions, but also for other data types referenced before definition, for example, classes. The system can generate useful skeletons of such classes with member and instance functions, class and instance variables, constants, and so on. Moreover, it can provide a description (given for example, in comments or in a pop-up window) of how a particular function, variable, or constant was used.

It is possible to make the system do a completeness checking of a code as well: in particular, whether all referenced routines are supplied, do they integrate correctly, are unit tests available for all code paths, etc.

References

[1] Agesen, O., The Cartesian Product Algorithm: Simple and Precise Type Inference Of Parametric Polymorphism, in *Proceedings of the 9th European Conference on Object-Oriented Programming*. pp. 2-26, 1995

[2] Aho, A.V., Sethi, R, and Ullman, J. D., *Compilers: Principles, Techniques, and Tools,* 2nd ed., Addison Wesley, 2006

[3] Cannon, B., *Localized Type Inference of Atomic Types for Python*, California Polytechnic State University, 2005

[4] Cardelli, L., Basic polymorphic typechecking, *Science of Computer Programming*, 2(8):147-172, 1987

[5] Chacko, J.J., *What is a Ruby Symbol? – symbols explained,* 2008, URL: http://www.rubytips.org/2008/01/26/what-is-a-ruby-symbol-symbols-explained, access date: 10/13/2009

[6] Cockburn, A., Using Both Incremental and Iterative Development, *CrossTalk (USAF Software Technology Support Center)*, 21(5): 27-30, 2008

[7] Damas, L. and Milner, R., *Principal type-schemes for functional programs*. Proc 9th ACM Symp on Principles of Programming languages, pp. 207-212, 1982

[8] Enebo, T., *The Future of JRuby,* 2008, URL: http://java.dzone.com/articles/discussing-jruby-with-thomas-e, date accessed: 10/20/2009

[9] Feynman, R., *Appendix F for Presidential Commission on the Space* Shuttle *Challenger Accident - Personal Observations on Reliability of Shuttle,* 1986

[10] Flanagan, D. and Matsumoto, Y., *The Ruby Programming Language,* O'Reilly, Inc, 2008

[11] Furr, M., et al., Static Type Inference for Ruby, *Proceedings of the 2009 ACM symposium on Applied Computing*, pp. 1859-1866, 2008

[12] Goldberg, A., Robson, D., and Harrison, M.A., *Smalltalk-80: the Language and its Implementation*, Addison Wesley, 1983

[13] Gosling, J., Joy, B., Steele, G., and Bracha, G., *The Java Language Specification,* 3d ed., 2005

[14] Hanna, M., Maintenance Burden Begging for a Remedy, *IEEE Software* 13(6), IEEE, April 1993

[15] Isenhour, P. *String Equality and Interning,* 2007, URL: http://javatechniques.com/public/java/docs/basics/string-equality.html, access date: 10/23/2009

[16] JetBrains, *RubyMine official webpage,* 2009, URL: http://www.jetbrains.com/ruby/index.html, access date: 10/12/2009

[17] Jim, T., What are principal typings and what are they good for? . *Proceedings of the 23rd ACM Symposium on Principles of Programming Languages,* pp. 42-53, 1996

[18] Krishnamurthi, S., *Programming Languages: Application and Interpretation,* 2003

[19] Kristensen, K., *Ecstatic — Type Inference for Ruby Using the Cartesian Product Algorithm*, M. Sc. Thesis, Aalborg University, 2007

[20] Larman, C. and Basili V.R., Iterative and Incremental Development: A Brief History, *IEEE Computer*, 36(6): pp. 47–56, 2003

[21] Mechanize Development Team, *Mechanize official website,* 2009, URL: http://mechanize.rubyforge.org/mechanize/, access date: 10/15/2009

[22] Microsoft Corporation, Type Inference (F#), *MSDN Documentation*, 2009

[23] Nutter, C., *Duby and Juby Languages,* 2009, URL: http://skillsmatter.com/podcast/ajax-ria/charles-nutter-duby-and-juby-languages, access date: 10/11/2009

[24] O'Sullivan, B., Goerzen, J., and Stewart, D., *Real world Haskell,* O'Reilly, 2008

[25] Pierce, B.C., *Types and Programming Languages*. MIT Press, 2002

[26] Pilone D. and Miles.R., *Head First Software Development,* O'Reilly., 2007

[27] Pottier, F., *A modern eye on ML type inference: old techniques and recent developments*. Lecture notes for the APPSEM Summer School, 2005

[28] Pottier, F and Rémy, D., The Essence of ML Type Inference. *Advanced Topics in Types and Programming Languages*, pp. 389-489, MIT Press, 2005

[29] Pressman, R. S., *Software Engineering: A Practitioner's Approach,* 5th ed., McGraw-Hill, 2001

[30] Prins, P., *Ruby: Productive Programming Language,* 2002, URL: http://www.linuxjournal.com/article/5915, date accessed: 10/13/2009

[31] Redmil, F., *Software Projects: Evolutionary VS. Big-Bang Delivery*, Wiley, 1997

[32] Rigo, A., *Psyco,* webpage*,* 2009, URL: http://psyco.sourceforge.net, access date: 10/02/2009

[33] Robinson, J.A., A Machine-Oriented Logic Based on the Resolution Principle, *Journal of the ACM 1(12):*23-41, 1965

[34] Rossum, G.v. *Optional Static Typing*. Artima weblogs, 2000, http://www.python.org/~guido/static-typing date accessed: 04/11/2009

[35] Ruby Community, *Official Ruby Documentation,* 2009

[36] Ruby Community, *Official Ruby Documentation, Array,* 2009

[37] Ruby Community, *Official Ruby Documentation, Hash,* 2009

[38] Ruby Community, *Official Ruby Documentation, Range,* 2009

[39] Ruby Community, *Official Ruby Documentation, Symbol,* 2009

[40] rubyforge.org, *List of the most popular projects on rubyforge.org by downloads,* 2009, URL: http://rubyforge.org/top/toplist.php?type=downloads, access date: 10/03/2009

[41] Salib, M., *Starkiller: A Static Type Inferencer and Compiler for Python,* M. Sc. Thesis, Department of Electrical Engineering and Computer Science, MIT, 2001

[42] Singh, A., *Using Reflection in Ruby,* 2006, URL: http://angrez.blogspot.com/2006/11/using-reflection-in-ruby.html, access date: 10/17/2009

[43] Sramek, D., *Ruby class hierarchy,* image, 2002, URL: http://www.insula.cz/dali/material/rubycl/RubyDataClasses.jpg, accessed date: 10/22/2009

[44] Strandh, R., *Top-down Programming,* 2006, URL: http://dept-info.labri.fr/~strandh/Teaching/MTP/Common/Strandh-Tutorial/top-down-programming.html, access date: 09/30/2009

[45] Thomas, D., Fowler, C., and Hunt, A., *Programming Ruby 1.9: The Pragmatic Programmers' Guide,* 3rd edition, Pragmatic Bookshelf, 2009

[46] TMail Development Team, *TMail official website,* 2009, URL: http://tmail.rubyforge.org/, access date: 09/27/2009

[47] Webgen Development Team, *webgen official website,* 2009, URL: http://webgen.rubyforge.org/, access date: 09/28/2009

[48] Wirth, N., Program Development by Stepwise Refinement, *Communications of the ACM*, 14(4):221-227, 1971

Appendix A. Bugs Descriptions of Evaluated Ruby Projects

This appendix gives exact bug reports for projects, discussed in the section 5.2.

A.1 **Project name: Mechanize**

Below we give a precise bug report, submitted for Mechanize:

```
c:/ruby/lib/ruby/gems/1.8/gems/mechanize-
0.6.10/lib/mechanize/inspect.rb:57: undefined method
`pretty_ins pect' for class `WWW::Mechanize::Link'
(NameError)
      from
c:/ruby/lib/ruby/site_ruby/1.8/rubygems/custom_require.rb:2
7:in `require'
      from c:/ruby/lib/ruby/gems/1.8/gems/mechanize-
0.6.10/lib/mechanize.rb:42
      from
c:/ruby/lib/ruby/site_ruby/1.8/rubygems/custom_require.rb:3
2:in `require'


In Mechanize 6.10 lines 44 and 56 of inspect.rb:


            if RUBY_VERSION > '1.8.2'
                  alias :inspect  :pretty_inspect
            end


The Alias method's proper syntax is:


alias :new_name :old_name


See http://phrogz.net/ProgrammingRuby/language.html


Using Ruby 1.8.4, I get the error below when requiring
mechanize until I correct these lines to:
```

```
alias :pretty_inspect :inspect

Thanks, Eric Beland
```

A.2 Project name: TMail

This is a bug report, submitted for TMail.

```
The HeaderField#new_from_port member fails with the
following exception.


/usr/local/lib/ruby/gems/1.8/gems/tmail-
1.2.0/lib/tmail/header.rb:58:in `new_from_port':
uninitialized constant Class::Regep (NameError)


The cause of the error is a typo:


      re = Regep.new('\A(' + Regexp.quote(name) + '):',
'i')


…
```

3) Project name: webgen

Below is a bug report, submitted for TMail.

```
"webgen clean" crashes with


/usr/lib/ruby/1.8/webgen.rb:124:in `handle_node': undefined
method `ask_before_delete' for FileUtils:Module
(NoMethodError)
The fix is remplacing in lib/webgen.rb line 94


   def ask_before_delete( ask, func, list, options = {} )


by


   def FileUtils.ask_before_delete( ask, func, list,
options = {} )
```

Appendix B. Rubin’s User Manual

Rubin: a type system for Interactive Ruby

User Manual

Andriy Hnativ and Christopher Dutchyn

University of Saskatchewan

UNIVERSITY OF SASKATCHEWAN
DEPARTMENT OF COMPUTER SCIENCE

Overview

Rubin is an extension to the Ruby interpreter (JRuby), and its purpose is to make irb better support interactive program development.

The purpose of the Ruby interpreter, irb, is to develop and test code fragments that will eventually form a complete program. Programming with irb is an interactive process, expected to provide immediate feedback from the Ruby interpreter, typically warnings and error messages that show inconsistencies or programming mistakes, when code is executed. However, a program fragment can be executed only when all dependent fragments are also written.

This may introduce a potentially substantial delay between programming and validation, filled with distractions from writing the needed dependent code. Another possible problem arises with the use of control-flow statements, as Ruby (unlike Java or C) allows programmers to produce different results in different branches. This may lead to unexpected results much later in the code.

As a result, error messages are emitted later than necessary, and may appear in other blocks of code than where the error originates. Essentially, irb lacks a type checker: a system to infer types and check their consistency before code is executed. This is not surprising, given the obstacles that irb's interactive code development raises for type checking. As one develops a program, code fragments are written and re-written; each is difficult to validate in isolation, and complex to merge and re-check collectively.

We have implemented a system, Rubin, which reports type errors for incomplete irb programs, by verifying code blocks as they are developed, and checking that they mesh correctly with other code blocks as the developer changes and replaces them. By using principal typings inference, our tool adds lightweight type checking to the Ruby language without changing the syntax. As a result, coders will be able to reduce development time by more precisely locating errors at a better time in the development process.

Obtaining the System

Rubin can be downloaded from http://www.cs.usask.ca/research/research_groups/selab/projects/index.html. It is available as a *tar* file that was compressed further into a *zip* file. The size of the file is **15.6 MB**.

If you have trouble downloading the file, please contact the developers:

- Andriy Hnativ hnativ@cs.usask.ca
- Christopher Dutchyn dutchyn@cs.usask.ca

Installation

In order to install the system, decompress the zip archive to generate the `rubin.tar` archive; then unarchive the tar file into a user-created directory; for example, `C:\Program Files\Rubin` under Windows, or `/usr/local/rubin` under Linux or MacOS. The installation directory will be populated with a copy of this document, a jruby.jar file, and several subdirectories including bin and lib. The tar archive (and the zip file) can be removed once Rubin is installed.

Deinstallation

Deinstallation of the system is as simple as deleting the directory into which Rubin was installed.

Use

In order to run the system, go to Rubin installation directory, and type in the command line:

```
java -jar jruby.jar -I ./bin --command jirb --prompt default
```

Alternately, the `jruby.jar` can be explicitly pathed in the `java` command. This will start an apparently ordinary Ruby interpreter (irb) with Rubin disabled. This means that the Rubin system will not monitor any of the users interactions with the Ruby interpreter, and that it will neither check any code nor emit any error messages.

There are two ways to enable Rubin to begin checking the user's Ruby interactions – the *verbose* mode and the *silent* mode. The difference between the two is that in verbose mode the system gives messages to a programmer, while in the silent mode it monitors the user's input without producing any messages.

1) Enter the verbose mode – type `tinf_verbose` in the interpreter[31]
2) Enter the silent mode – type `tinf_silent` in the interpreter

The system performs the type checking in both modes. If a programmer wants to disconnect Rubin and return to an ordinary irb, he should type "`tinf_exit`".

3) Disable Rubin – type `tinf_exit` in the interpreter

It is possible to switch between the three modes (disabled, silent, and verbose) at any time of the coding process, provided that the programmer is at the root level of the interaction window.

Any code which is entered while Rubin is disabled, will not be type-checked. Furthermore, any code that depends on that not-type-checked code will report errors because the types for the unchecked code are not available. This is especially important when code will be programmed after some is read in without checking and then checking is enabled. For this reason, the silent mode is recommended over disabling Rubin.

[31] *Note that the there is an underbar, "_", not a space in the names of all three commands.*

Ambiguous Cases

Ruby is a very flexible language. It allows users

- to write functions that return different types,
- use variables that may instantiate to different types,
- create different variables in the different branches of control-flow structures,

and so forth. In each of these cases, there is a risk of program execution errors, because expectations that a function returns a value of a given type, or a variable contains a value of a given type in incorrect. In many cases, the programmer is aware of and takes care to handle these ambiguous cases. But, future program modifications may be made by those with less diligence or incomplete information regarding these cases.

Our system is not omniscient, it cannot discern whether the programmer is aware of and accommodating these cases. Hence, in all the cases when a type-clash occurs, Rubin recognizes the problem. If verbose mode is enabled, it generates a message to the user, informing her of the location and nature of the type error. It is the responsibility of the programmer to understand the message and take appropriate action, or simply ignore it. Rubin does not prevent the flawed code from running, but simply informs the programmer of potential errors.

In order to continue interactive type-checking after finding a type clash, Rubin must deal with ambiguous cases. A rule of thumb is that if there are several possible types, Rubin expects the first encountered one for future analysis. For example, if a function may return different types, the system reports a warning and remembers the first possible type for the future analysis.

For example, the code below, one branch of the `if` statement returns an integer (`3`), and the other an empty string.

```
def foo
  a=3
  if a>0
    3
  else
    ""
  end
end
```

```
ERROR!!!  The function 'foo' may return different types:
```

```
          Object::Integer
          Object::String
  For future analysis we will assume that the function will be returning Object::Integer
```

Another example: an array may contain values of differing types; again, for the future analysis the system will select the first one provided. The example below stores a string, an integer, and another array (containing an integer) in a three-element array:

```
a=["",1,[1]]
```

```
ERROR!!!  The array 'foo' may have different value types:
    Object::String
    Object::Integer
    Object::Array
For future analysis we will assume that the array will be of the type Object::Array[Object::String]
```

As a last example: a variable may have different types in different branches, for the future analysis the system will select the one from the **first assignment line**. For example, the following code will generate an constraint that variable `b` contains integers.

```
if a>0
  if c>10
    b=93
  else
    b="hi"
  end
else
  b=3.14
end
```

```
ERROR!!!  The local variable 'b' may have different value types:
    Object::Integer
    Object::String
    Object::Float
For future analysis we will assume that its type will be Object::Integer
```

Ruby Statements that Rubin Understands

Rubin type-checks the following language constructs:

Loops:

`while, for, until`

Conditionals:

`if, case, unless`

Blocks, Procs, and Functions

Each of the following different kinds of functions:

1. User-defined procedures: `def foo a` …
2. Built-in functions: `+`, `concat`, …[32]
3. Lambdas `lambda |x| …`
4. Aliased functions `alias new old`

Important: Rubin does not support methods in which the last argument indicates that they can accept an arbitrary number of arguments. For example, `def some_method(a, b=5, *p) …` is not supported.

Assignments, Multiple assignments

```
x, y, z = 1, "hi", 4.5
```

Classes

Including the following built-in classes:

- `Object`
- `Numeric`
- `Integer`
- `Float`
- `String`
- `Boolean`
- `Array`
- `Range`
- `Symbol`
- `Hash`

[32] See Appendix: List of Supported Functions for complete details.

Rubin knows also how to handle methods, inheritance, instance variables, class variables, singletons, and visibility (`private` and `public` statements).

Files and Modules

`include, require` statements

Module definitions are also checkable.

Local/Global variables, Constants

Messages (type errors and warnings) emitted by Rubin

This section is omitted, because it parallels Chapter 5.

Summary of Ruby Built-ins that are Checkable

Rubin supports many built-in operations for many types. For example, programmers who use Rubin can use all usual operations on numbers: they can use arithmetic operations (all common ones: addition, subtraction, multiplications, divisions, modulo, rounding up and down, exponentiation), comparisons (less than, greater that, equals), bitwise operators (bit shifts (<<, >>), bitwise OR (`|`), AND (`&`), and XOR (`^`)), base change (operators like `hex`, and `oct`), and type conversions.

Rubin tries to adhere to a type preservation policy, yielding the most precise type possible. For example, if two integers are used as arguments in the `+` operation, Rubin infers the result type of this expression to be integer as well rather than just a number.

A big set of operators is implemented for strings as well. Strings can be comparable (with the usual operations: `<`, `>`, `==`, `!=`, `===`, and `empty?` yielding booleans as results), extended, merged, truncated, concatenated, converted to other types or cases (`upcase`, `downcase`, `swapcase`).

Rubin supports also complex parametric type such as arrays, ranges, and hashes. Programmers who want to use these have an ability to use all the common methods for them: working with inner elements (all the popular operations: add an element to an array (hash), remove an element, replace an element, access a specific element, access elements one by one (for example, using built-in iterators such as `each`, `each_key`, `each_value`)), working with those types as a whole (merging similar types, truncating, mapping, reversing, filling).

Each element of Ruby language is an instance of some class that is a descendant of Object. That is why all operators for objects are available for all Ruby elements. Rubin supports a number of sporadic operations including hashing, freezing, comparisons (among supported comparison operators for objects are most popular: `==`, `===`, `eql?`, `equal?`), displaying, identifications (with operations like `object_id`, `__id__`), and others.

A complete list of supported built-in functions is given in the following appendix.

User Manual Appendix: List of Supported Functions

Before listing all the supported built-in Ruby methods of different classes we discuss one important issue – overloading. Often different classes have methods with the same names (like "**+**" for example). This is called *overloading* – when there exist functions with the same name that, depending on the arguments provided, perform different actions. Our system supports overloading by setting constraints. Let us assume that the system encountered "**+**" operation, where one of the arguments is a string. From Ruby specifications, we know that the other argument must have a string type too. There are cases when we don't know anything about the types of the arguments (or the information known is not sufficient) – then the system just sets constraints that whatever is provided as arguments must be consistent to the specifications.

Below we provide a list of built-in functions for which the system works. Description of each function consists of 2 pieces:

- Name (given in double quotes, for example "**+**")
- Different possibilities for this function

Format:

[b] N_{args} T_{ret} T_{arg1} … T_{argN} [**{…}**]

Each type possibility adheres to a special format. If a possibility requires a block as an argument, then the description of this possibility starts with the character 'b'. In this case the last part of this possibility description will be a description of the block. Required parameters (that all possibility descriptions have) consist of: number of function arguments (including a receiver), a return type and types for all the arguments. The following symbols indicate types:

I	Integer
F	Float
S	String
a($)	Array with the inner type $ (Can be any other type letter here – for example, if we want to indicate an array of integers, then we use a notation a(i))
r(#)	Range with the inner type #
h(!,@)	Hash with the key type !, and a value type @
$	An inner type of an array

#	An inner type of a range
!	A key type of a hash
@	A value type of a hash
*	Any type

There are several functions that take an arbitrary number of arguments. For those functions we put 32000 (if last n arguments unify to the same type) or 32001 (if last n arguments unify to the same alternating types – like for the “insert” function) as a number of arguments. The next 2 parameters for those possibility descriptions are the minimum and the maximum number of parameters (we put 32000 to indicate infinity).

A block notation:

```
{|[type1…]|retType}
```

Each description of a block is enclosed in {}. Symbols that are given within | | indicate types of block arguments (a first symbol indicate a type of the first argument, a second symbol – of the second argument, and so on).

Consider three examples of possibilities.

Possibility1: (may be one of the possibilities, for example, for the “+” operation)

```
2 I I I
```

There is no ‘b’ at the beginning, so this possibility does not require a block as an argument. Digit 2 at the beginning of the description indicates that this possibility counts on two provided arguments. The first ‘I’ indicates that a function returns an integer type, and two following ‘I’s indicate that types of 2 arguments, first of which is a receiver, must be integers.

Possibility2: (may be on of the possibilities for the “each” operation)

```
b 1 r(#) r(#) {|#|&}
```

This possibility assumes that a programmer must provide a block for an according function. If this possibility works, this function will take only one argument (a receiver) which type will be a range, and it will return the same type as the argument (a range with the same inner type). Moreover, a type of the first block argument must be the same as the inner type of those ranges.

Possibility3 (for example, for "insert")

```
32001 1 32000 a($) a($) I $
```

This possibility shows that an according function must take at least 1 argument (of an array type) and must return the same type as the first argument (an array of the same type). The next optional arguments must go in pairs – the first of the two will be an integer, and the second will unify to the inner type of the return array.

Below a complete list of the supported functions is given. This list is a precise representation of the file, used by Rubin for resolving constraints of built-in procedures.

```
"+"
2 I I I
2 f I f
2 f f n
2 s s s
2 a($) a($) a($)
2 * * *
2 * * n

"-"
2 I I I
2 f I f
2 f f I
2 f f f
2 s s s
2 a($) a($) a($)
2 * * *
2 f * *
2 * * n

"*"
2 I I I
2 f I f
2 f f I
2 f f f
2 s s i
2 a($) a($) I
2 s a s
2 * * *

"**"
2 n I I
2 f I f
2 f f I
2 f f f
2 * * *
```

```
"%"
2 I I I
2 f I f
2 f f I
2 f f f
2 s s o

"modulo"
2 I I I
2 f I f
2 f f I
2 f f f

"divmod"
2 a(n) f n
2 a(i) I I
2 a(n) I f

"/"
2 I I I
2 f I f
2 f f I
2 f f f
2 * * *

"<"
2 b I n
2 b f n
2 b * *

"<="
2 b n n
2 b * *

">"
2 b I n
2 b f n
2 b * *

">="
2 b n n
2 b * *

">>"
2 * * *
2 I * n
2 I I n

"<<"
2 I I i
2 s s i
2 s s s
2 a($) a($) $
2 * * *
2 * * o
```

```
"&"
2 a($) a($) a($)
2 b b o
2 I I n
2 * * *
2 I * n

"concat"
2 s s i
2 s s s
2 a($) a($) a($)

"crypt"
2 s s s

"=~"
2 I s o
2 b o o

"__id__"
1 I o

"object_id"
1 I o

"display"
1 v o
2 v o *

"eql?"
2 b o o

"equal?"
2 b o o

"freeze"
1 o o

"=="
2 b o o

"==="
2 b o o

"hash
1 i o

"-@"
1 I I
1 f f
1 n n

"+@"
1 I I
1 f f
1 n n
```

```
"finite?"
1 b f

"nan?"
1 b f

"id"
1 I o

"zero?"
1 b f
1 b I
1 b *

"abs"
1 I I
1 f f
1 * *

"to_i"
1 I I
1 I f
1 I s
2 I s I
1 I *

"induced_from"
2 f f o
2 I I o

"to_s"
1 s o
2 s i i

"to_str"
1 s s
1 s *

"to_f"
1 f I
1 f f
1 f s
1 f *

"to_int"
1 I n
1 I *

"floor"
1 I n

"ceil"
1 I n

"round"
1 I n
```

```
"truncate"
1 I n
3 I * s I
2 I * i

"chr"
1 s I

"integer?"
1 b n

"next"
1 I I
1 s s
1 * *

"succ"
1 I I
1 s s
1 * *

"infinite?"
1 I f

"capitalize"
1 s s

"capitalize!"
1 s s

"downcase"
1 s s

"downcase!"
1 s s

"upcase"
1 s s

"upcase!"
1 s s

"swapcase"
1 s s

"swapcase!"
1 s s

"dump"
1 s s

"inspect"
1 s o

"length"
1 i s
1 i a
1 i h
```

```
"size"
1 i s
1 i a
1 i h
1 I I

"begin"
1 # r(#)
1 I *

"end"
1 # r(#)
1 I *

"exclude_end?"
1 b r

"include?"
2 b s s
2 b s i
2 b h o
2 b r(#) o
2 b a($) o
2 b * *

"member?"
2 b h o
2 b r(#) o
2 b * *

"tr"
3 s s s s

"tr_s"
3 s s s s

"unpack"
2 a s s

"ljust"
2 s s I
3 s s I s

"rjust"
2 s s I
3 s s I s

"lstrip"
1 s s

"rstrip"
1 s s

"strip"
1 s s

"intern"
1 * s
```

```
"to_sym"
1 * s
1 * i

"hex"
1 i s

"oct"
1 i s

"sum"
1 i s
2 I s i

"empty?"
1 b s
1 b a
1 b h

"merge"
2 h(!,@) h(!,@) h(!,@)
b 2 h(!,@) h(!,@) h(!,@) {|!,@,@|&}
2 * * *

"merge!"
2 h(!,@) h(!,@) h(!,@)
b 2 h(!,@) h(!,@) h(!,@) {|!,@,@|&}
2 * * *

"update"
2 h(!,@) h(!,@) h(!,@)
b 2 h(!,@) h(!,@) h(!,@) {|!,@,@|&}

"replace"
2 s s s
2 a($) a a($)
2 h(!,@) h h(!,@)
2 * * *

"reverse"
1 s s
1 a($) a($)

"reverse!"
1 s s
1 a($) a($)

"casecmp"
2 I s s

"instance_of?"
2 b o *

"instance_variables"
1 a(o) o
```

```
"remove_instance_variable"
2 o o s
2 o o *

"is_a?"
2 b o *

"kind_of?"
2 b o *

"taint"
1 o o

"untaint"
1 o o

"tainted?"
1 b o

"respond_to?"
2 b o s
2 b o *
3 b o s o
3 b o * o

"methods"
1 a(s) o

"type"
1 * o

"singleton_methods"
1 a(s) o
2 a(s) o o

"nil?"
1 b o

"center"
3 s s I s

"chomp"
1 s s
2 s s s

"chomp!"
1 s s
2 s s s

"chop"
1 s s

"chop!"
1 s s

"rehash"
1 h h
```

"default"

```
1 @ h(!,@)
2 @ h(!,@) !
```

"default="

```
2 h(!,@) h(!,@) @
```

"[]"

```
2 $ a($) I
3 a($) a($) I I
2 a($) a($) r
2 I s I
3 s s I I
3 s s * I
2 s s r
2 s s s
2 i i I
2 i i f
2 o * *
2 o * i
```

"slice"

```
2 $ a($) I
3 a($) a($) I I
2 a($) a($) r
2 I s I
3 s s I I
3 s s * I
2 s s r
2 s s s
```

"push"

```
32000 1 32000 a($) a($) $
```

"squeeze"

```
32000 1 32000 s s s
```

"count"

```
32000 1 32000 i s s
```

"insert"

```
32001 1 32000 a($) a($) I $
3 s s s s
```

"downto"

```
b 2 I I I {|i|&}
b 2 * * * {|*|&}
```

"upto"

```
b 2 I I I {|i|&}
b 2 s s s {|s|&}
b 2 * * * {|*|&}
```

"times"

```
b 1 I I {|i|&}
1 * *
```

"each"

```
b 1 r(#) r(#) {|#|&}
b 1 s s {|s|&}
b 2 s s s {|s|&}
b 1 a($) a($) {|$|&}
b 1 h(!,@) h(!,@) {|!,@|&}
```

"reverse_each"

```
b 1 a($) a($) {|$|&}
```

"each_byte"

```
b 1 s s {|i|&}
b 1 * * {|*|&}
```

"each_line"

```
b 1 s s {|s|&}
b 2 s s s {|s|&}
```

"each_index"

```
b 1 a($) a($) {|i|&}
```

"each_key"

```
b 1 h(!,@) h(!,@) {|!|&}
```

"each_value"

```
b 1 h(!,@) h(!,@) {|@|&}
```

"collect"

```
b 1 a($) a($) {|$|&}
b 1 * * {|*|&}
```

"collect!"

```
b 1 a($) a($) {|$|&}
b 1 * * {|*|&}
```

"map"

```
b 1 a($) a($) {|$|&}
b 1 * * {|*|&}
```

"map!"

```
b 1 a($) a($) {|$|&}
b 1 * * {|*|&}
```

"step"

```
b 3 n n n n {|2|&}
b 1 r(#) r(#) {|#|&}
b 2 r(#) r(#) I {|#|&}
b 3 o * o o {|*|&}
```

"compact"

```
1 a($) a($)
```

"compact!"

```
1 a($) a($)
```

```
"gsub"
3 s s * s
b 2 s s * {|s|&}
2 s * s
b 1 s * {|s|&}

"gsub!"
3 s s * s
b 2 s s * {|s|&}
2 s * s
b 1 s * {|s|&}

"sub"
3 s s * s
b 2 s s * {|s|&}
2 s * s
b 1 s * {|s|&}

"sub!"
3 s s * s
b 2 s s * {|s|&}
2 s * s
b 1 s * {|s|&}

"scan"
2 a(s) s *
b 2 s s * {|s..|&}
1 a(s) s
b 1 * * {|*|&}

"split"
1 a(s) s
2 a(s) s *
2 a(s) s I
3 a(s) s * I
2 a(s) * s
1 a(s) *
2 a(s) * I

"delete_at"
2 $ a($) i

"delete_if"
b 1 a($) a($) {|$|&}
b 1 h(!,@) h(!,@) {|!,@|&}

"fetch"
2 $ a($) I
3 a($) a($) I $
b 2 $ h(!,@) I {|i|&}
2 @ h(!,@) !
3 @($) h(!,@) ! @
b 2 @ h(!,@) ! {|!|&}
```

"fill"

```
2 a($) a($) $
3 a($) a($) $ I
4 a($) a($) $ I I
3 a($) a($) $ r(i)
b 1 a($) a($) {|i|$}
b 2 a($) a($) I {|i|$}
3 a($) a($) I I {|i|$}
2 a($) a($) r(i) {|i|$}
```

"first"

```
1 # r(#)
1 $ a($)
2 a($) a($) I
```

"last"

```
1 # r(#)
1 $ a($)
2 a($) a($) i
```

"has_key?"

```
2 b h(!,@) !
```

"has_value?"

```
2 b h(!,@) @
```

"key?"

```
2 b h(!,@) !
```

"value?"

```
2 b h(!,@) @
```

"keys"

```
1 a(@) h(!,@)
1 a *
```

"frozen?"

```
1 b o
```

"join"

```
1 s a
2 s a s
```

"nitems"

```
1 i a
```

"pack"

```
2 s a s
```

"pop"

```
1 $ a($)
```

```
"clear"
1 a a
1 h h
1 * *

"index"
2 I a($) $
2 ! h(!,@) @
2 I s I
2 I s s
2 I s *
3 I s I I
3 I s s I
3 I s * I

"reject"
b 1 a($) a($) {|$|&}
b 1 h(!,@) h(!,@) {|!,@|&}
b 1 * * {}

"reject!"
b 1 a($) a($) {|$|&}
b 1 h(!,@) h(!,@) {|!,@|&}
b 1 * * {}

"initialize"
1 s s
2 s s s
1 a a
2 a($) a a($)
b 2 a(s) a I {|i|$}

"attr_reader"

"attr_writer"

"attr_accessor"
```