Ecstatic – Type Inference for Ruby Using the Cartesian Product Algorithm

Spring 2007

Master's Thesis (SW10)

by

Martin Madsen, Peter Sørensen, and Kristian Kristensen

Department of Computer Science

Aalborg University

Ruby, Ruby, Ruby, Ruby
Ahaa-ahaa-ahaa
Do ya, do ya, do ya, do ya
Ahaa-ahaa-ahaa
Know what ya doing, doing to me?
Ahaa-ahaa-ahaa
Ruby, Ruby, Ruby, Ruby
Ahaa-ahaa-ahaa
– "Ruby" by Kaiser Chiefs

Aalborg University
Department of Computer Science, Frederik Bajers Vej 7E, DK-9220 Aalborg East

Title:
Ecstatic – Type Inference for Ruby Using the Cartesian Product Algorithm (CPA)

Project Period:
Master's Thesis (SW10),
Spring 2007

Project Group:
D618A

Group Members:
Martin Madsen,
epsen@cs.aau.dk

Peter Sørensen,
ptrs@cs.aau.dk

Kristian Kristensen,
kk@cs.aau.dk

Supervisor:
Kurt Nørmark,
normark@cs.aau.dk

Abstract

This master's thesis documents Ecstatic – a type inference tool for the Ruby programming language. Ecstatic is based on the Cartesian Product Algorithm (CPA), which was originally developed for use in the Self language.

The major contributions of this thesis are: the Ecstatic tool that can infer precise and accurate types of arbitrary Ruby programs. By implementing CPA we confirm that the algorithm can be retrofitted for a new language. Utilizing RDoc we devise a method for handling Ruby core and foreign code both implemented in C. Using Ecstatic a number of experiments were performed that gained insights into the degree of polymorphism employed in Ruby programs. We present an approach for unit testing a type inference system. We compare Ruby to Smalltalk and Self, and conclude that their semantics are similar.

Copies: 7
Pages: 118
With Appendices: 146
Finished: June 14th, 2007

Signatures

Martin Madsen

Peter Sørensen

Kristian Kristensen

Contents

CHAPTER 1

Introduction

Ruby is a dynamic programming language, and therefore defers as many decisions as possible to runtime. Dynamic programming languages tend to have a high degree of flexibility and expressiveness. These factors makes a language like Ruby preferred by some programmers.
Although Ruby has existed since 1995, it is still relatively new to the Western world and almost unknown to the academic world. The language comes from Japan and most of the early documentation was in Japanese. Since then English resources have slowly emerged. With the release of the Ruby on Rails web framework in 2004 the language have gained more attention, and subsequently many books have been published. O 'Reilly (a publisher of technical books) publishes a book sales list every quarter. Sales of Ruby books have increased in the last years, and surpasses the sales of languages like Perl and Python.

TIOBE Software [55] publishes a list once a month of the most popular programming languages. Their list from June 2007 places Ruby on a 10^{th} place. They have statistics from June 2002, and Ruby has featured growth since then.
Sun Microsystems support a project called JRuby [49] and Microsoft's has prereleased IronRuby [50]. The projects implement the Ruby language on the Java Virtual Machine (JVM) and the Common Language Runtime (CLR), respectively. These two announcements illustrate the increasing momentum behind Ruby. After visiting RailsConf 2007, Thorup [54] notes his impression of the adoption of Ruby. By his observation, Ruby is primarily used in start-up shops and in PHP shops wanting to switch to Ruby on Rails Hansson [28]. He believes that the reason is primarily, that developers in these companies are more free to make technology choices compared to big companies.

The nature of dynamic programming languages often require their programs to be type checked dynamically. A dynamic type check happens at runtime. This implies that a programmer must run a program to get any feedback about its behaviour and possible errors. The need to constantly run the program while programming is undesired for several reasons. Furthermore, feedback from running a program is often limited and only covers part of the code.

Additional and possibly better feedback can often be provided by analysing the code statically. One aspect of static code analysis is type inference, in which

types are ascribed to variables, expressions, etc. in the code. Utilizing type inference, the programmer can get valuable feedback on how types flow through the program and, thus, discover errors or wrong behaviour before runtime.

This thesis couples the field of type inference and the Ruby programming language to create tool that does type inference. The developed tool is called Ecstatic and it uses the Cartesian Product Algorithm (CPA) to analyze the types and their flow through a Ruby program. Ecstatic is used to conduct a number of experiments on real life Ruby programs. These experiments form the basis of a discussion of the value of type inference on Ruby and our developed tool.

In the following two sections we present the contributions of this thesis.

Major Contributions

Ecstatic: We have implemented a tool that performs type inference on Ruby programs using the Cartesian Product Algorithm (CPA).

CPA Works on Ruby: We have implemented CPA to work on Ruby programs. The CPA was developed for use on the Self programming language, so it was not immediately apparent that it would work on Ruby.

Experiences in Implementing CPA: We have gained a number of insights and considerable experience in implementing CPA on Ruby. Some of these details and concepts are not readily available in Agesen [1]'s work on CPA.

Foreign Code Inclusion: We present a method for extracting type information from the Ruby Core's RDoc. This enables us to perform more precise type inference, because we are aware of the types of the built-in libraries.

Experiments on Ruby Code: With Ecstatic we have conducted a number of experiments on Ruby programs found at the Ruby Application Archive (RAA). This enables us to collect a set of statistics on how Ruby programs are written, including statistics on data and parametric polymorphism.

Testing a Type Inference (TI) System: Based on compiler validation, we present a method for testing the type inference system. The tests are based on Ruby source code samples and unit tests based on an extension to JUnit.

Comparing Ruby, Self, and Smalltalk: We perform a language comparison between Ruby, Self, and Smalltalk. We conclude that the three languages are very similar on a semantic level. Although the similarities between Ruby, Self, and Smalltalk are often presented, a more thorough comparison has not been done before.

Minor Contributions

Ideas for Future Research: We present four hypotheses regarding Ruby programs and programmers. Three of these are considerably broad, and are hence not confirmed or rejected in this thesis. However, they can be considered ideas for future work and research.

Types in Ruby: We present a suggestion on how to understand types in Ruby. There are different views on this, and we compare these views from a type inference angle.

Outline

Chapter 2 gives an overview of Ruby as a programming language. The purpose is to give the reader an intuitive understanding of Ruby. Furthermore a comparison between Smalltalk, Self, and Ruby is performed. Focus is placed on the semantic similarities and differences between the three languages.
Chapter 3 presents the field of type inference. This includes a definition and discussion of types and how they are understood, as well as a description of two type inference algorithms: the Hindley-Milner algorithm and the Cartesian Product Algorithm (CPA).
Chapter 4 establishes a set of hypotheses that constitute the motivation for this project. This is supplemented by a list of requirements for the development of a type inference tool for Ruby called Ecstatic.
Chapter 5 describes the implementation of the tool using CPA as algorithm. Focus is placed on conveying the experiences gained in implementing CPA and retrofitting it from the Self language to Ruby.
Chapter 6 discusses the relation between compiler validation and testing a type inference system. It also documents the development of a validation suite for Ecstatic, and the results obtained from testing Ecstatic using the suite.
Chapter 7 performs a set of experiments offset in the hypotheses described in Chapter 4. The experiments are conducted using Ecstatic.
Chapter 8 discusses the obtained experiment results, experiences gained in implementing CPA, contributions from this thesis, and finally confirms or rejects the hypotheses.
Chapter 9 concludes the thesis.
Acronyms are listed in Appendix A.

CHAPTER 2

Ruby and Dynamic Object Oriented Languages

At a panel discussion hosted by MIT's Dynamic Languages Group in 2001 Steele [51] stated,

> A dynamic language is one that defers as many decisions as possible to runtime.

In a colloquial way this quote summarizes what dynamic languages are about. The following elaborate on what it means in practice. Ultimately languages are called dynamic because they perform many operations at runtime that other languages perform at compilation time. An example of the dynamic nature of these languages is that programs can evolve and change as they are running. In a dynamic object oriented language this can imply the ability to extend objects and class definitions during program execution. E.g., they can be extended by adding new methods, changing methods, or changing the superclass of a class. Some of these capabilities are available in other non-dynamic languages as well, but the dynamic languages have direct support for it built-in.

Since a program written in a dynamic language can change at runtime, the types (or classes) can change too. This can require the language to be dynamically type checked. Dynamic type checking is discussed further in Chapter 3.

This remainder of this chapter describes Ruby – an object oriented language – via code examples. Following this discussion Section 2.2 describes how Ruby relates to more academically well-established languages like Smalltalk, Self, and Python. This comparison is performed, because previous work done in the type inference field was based on these languages. The comparison will be based on the semantics of the languages and not on the syntactical constructions they use.

2.1 Ruby

In this section we describe the parts of the Ruby programming language that we find most relevant in a type inference context. Furthermore, some basic Ruby is explained to enable the reader unfamiliar with Ruby to understand the code

examples presented throughout the report. For a more thorough documentation we refer to Thomas et al. [53].

The description will follow a "Show, don't tell"-approach popularized in the Ruby world by David Heinemeier Hansson – the author of the Ruby on Rails framework. Hence, emphasis will be placed on code examples, instead of more theoretical explanations. The overall purpose is to give the reader an intuitive understanding of the Ruby programming language.

Ruby is a dynamic object-oriented programming language. Its type system is commonly referred to as **Duck Typing** the concept of which is summarized in the following quote known as the **duck test**:

> If a bird looks like a duck, swims like a duck and quacks like a duck, then it is indeed a duck [64].

This implies that in Ruby, classes are not the way to distinguish one object from another, i.e. the class is not the type in Ruby as it is in languages like Java. Instead, if a Ruby object possess the characteristics required by a caller, then for all intents and purposes it is what the caller wants it to be. A more theoretical definition of duck typing is provided in Section 3.1.1.

A table presenting the Ruby variable naming scheme is seen in Table 2.1. It utilizes the code example seen in Listing 2.1. In general, the first two characters of a name help Ruby and developers distinguish its use.

Variable Type	Characteristics
Local variables	Start with a lower case letter or an underscore (line 1 and 2)
Instance variables	Start with an "at" sign (@) (line 3)
Class variables	Start with two "at" signs (@@) (line 4)
Global variables	Start with a dollar sign "$" (line 5)
Constants	Start with an uppercase letter (line 6). Names of classes and modules are constants (line 8-10)

Table 2.1: Ruby's naming scheme with complementary code example in Listing 2.1.

```
1 local_var
2 _local_var
3 @instance_var
4 @@class_var
5 $global_var
6 MyConstant
7
```

```
8  class MyClass
9    # class body
10 end
```

Listing 2.1: Ruby's naming scheme. The different variables are explained in Table 2.1

Variables in Ruby does not need to be defined before used. If you need a variable you start using it. Often variables are defined assigning to them using an equal sign.

2.1.1 Classes and Objects

In Ruby almost everything is modelled as objects. For example a class definition is an object being an instance of the core class `Class`.

A class definition in Ruby is shown in Listing 2.2.

```
1 class NewClass < SuperClass
2   def initialize
3     # constructor body
4   end
5
6   # class body
7 end
8
9 obj = NewClass.new
```

Listing 2.2: A class definition. `NewClass` inherits from `SuperClass`.

The `SuperClass` definition is optional and defaults to the core class `Object` described later. The class constructor is named `initialize` and is optional. In line 9 an instance of the class is created and assigned to the local variable `obj`.

In Ruby class definitions are executed, and Listing 2.3 gives an example of this. This examples also demonstrates some of the dynamic characteristics of Ruby.

```
1  a = -8
2
3  class NewClass
4    if a < 0 then
5      def my_method
6        # do one thing
7      end
8    else
9      def my_method
10       # do another thing
11     end
12   end
13 end
```

```
14
15 obj = NewClass.new
16 obj.my_method
17
18 class NewClass
19   def my_method
20     # redefinition of my_method to do something else
21   end
22
23   def my_second_method
24     # this method is added to NewClass
25   end
26 end
27
28 obj.my_method
29 obj.my_second_method
```

Listing 2.3: Definitions are executed, which make for interesting use cases.

If the value of `a` is less than 0 at the time `NewClass` is defined, `my_method` will be defined as in lines 5-7. If the value of `a` is more than or equal to 0, `my_method` will be defined as in lines 9-11.

In line 18 the definition of `NewClass` is reopened. `my_method` is redefined in lines 19-21, and a new method is added to the class in line 23-25.

The object `obj`, which was instantiated in line 15, now features a new definition of `my_method` in line 28 and the newly added `my_second_method` in line 29. This is because calling methods uses dynamic dispatch (the method to invoke is located at runtime based on the `receiver`) and is implemented as messages in Ruby.

2.1.2 Methods and Messages

What looks like a regular method call in the code above is actually a message being sent to that object. The syntax is `receiver.message` or `receiver.message()` or just `message`. The part before the dot (`.`) is the receiver of the message, and the part after the dot is the name of the message. The parenthesis after the message are optional. If the receiver part is omitted, and only a message name is present the message is sent to `self`.

When sending a message to an object, the object checks if it has a method matching the name of the message. If it does that method is invoked and the result is returned. If it does not the message is forwarded to the class of the object. The same check goes on at the class level, and if not implemented here, the message will be forwarded to any ancestors of the class. If no ancestor of the receiver implements a method with the name of the message, the exception `NoMethodError` is raised. This process is known as duck typing.

A method returns the value of the last expression evaluated in the method body. This is often just the last line of code. One can also use an explicit `return`.

Methods can be defined in three different ways as displayed in Listing 2.4.

```
1  class NewClass
2    def instance_method
3      # method body
4    end
5
6    def NewClass.class_method
7      # method body
8    end
9  end
10
11 obj = NewClass.new
12
13 def obj.singleton_method1
14   # method body
15 end
16
17 class << obj
18   def singleton_method2
19     # method body
20   end
21 end
```

Listing 2.4: Defining methods.

Line 2 defines a normal instance method on a class, which is accessible after the class is instantiated.
Line 6 defines a class method by prepending the name of the class being defined (in this case `NewClass`) to the method name. A class method is accessible directly from the class just like static methods in Java.
Line 13-15 and 17-21 shows how singleton methods can be defined. The two singleton methods exist only on the object `obj` and do not affect `NewClass`. If a second object was instantiated from `NewClass` it would not have the methods `singleton_method1` and `singleton_method2`. The singleton method definition in line 13 prepends the name of the object it is created on, which resembles the syntax of the class method definition in line 6.
The syntax in line 17-18 is different. In line 17 a special virtual class (called a singleton class, see Section 2.2.2) of `obj` is opened and in line 18 a method is inserted into that class, which makes the method a singleton method of `obj`.

2.1.3 Modules and Mixins

Modules are classes that cannot be instantiated. A module definition basically resembles a class and can contain the same elements as a class definition. As the name "module" indicates it is a wrapper for grouping functionality. This

functionality can be used directly from the module or by mixing it into a class. Listing 2.5 shows an example.

```
1  module NewModule
2    def method1
3      # method body
4    end
5  end
6
7  class NewClass
8    include NewModule
9
10   def method2
11     method1
12   end
13 end
14
15 obj = NewClass.new
16
17 obj.method1
18 obj.method2
```

Listing 2.5: Modules and mixins

Line 1-5 defines a module with an instance method `method1` in line 2. `method1` is similar to an instance method, and is not yet accessible in any way, because modules cannot be instantiated. In line 8 the `NewModule` is mixed into the class `NewClass` by calling `include`. The method of `NewModule` is now available in `NewClass`. `NewClass` can therefore define a `method2` in line 10 that calls `method1` in line 11. Both methods can be called from an instance of `NewClass` in line 17-18.

2.1.4 Core Classes and Modules

The core of Ruby is implemented in C and has 34 classes and 14 modules. Most of the classes include one or more modules as mixins. The base class `Object` includes the module `Kernel` and this class and module provide much of the built-in functionality of Ruby.

Basic functions in Ruby are implemented as methods. For example, the function `puts` that prints to standard output is a method defined on `Kernel`. The "hello world" example looks like this in Ruby:

```
puts "Hello World!"
```

`puts` is a message sent to `self`. `self` in global space is an object called `main`, which is an instance of `Object`. The message `puts` is thus sent to `main`, which

forwards it to its class `Object`, which forwards it to its included module `Kernel`. The general order of method lookup is:

1. The class of the receiver
2. Any included modules of the receiver class
3. The superclass of the receiver class
4. Any included modules of the superclass
5. The superclass's superclass
6. And so on...

If an object gets singleton methods these lookup rules are still valid. However, the immediate class of the receiver may change, which is described in Section 2.2.2.

In Ruby, primitive types are classes. A string literal is an instance of the core class `String`. A small integer literal is an instance of `Fixnum`, and a big integer literal is an instance of `Bignum`. Both `Fixnum` and `Bignum` inherit from the core class `Integer`. Likewise a float literal is an instance of `Float`. Both `Integer` and `Float` inherit from the class `Numeric`. In this way every kind of data in Ruby is an instance of a class.

2.1.5 Attributes and Dynamic Programming

In Ruby instance variables of a class are always private to that class. To access them from outside the class, you must make attributes for them. An example of attributes is shown in Listing 2.6.

```
1  class NewClass
2    attr_reader :readonly_var
3    attr_writer :writeonly_var
4    attr_accessor :readwrite_var
5
6    def initialize
7      @readonly_var = 1
8      @writeonly_var = 2
9      @readwrite_var = 3
10   end
11 end
12
13 obj = NewClass.new
14
15 puts obj.readonly_var
16 obj.writeonly_var = 5
17 obj.readwrite_var = 10
18 puts obj.readwrite_var
```

Listing 2.6: Attributes in Ruby

`NewClass` has a constructor that initializes three instance variables in lines 7-9. Attributes for these variables are created in lines 2-4. In many other programming languages, `attr_reader`, `attr_writer`, and `attr_accessor` would be language keywords. In Ruby they are methods defined on the `Module` class, and thereby accessible to all classes. In line 2, a message `attr_reader` with the argument `:readonly_var` is sent to `self`. `:readonly_var` is an instance of the core class `Symbol`. `attr_reader` creates a read-only method on `NewClass` with the name of the `Symbol` argument, which makes the instance variable with the name of the `Symbol` available for access in line 15. `attr_writer` correspondingly creates a write-only method. `attr_accessor` creates both a read-only and a write-only method.

The three attribute methods (`attr_reader`, `attr_writer`, and `attr_accessor`) utilizes Ruby's dynamic programing features to create the methods that wraps the instance variables on the class. This concept is also referred to as meta programming because you program the language.

```
1  class MyClass
2    def initialize
3    end
4
5    def my_reader(name)
6      MyClass.class_eval <<-INJECTED_CODE
7        def #{name}()
8          return "you called #{name}"
9        end
10     INJECTED_CODE
11   end
12 end
13
14 x = MyClass.new
15 x.my_reader(:hello)
16 puts x.hello()
17 # outputs: you called hello
```

Listing 2.7: Adding a method to a class dynamically using `class_eval`

Listing 2.7 shows an example that uses the meta programming features of Ruby to create something similar to `attr_reader`. The example declares a method named `my_reader`, an instance method defined on `MyClass`, which takes a single parameter `name`. Calling `my_reader` adds a method to the class `MyClass` with `name` as the method name. This method returns "you called" and the name of the method.
Line 6 calls `class_eval` on `MyClass` with a HERE document (basically a string) as parameter delimited with `INJECTED_CODE`. The string contains a normal method

definition and uses `#{name}` to access the method name. `class_eval` evaluates its given parameter in the context of a class. So the result of calling this method is that `MyClass` and all instances of it will have a newly declared method. Line 15 calls `my_reader` with a symbol `:hello`. Line 16 calls the newly defined method and outputs the return value, resulting in `you called hello` to be printed to the console.

There are four different eval methods in Ruby, and the difference between them are basically in what environment they are evaluated, i.e. what `self` is. The four methods are, `eval`, `class_eval`, `module_eval`, and `instance_eval`. `class_eval` and `module_eval` are synomynous and evaluate with respect to a class or module, and are often used to add methods to these. `instance_eval` sets `self` to the receiver, which means you have access to everything in the instance including private variables. `eval` evaluates its given string in the current context, and is as such not restricted to a specific module, class, or instance.

2.1.6 Code Blocks

Code blocks in Ruby are similar to lambda expressions, in that they are treated as anonymous methods. Blocks are used extensively in Ruby, and a common pattern of use is seen in Listing 2.8.

```
1 a = [1, 2, 3]
2 a.each {|x| puts x}
```

Listing 2.8: Using a block to print the values of an array in Ruby

Line 1 declares an array of integers. The `Array` class has a method called `each` that takes a block as a parameter. This block will be executed for each element in the array. Hence, Listing 2.8 will print out `1`, `2`, and `3`. The `each` method is called an iterator, because it repeatedly executes the same code block for each element.

Ruby also supports generators, which are used to feed an iterator. This is supported via the `yield` keyword. An example of this is shown in Listing 2.9.

```
1 def method_123
2   yield 1
3   yield 2
4   yield 3
5 end
6
7 method_123 {|x| puts x}
```

Listing 2.9: Generators and iterators in Ruby

In line 7, a code block is passed as an argument to `method_123`. The code block has a parameter `x`, which it prints to the screen. `method_123` calls the code block by the message `yield`. In line 2-4 the code block is called three times with the arguments `1`, `2`, and `3`. These arguments are passed to the code block and `1`, `2`, `3` is printed to the screen.

Ruby supports closures as well and have a number of syntactical constructs for using them. We briefly discuss two of these, `lambda` and `Proc.new`. Their basic difference is the way they handle the `return` statement. Listing 2.10 shows a code example that illustrates this difference. The call to `Proc` in line 3 results in control being handed over to the closure created in line 2. Upon returning, it will return to the original call site in line 13. `lambda` works differently as seen in line 8-10 and line 14. Executing a `lambda` and returning from it yields control back to the call site that called the closure.
`Proc.new` does non-local returns and works as a LIFO block, i.e. it does not work after its declaring context disappears [4, 9]. `lambda` does local returns and works as a non-LIFO block, i.e. it works even after its defining context has disappeared [9].

Cantrell [9] discusses by example the different ways to create closures in Ruby. We refer the reader to him for further details.

```
1  def foo
2     f = Proc.new { return "return from foo from inside proc" }
3     f.call # control leaves foo here
4     return "return from foo"
5   end
6
7   def bar
8     f = lambda { return "return from lambda" }
9     f.call # control does not leave bar here
10    return "return from bar"
11  end
12
13  puts foo # prints "return from foo from inside proc"
14  puts bar # prints "return from bar"
```

Listing 2.10: Closures in Ruby, `lambda` and `Proc.new` [63].

2.2 Programming Languages Similar to Ruby

Most new programming languages are a mix of old and new ideas, or at least old ideas mixed in a new way. Ruby is no exception. Being a dynamic and object-oriented language it joins a family of languages with the same basic premises. In this section we describe and discuss Ruby as a programming language, and compares it to other similar languages. We will primarily compare Ruby to Self

and Smalltalk; Python will be touched upon briefly. These languages were chosen, because they have been the target of dynamic language type inference. The comparison will indicate how the language similarities render type inference for the languages similar.
For a more detailed and feature by feature comparison we refer to Voegele [60].

2.2.1 Historical Background and Language Relations

The historical background of the creation of the languages is interesting because it shows how the languages are related to each other.

Smalltalk is the oldest language of the four. It was invented by Alay Kay, Dan Ingalls and Adele Goldberg and developed by a team of researchers at Xerox PARC during the 70ties and 80ties. The main development of the language was done during the 70ties with the release of Smalltalk-80 in 1980. Smalltalk-80 is the language specification normally referred to as Smalltalk today.

The development of Self was started in 1986 at Xerox PARC by David Ungar and Randall Smith. At that time and place Smalltalk was the big thing, and Self is very much an offspring from that. Most of the semantics of Smalltalk are reused in Self except the new prototype-based object system of Self.

Python appeared first in 1991 as a version 0.9 release and reached version 1.0 in 1994. It was developed by Guido van Rossum at CWI in the Netherlands. Even though the design choices of Python place the language in the same family as Smalltalk, Van Rossum does not mention Smalltalk as a source of inspiration when asked in a couple of interviews [58, 32].

The development of Ruby started in 1993 with the first release in 1995. It is slightly later than Python, which have allowed the creator, Yukihiro Matsumoto, to find inspiration in both Python and Smalltalk together with Perl.

2.2.2 Language Similarities and Discussion

The semantics of Smalltalk and Self are very similar except for their object system. They are both entirely based on objects but differ in their way of defining, creating and relating objects. Likewise Smalltalk, Self, and Ruby all treat values (primitive as well as complex) as objects.

The object system of Smalltalk is class-based. Classes are used to define the structure of objects, and objects are made by creating instances of classes. The class used to instantiate an object is called the class of that object. Classes are

related through inheritance where each class has one superclass, except the top level class `Object`. The structure of instances are defined by its class and ancestors. A class is itself an object with a class (called the **metaclass**). Classes define named instance variables and methods, that are available to their instantiated objects. Class variables and class methods are defined as instance methods of a class's metaclass [69]. The class of the metaclass is an instance of `Metaclass` and thus ends the instance-of chain.

Figure 2.1 illustrates the way objects, classes, and metaclasses are connected in Smalltalk. It illustrates a simple hierarchy with an `Employee` inheriting from a `Person`. It shows the correspondence between the class of an object and its associated metaclasses. The class and metaclass hierarchy are equal in structure. An interesting thing to note is that vieweing the instance, class and metaclass hierarchy in a left to right fashion makes the entity to the right describe the entity to the left. I.e. the class describes the instance (instance methods), and the metaclass describes the class (class methods). Furthermore the figure illustrates how the `MetaClass`'s class is an instance of `MetaClass` thereby ending the instance-of chain.

Variables are private to an object. They are accessed through methods, and methods are invoked by sending messages to objects. When an object receives a message it searches for a method in it self, and then its super classes if defined. If no method is found that matches the method, a "method not understood" error occurs.
In Smalltalk lingo the object receiving the message is called the **receiver** and the method name is called the **selector**.

Self is a **prototype-based language** and does not have classes. Objects are defined directly instead of being instantiated from a class. If several identical objects are needed, a prototype object is created and then cloned into several objects. Inheritance exists between objects by an object having a parent pointer. The class/instance relationship is typically simulated by objects (the instances) having a shared parent called a traits object (the class)[57].

In *"self includes: Smalltalk"*, Wolczko [69] describes how he implemented Smalltalk on top of Self. The purpose of this implementation was to demonstrate the flexibility of a prototype based language like Self. Wolczko [69] explains how he maps the Smalltalk constructs of classes, metaclasses, and instances to an object structure in Self. He concludes that it bodes well for the quality of Self and the prototype based paradigm that you can implement and emulate a class based language like Smalltalk. We would like to expand this conclusion by stating that it exemplifies the similarities between the semantics of Self and Smalltalk.

In Ruby every object has an associated class, which is an object of class `Class`.

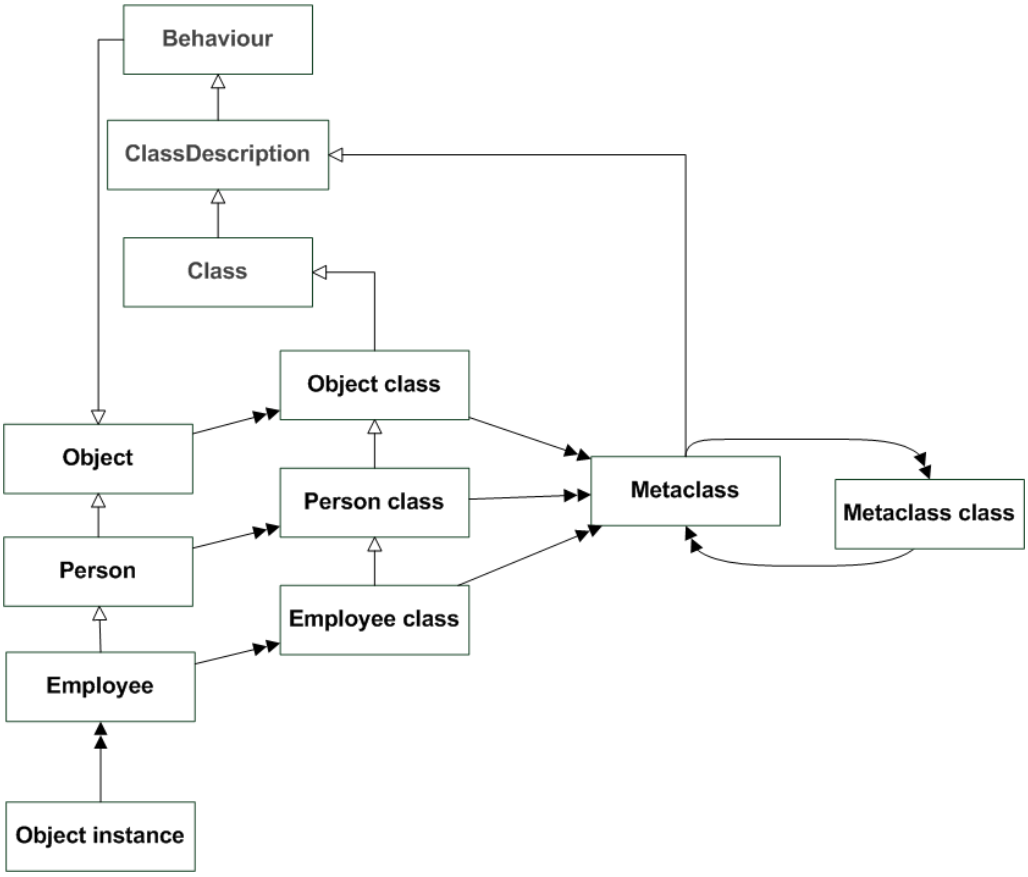


Figure 2.1: An example Smalltalk class diagram. Double ended arrows signify "instance-of" relationships, single ended arrows inheritance. Adapted from Nierstrasz [36].

This class describes the instance methods of the object, and holds a reference to the super class. Every class object has an associated class as well, which contains the class methods of the object. This class is sometimes called the singleton class or metaclass, but it is in fact just an object of `Class`. If singleton methods are added to an object an extra class will be inserted between the object instance and the class object. This extra class is also called a singleton class.
Mixins are equivalent to module inclusion. Effectively they incorporate the definitions in the module with the classes definitions. Mixins are implemented using a special proxy class, which is inserted between the object instance and its class. This proxy class holds a reference to the instance methods of the module and contains the modules instance variables. That means two classes including the same module does not share the modules instance variables. If more modules are included more proxy objects are created and chained together. This also forms the lookup chain, because the order of module inclusion signifies in what

order the modules will be searched.

Figure 2.2 illustrates a class diagram in Ruby. It uses the same example as the Smalltalk version above (Figure 2.1).

Thomas et al. [53] describes how Ruby deals with objects, classes, etc. However, the view presented in Thomas et al. [53] is challenged by DeNatale [17, 18]. DeNatale [17] explains with reference to the current Ruby implementation how objects, classes, and singleton classes are dealt with. Figure 2.2 incorporates the most precise illustration of how objects and classes are represented in the current Ruby implementation.

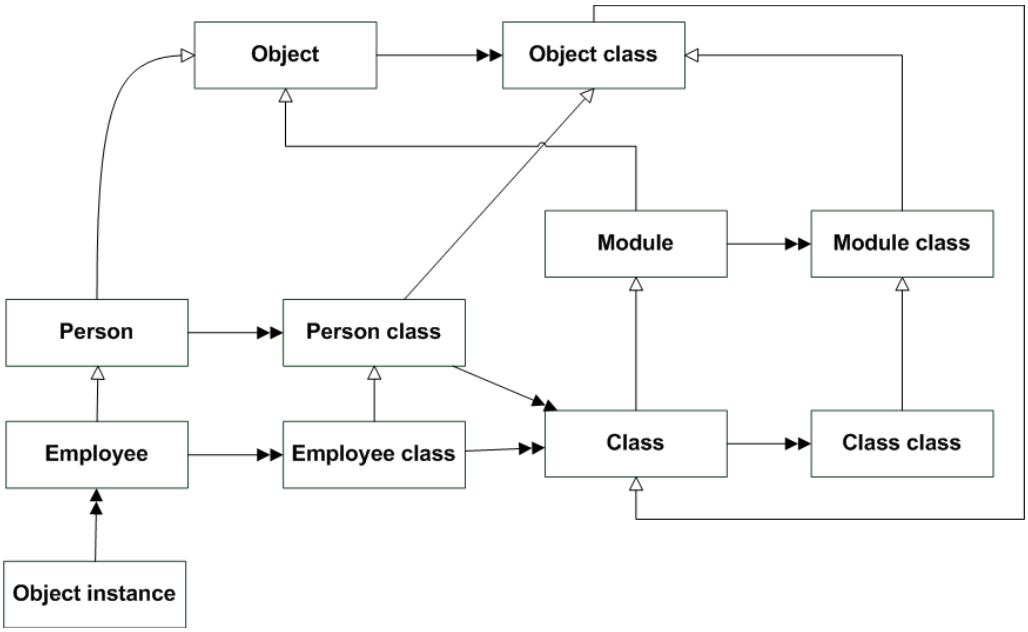


Figure 2.2: An example Ruby class diagram. Double ended arrows signify "instance-of" relationships, single ended arrows inheritance.

Ruby's way of dealing with classes is similar to that of Smalltalks, however, there are differences. In Smalltalk the class of a class is of type `MetaClass` (see Figure 2.1), where in Ruby it is of type `Class` (see Figure 2.2). So the singleton class of a Ruby class is equivalent to Smalltalk's metaclass. Ruby allows the creation of singleton methods, i.e. methods that only exist on a single object instance. This is not supported in Smalltalk.

The nature of Figure 2.2 is similar to Figure 2.1. They both utilize the concept of an object's class being an instance of an object. Likewise the class of a class is also a class. However, they differ in the way this hierarchy of classes is built.

All classes, even the class of a class, in Ruby are represented by instances of

the core class `Class`. This is different to Smalltalk in which a classes class is of `MetaClass`. For Ruby this means that classes can be chained together, because ultimately classes always look like a `Class`. This is exploited in singleton classes, which are used when a singleton method is added to a class. This new class is pushed between the object instance and the original class. The singleton class thus contains the singleton methods, i.e. the methods that are only defined on this object instance. Method lookup will continue to function as before, because if the message is not understood at the immediate class (the singleton class) it is delegated to the original class and so on until the method is found or a "No method error" occurs.

This is not possible in Smalltalk because of the way the class diagram is constructed. However, a disadvantage with the Ruby version is that diagrams often get very complex and it is not as intuitive as the Smalltalk version.
Another difference is that most of the classes seen in Figure 2.2 do not exist or are at least not accessible. They are created virtually in the interpreter during execution.

Mixins can be thought of as a kind of Multiple Inheritance (MI). The difference between Mixins and interfaces found in languages like Java and C#, is that the latter does not support implementation of code and the former does. Coupling mixins with "duck typing" you get a similar effect to MI. Chambers et al. [13] describes how inheritance works in Self and references previous approaches to MI. We refer the reader to this work for a more thorough discussion of MI in Self. Mixins and Duck Typing make Ruby similar to Self, which allows an object to have multiple parents. This is accomplished by **delegation** in which an object can delegate any message it does not understand to a parent object. Parent objects are declared by adding an asterisk to the slot holding the reference.

Smalltalk, Self, and Ruby are all dynamic object oriented languages. Furthermore they all adhere to "duck typing", because objects respond to messages. Each language utilizes garbage collection: Smalltalk's algorithm depends on implementation, Self uses a Generational algorithm[66, 65], and Ruby uses a mark-and-sweep algorithm[53, 65].

A key differentiator for Smalltalk and Self compared to Ruby is that the former utilize an image metaphor. Using Smalltalk and Self means starting the image like an application and then develop your program within the image. When done the image is saved, and the next time you can continue where you left off. The systems are thus not based around source code placed in text files, but rather in a binary environment that contains everything needed to program Smalltalk or Self. Ruby is an example of a language that uses text files to structure source code. In this respect it mimics contemporary and popular languages like Java, C#, C++, etc.

The Smalltalk and Self systems are based around a lightweight core implemented in C. The basic features of the language are implemented here, and everything else is programmed in the language itself. This means that the libraries, etc. are available for modification to the programmer. In Ruby the basic language features as well as the core libraries are implemented in C. The standard library builds upon this and is programmed in Ruby.
Smalltalk and Self are highly reflective, meaning they can inspect and change everything at runtime. This means you can traverse the heap and for example use it as the basis of analysis. Coupling this reflection with the image based development and distribution, means that you can change the entire environment. Ruby supports some kinds of reflection aswell, and these capabilities are called introspection. However, it is not as extensive as its Smalltalk and Self counterparts. Primarily because some part of the Ruby environment are unavailable or at least difficult to change and reflect. The core coded in C is an example of this. Yegge [70] lists three problems with the dynamic programming features of Ruby: the group of `eval` methods[1] are atomic, calls to injected methods are atomic, no access to injected code. The first two points implies that external tools like a debugger does not work properly. All three indicate that the reflective capabilities on injected methods are limited. Hence, although Ruby supports meta programming it appears to not be a first class citizen.

Blocks or closures are dealt with differently in Self and Smalltalk; Ruby incorporates both ways. Wolczko [69] states that Self and Smalltalk differs in their way of dealing with blocks. Smalltalk blocks can be executed any time after they are defined, even after their defining context has disappeared; they are non-LIFO blocks. Self blocks cannot execute if their defining context has exited, i.e. they are LIFO blocks. Ruby supports both ways of block definition as described in Section 2.1.6. `Proc.new` mimics Self blocks (even for non-local returns [4, page 7]), and `lambda` mimics Smalltalk blocks except for return where Ruby does local returns and Smalltalk non-local.
All three languages support treating functions as first-class values. They can be passed to methods as parameters and returned as well.

Figure 2.3 illustrates the differences and similarities between Ruby, Self, and Smalltalk as exemplified through the discussion above.

Colin Steele in quoted for saying, “Ruby is two parts Perl, one part Python, and one part Smalltalk” [52]. We want to challenge that statement and say, “Ruby is half Smalltalk, half Perl, and no Python.” This does not mean that Ruby and Python has nothing in common. But when Matsumoto talks about Python, he only talks about how Ruby differs from Python [34, 59]. He obviously knew about Python, but did not want to copy any of it. Instead, Matsumoto has taken the semantics of Smalltalk. The Ruby FAQ declares that “If you like Perl, you

[1] See Section 2.1.5

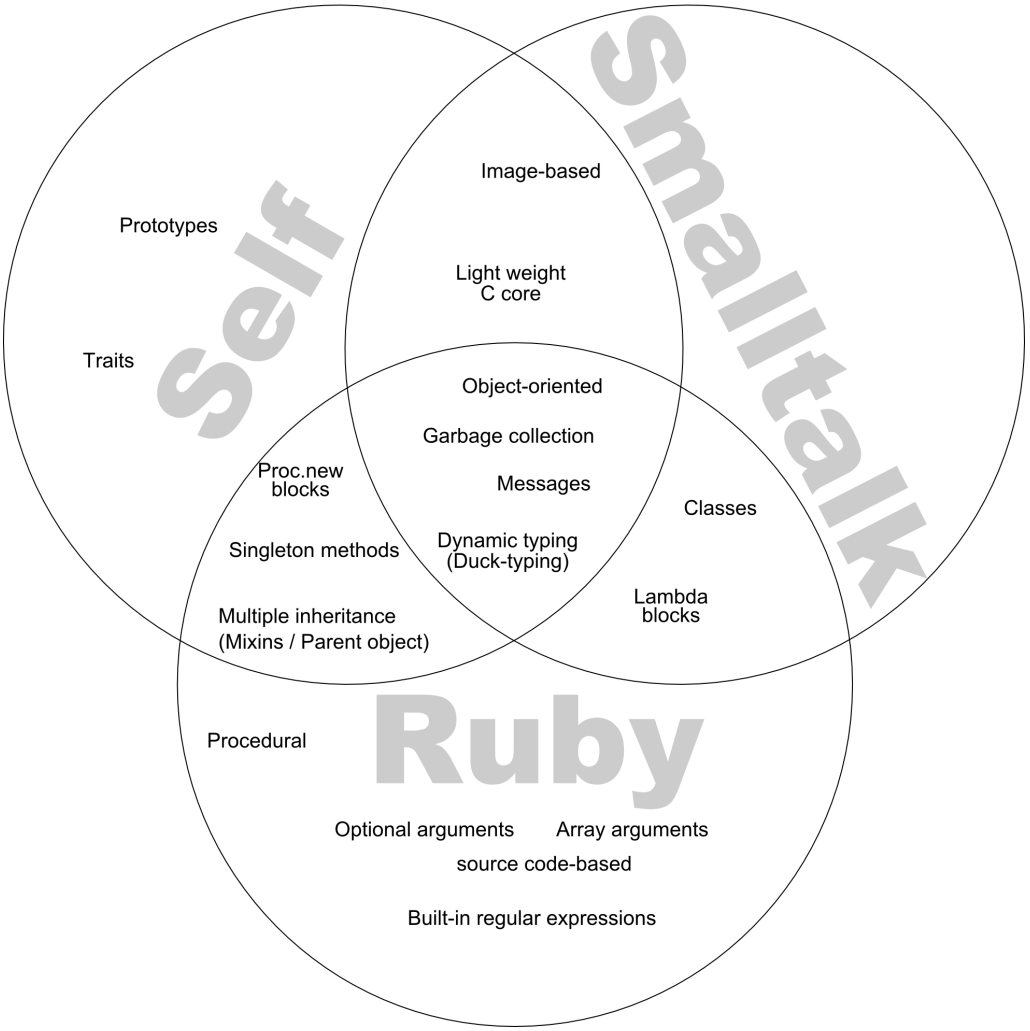


Figure 2.3: A summary of the differences and similarities between Ruby, Self, and Smalltalk.

will like Ruby and be right at home with its syntax. If you like Smalltalk, you will like Ruby and be right at home with its semantics" [20]. The question is then how you weigh syntax against semantics. In this view, Steele weighs syntax 2:1 over semantics. We suggest weighing the two more evenly. Furthermore, when it comes to working with a language like when making type inference, the syntax is almost negligible and only the semantics matter. That is why Ruby is so much like Smalltalk and why type inference algorithms for the one language can be used for the other.

2.3 Summary

This chapter has had two purposes: first to introduce Ruby as a programming language, second to compare it to Smalltalk and Self. On a semantic level Ruby is very similar to these two languages and is clearly inspired from them. However, there are differences and Ruby does bring something new to the table. Ruby may not look as clear and concise as Smalltalk and Self, and the corners of the language are not necessarily sharp. Ultimately coming from a Smalltalk or Self background makes Ruby easier to grasp, and such knowledge can be immediately put to good use in a Ruby world.

This thesis will utilize the results presented in this chapter. This will especially be evident in the many code examples used to illustrate concepts and issues.

CHAPTER 3

Understanding Types and Type Inference Algorithms

This chapter gives a description of types and related concepts in programming languages. Focus is placed on providing an understanding of types and how they are used in this thesis. This understanding will be used in subsequent chapters to develop a type inference tool for Ruby. We will also present an overview of the type inference field, and presents two type inference algorithms: the Hindley-Milner algorithm and the Cartesian Product Algorithm (CPA).

3.1 Definition of Types and Related Concepts

Programmers compose terms into expressions and statements, which are combined into programs. Almost any part of a program has restrictions on it, and some of these are domain ranges. Like functions in mathematics can be defined for a set of values, so can terms, expressions, etc. have a range of values. We refer to such a domain range as a **type**. Examples of types in an object oriented language like Ruby are the system defined types `Fixnum` and `String`, and user defined types such as `Employee` and `Person`.

Types restrict what variables can contain. The type of a variable restricts the range of objects to a specific kind of data, e.g. `Float` or `Fixnum`. An expression of type `Fixnum` means to ensure that the expression will be used correctly. "Correctly" means that an expression object of type `Fixnum` is never used in a way that conflicts with for example a `Person` object.

Even though we state that types are used to restrict the contents of variables they can be used for other purposes as well. Consider an expression used in an assignment as seen in Listing 3.1. The expression in line 2 on the right hand side of the equal sign has a type too even though it is not defined in terms of a variable. Likewise the individual parts of the expression has types too. `a` has the type `Fixnum` obtained from line 1, `2` and `3` are `Fixnum`'s as well.

```
1 a = 5
2 b = (a * 2) + 3
```

Listing 3.1: Assignment in Ruby. All parts have an assigned type even if that part is not directly referenced by a variable. Hence, `2` and `3` have types too (`Fixnum`) even though they are used as sub parts in an expression.

3.1.1 Types in Ruby

Borning and Ingalls [7] define the type of an object in terms of what messages the object understands. This understanding of types correspond to using classes as types and thereby limit the messages an object understands. However, it can also resemble duck typing, where the class of the object does not matter only the messages available. Ruby uses a definition of types along the lines of Borning and Ingalls [7], and explicitly refers to it as duck typing (see Section 2.1).

However, there is a problem with using classes as types in Ruby. This stems from Ruby's dynamic nature that allows the behaviour of objects to be modified after they are created. As explained in Section 2.1.2, singleton methods can be added to an object without affecting the class that the object was initially instantiated from. That is why the type of a Ruby object must be defined in terms of what messages the object understands.

To illustrate this, let us assume that we have two object instances, `alice` and `bob`, instantiated from a `Person` class (see Listing 3.2).

```
1  class Person
2    def initialize(name)
3      @name = name
4     end
5
6    def name
7      return @name
8    end
9  end
10
11 alice = Person.new("Alice")
12 bob = Person.new("Bob")
13
14 def alice.weight
15   return "Do not ask a girl about her weight"
16 end
```

Listing 3.2: Defining a `Person` class and instantiating two object instances from it. A singleton method `weight` is defined on `alice` in line 14-16.

In line 14 a `singleton method` named `weight` is added to `alice`. The method

`weight` is not accessible on the object instance `bob` only on `alice`. Now `alice` and `bob` understand different messages even though they were originally instantiated from the same class.
As described in Section 2.2.2 defining a singleton method on an instance injects a new class (called a `singleton class`) between the object instance and its original class.

There is another way to view types in Ruby. Instead of defining types as the list of messages they understand, types can be based on their immediate class. This definition of types is not a complete departure from the previous definition. As described in Chapter 2 the class of an object contains the methods for that object. When a singleton method is added to an object, a singleton class containing the method is inserted as the new class of that object. The old class then becomes the super class of the singleton class, so the old methods are still available to the object. This means that modifying an object, i.e. by adding a new method, effectively changes its type, because the class is changed. In the above example of `alice` and `bob`, when `alice` gets a singleton method it gets a new class, `Person'` instead of `Person`.

Adding a singleton method to a Ruby object corresponds to creating a new subtype in a traditional class based language like Java. The new class will be a subclass of the old, and the new method added to new class. Likewise in Ruby the inserted singleton class for singleton methods becomes a subclass of the old class, whereby the subtype relationship is preserved.
Mixins change the signature of a class or object as well. Recall from Chapter 2 that mixing in a module sparks the creation of a proxy class that sits between the original class of the object and this class' superclass. Such a proxy class is equal to the singleton class created when introducing singleton methods on object instances (only its place in the hierarchy is different). Following the same argument for singleton classes regarding the subtype relationship, we see that mixins do not change the way one should view a type in Ruby.

The understanding of types influences how and when two types are equivalent. Cardelli [11] uses the following definitions of type equivalence:

Structural Equivalence: two types are the same if they have the same structure.

Nominative Equivalence: two types are the same if they have the same name.

Ruby uses a structural equivalence relation also known as `duck typing`. In the example above this has the implication that the two object instances `alice` and `bob` are considered to have different types. Adding a method to the object `alice` changes its structure, and therefore `bob` and `alice` are objects of two different

types.

3.1.2 Polymorphism

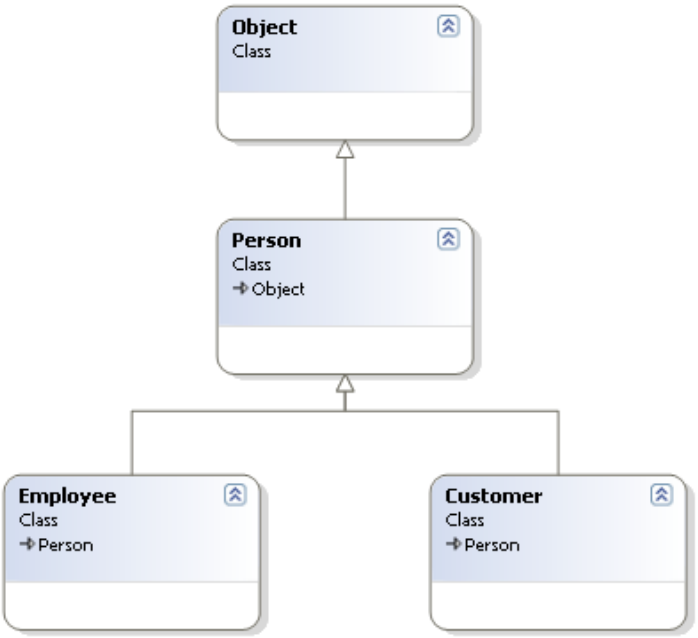


Figure 3.1: A sample class hierarchy for a generic object oriented language.

Figure 3.1 shows a simple class diagram in a generic object oriented language. It defines a `Person` class which is the parent of a `Employee` and `Customer`. `Person` is a more **abstract type** than `Employee`, likewise the latter is more **concrete** than the former. This class hierarchy can be used to illustrate polymorphism. A context could require the use of an instance of `Person`, which would enable both the use of `Employee` and `Customer` instances. This context is polymorphic because it can assume many forms. Polymorphism facilitates reuse and promotes the use of abstract types. Monomorphism denotes the opposite, namely that a context can contain one and only one type.

There is a connection between monomorphic variables and runtime and polymorphic variables and analysis time. At a given time during execution all variables are monomorphic, i.e. they contain only one type and this type is known. Consider the example of performing a method call, which at runtime implies the allocation of an activation record. The parameters to the method call are monomorphic at the activation records creation time, because at this time the exact and most precise type is known. Conversely variables are polymorphic at

analysis time. The discussion on monomorphic and polymorphic variables and their relation to activation records is continued in Section 3.3.

In the context of this thesis we will refer to two types of polymorphism [1]:

Parametric Polymorphism: Refers to the ability of a method to work on parameters of different types. Continuing the example from above a method with a parameter of type `Person` demonstrates parametric polymorphism because the parameter can assume both an `Employee`, `Customer` beside a `Person` object.

Data Polymorphism: Refers to the ability of variables to hold objects of different types. For example an instance variable might exhibit data polymorphism if it can contain objects of varying type.

3.1.3 Type Checking

Programming languages are called **explicitly typed** if types are explicitly required to be stated. Examples of explicitly typed languages are Java, C#, C, and C++. Other languages do not require the programmer to add type information to his program; these are called **implicitly typed**. Examples of such languages are Python, Perl, Ruby, and Java Script.

When types are known, either stated or inferred, a **type checker** can check that the types adhere to the rules of how types are to be understood in the programming language. The type checker can, if the types of the language is defined in terms of methods, check that a message sent to an object can be answered by that object. A **type error** depends on the typing rules and type system of the language, and are reported by the type checker. An example type error in an explicit statically typed language like Java could be to put a string into a variable declared as an integer. In Ruby a type error in Ruby would result in a *MethodMissing* error.

Type checking can be done in two ways:

Statically: type checking is done on compile time.

Dynamically: type checking is done during the execution of the program. As an example consider $v = e$ where v is a variable and e is an expression. The type of the value of the computed expression will be check against the type of the variable.

Type checking differs depending on how types are represented in the language.

For explicitly typed languages the check can be performed at compile time. Naturally the check can be skipped at compile time and use a dynamic check at runtime instead.

For implicitly typed languages it is also possible to perform both forms of checks. Performing a static check on an implicitly typed language implies that the types must first be found. Discovering the types is the job of a type inference algorithm. The inferred types can then be checked like for an explicitly typed language. However, the natural choice for implicitly typed languages is to type check at runtime. As explained previously at a given time during execution all objects and variables are monomorphic. It is because of executions monomorphic nature that implicitly typed languages are typically checked at runtime.

3.1.4 Type Inference

The process of analysing and discovering the types of a program is called **type inference**. Type feedback[2] and type inference are opposites in a world of type discovery. Although this chapter primarily discusses type inference, we have included a discussion on type feedback to give an understanding of the similarities between runtime and analysis time and what this means for type discovery.

The most specific types that can be obtained for a program are the types that result from executing the program. These types are monomorphic, because (as stated previously) there are no polymorphic variables or expressions during an execution. Being monomorphic these types are the most concrete types. Executing a program to obtain type information is known as **type feedback**.

Looking back at Figure 3.1 it illustrates a type hierarchy for a programming language. Actually this hierarchy is how classes are related in Ruby. The diagram show the super and sub-type relationship between types. The further up in the hierarchy the more abstract the type. In Ruby the most abstract or general type is `Object`.

An algorithm that uses type feedback as the method for discovering types could be specified as:

1. Run the program.
2. During execution collect type information on variables and expressions.
   (a) If a type error occurs report this.
3. Annotate the type information collected.

4. Use the annotated information during further development of the program.

There are a number of flaws with this method. First, if the program is in a state where it can not be executed, no type information is available. This could occur if there are syntactical errors in the program. Second, if the program operates on critical data or somehow changes the environment or state in which it is running it is a bad idea to have the incomplete program access and possibly modify or delete this data. However, the biggest problem is that in executing a program you are not certain that all parts of the program will be executed. Therefore, executing a program is not a sure way to obtain types.
A type inference algorithm will try to infer the same types as if the program had been executed without having the flaws listed above. It does so by analysing the program without executing it.

In the remainder of this chapter we will discuss two algorithms for type inference. The Hindley-Milner algorithm commonly used in functional programming languages, and the Cartesian Product Algorithm (CPA) used for the Self programming language.

3.2 The Hindley-Milner Algorithm

The algorithm was first described in the context of combinatory logic by Robert Hindley. Later and independently it was described by Damas and Milner [16] in the context of the ML programming language. Damas and Milner [16] show that the algorithm is sound and complete. The algorithm set out by introducing a set of type inference rules for the core language of ML. The algorithm described by Damas and Milner [16] requires the unification algorithm developed by Robinson [44].

The algorithm is part of the type inference system of many functional programming languages. It is an integral part of the ML programming language and many of its variants. Furthermore it forms the foundation for the type system of Haskell, and is used in many functional programming languages.

In the following sections we will give the examples in ML rather than in Ruby. We make the switch because the Hindley-Milner algorithm is primarily concerned with functional languages.

The Algorithms Process

To explain the workings of the Hindley-Milner algorithm we will use an example `map` function as seen in Listing 3.3 [27]. This function applies a function to every element of a single-type list

```
1 fun map f [] = []
2   | map f (h::t) = f h :: map f t
```

Listing 3.3: The `map` function in ML [27].

This function has the following signature:

$$(\alpha \rightarrow \beta) \rightarrow (\alpha\text{list}) \rightarrow (\beta\text{list}) \tag{3.1}$$

From the signature we deduce that `map` takes a function and applies it to all elements of a list. The function `f` must have the signature $\alpha \rightarrow \beta$. The second parameter is a list containing elements of type α. The result of applying a the `map` function with argument `f` and a list is a list whose elements will be given a type of β.

The algorithm proceeds by setting up a set of type equations and then solving them with respect to the desired type variable [10]. Solving the set of equations is the job of the unification algorithm. Using the example from above the algorithm would create a list of equations to satisfy `map`. The following list gives examples of facts that can be extracted from the function:

1. `map f [] = []` – the return value of `map` is a list.
2. `(h::t)` – `h` represents the type of the list. `h`'s type will be assigned the type variable α. `t` is th tail of the list and is therefore of type α list.
3. `f h` – `f` can be applied on `h`. The return type of `f` is assigned the type variable β.
4. `map f t` – the return type of `map`.
5. `f h :: map f t` – the return type of `map`. Composite of `f`'s return type (β) and `map`'s (a list).

Line 1 states that the return type of `map` must be a list. Line 2 assign a type variable to `h`. Line 3 applying `f` on `h` yields a return type of β. From line 4 we know the return type of `map` is a list. Line 5 gives the final clue. `map` must return a list of β, because line 3 states that the left hand side of the list concatenation is β and ML lists can only contain one type therefore `map f t` must be β or a list of β as well.

The deduced function signature of the `map` function is polymorphic, i.e. it can contain many different types as long as they satisfy the equations. This is exemplified by α and β. Upon actual application of the function the type variables will be replaced with actual types, and checked.

Cardelli [10] describes an interesting feature of the Hindley-Milner algorithm, namely that it "*finds the most best (most abstract) type for programs*"[10, page 4]. However, in some cases this is also a limitation as Jones [30] writes. The problem is that types are either too broad or too specific, i.e. either monomorphic or polymorphic, and the types need a relation between them. Hence a type cannot be a set of types, but must be a type in a hierarchy.

3.3 The Cartesian Product Algorithm (CPA)

This section describes and discusses the Cartesian Product Algorithm (CPA) developed by Agesen [1]. His algorithm is built upon the results from others [38, 40, 3]. We recognize the importance and necessity of the work of Palsberg and Schwartzbach [38], Plevyak and Chien [40], Agesen et al. [3], Phillips and Shepard [39], however, this section will mainly deal with CPA [1]. We refer the reader to their original work or the overview presented in Agesen [1], chapter 2 and 3.
The following discussion will focus on conveying an intuitive understanding of the algorithm. Therefore specific details and theory will be disregarded and concepts and fundamental ideas will be given preference.

Although we refrain from discussing the work preceding CPA, a short history will help understand the context of the algorithm. Palsberg and Schwartzbach [38] conceived the idea to model a program as a set of constraints and to model types as sets. This algorithm was applied on a subset of the Smalltalk language. Agesen [1] calls this algorithm the **Basic Algorithm**. Agesen, working with Palsberg and Schwartzbach, developed the algorithm and converted it from a mini Smalltalk language to Self [3]. Independent of CPA's development Plevyak and Chien [40] adapted the Basic Algorithm to another programming language called Concurrent Aggregates[1]. Agesen [1] further developed the Basic Algorithm for use in the Self language. His modified algorithm is called the Cartesian Product Algorithm (CPA).

The algorithm has three steps of which the first two can be thought of as initialization. It proceeds by creating type variables for all parts of a program and defining constraints between them. The constraints define subset relationships between type variables and thus model the flow of type information in the

[1] "*A dynamically typed, single inheritance, Scheme-based, concurrent language*" [1, page 25]

program. The result of the algorithm is a graph, where the type variables are modelled as nodes and the constraints as directed edges. The graph is called a **constraint graph** and effectively models the data flow of the analyzed program.

CPA has three steps:

Step 1 – Allocate type variables: For each variable and expression in the program a type variable is allocated. This means creating a node in the graph for each. Initially, these nodes are empty, i.e. containing no types.

Step 2 – Seed type variables: The type variables are seeded with their initial type. This step ensures that the initial state of the program – before execution – is included.
Examples are variables that are defined with no value (they will be given their default value of `nil`), or variables that are declared to contain literals such as strings, integers, etc. A Ruby code example illustrating this is seen in Listing 3.4.

```
1 defaultValue # my node will contain the type of nil
2 str = "I am seeded as a string" # my node will contain the type
      String
3 number = 5 # my node will contain the type Fixnum
```

Listing 3.4: Ruby example illustrating what type each type variable for each line will contain.

Step 3 – Establish constraints and propagate: The last step is where the actual types are inferred.

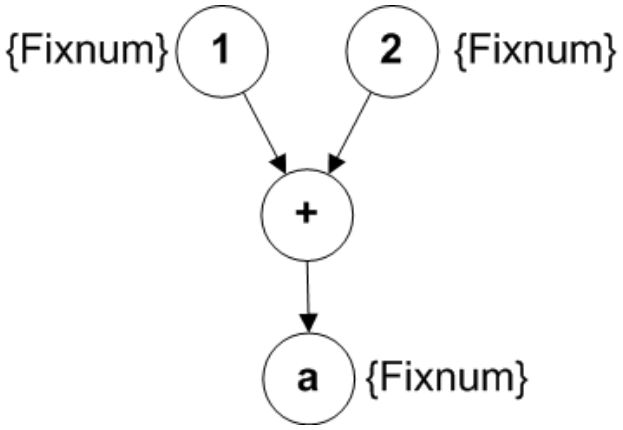


Figure 3.2: A constraint graph showing $a = 1 + 2$. The type sets of 1 and 2 have flowed into a's type set.

1. First add constraints (edges) to the graph to model the data flow. An

example is to declare a variable `a` that holds the expression $1 + 2$, i.e. an assignment. Figure 3.2 shows the resulting constraint graph. The graph contains an edge between the node of `a` and the expression, because there is a flow from the result of the expression to the value of a. Because of the flow the type set of `a` will contain the type `Fixnum`.

2. Second, propagate type information along the edges in the graph. Whenever types are added to the type set of a node, the added types will flow along the outgoing edges of the node. Suppose we have the program fragment seen in Listing 3.5:

```
1 a = 1 + 2
2 b = a
3 a = 7.5
```

Listing 3.5: Sample Ruby code for the flow graph in Figure 3.6

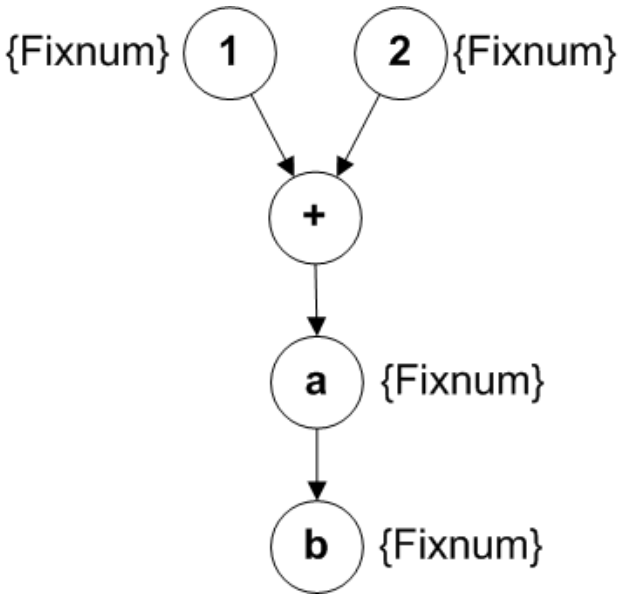


Figure 3.3: After line 2 in Listing 3.5, the type set of `a` have flown into `b`.

Line 1 would result in a constraint graph as seen in Figure 3.2. After analyzing line 2, the graph would have an added edge between node `a` and `b` as seen in Figure 3.3. The `b`'s node would contain `Fixnum` in its type set, because the type of `a` propagate along its outgoing edges into `b`.
Following line 3, the graph would get an added edge from the literal node containing 7.5 as seen in Figure 3.4. This will propagate the `Float` type to node `a`, and in turn propagate to node `b`. Since `b` does

not have any outgoing edges, the propagation will stop.
This example also illustrates a flow insensitive algorithm, which will be discussed later in this chapter.

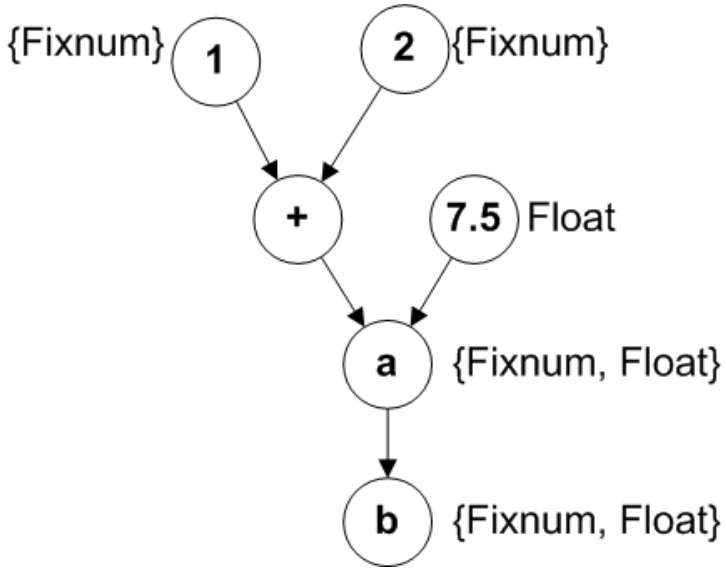


Figure 3.4: After line 3 in Listing 3.5 the added `Float` to `a`'s type set have flowed into `b`.

Propagation ensures that whenever a type is added to a node, it will flow to all dependent nodes thereby ensuring type soundness. It also implies that types are monotonically growing. Types are never removed from a type set, only added.

It is quite possible that the constraint graph will contain cycles. Therefore propagation needs to accommodate this. A simple example illustrating how a cycle might be created is seen in Listing 3.6.

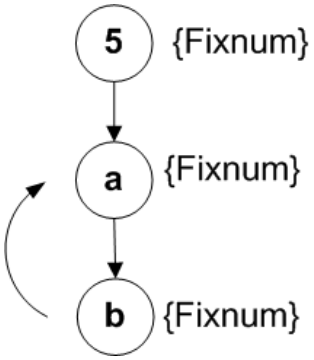


Figure 3.5: A constraint graph generated from Listing 3.6, which includes a cycle

```
1 a = 5
2 b = a
3 a = b
```

Listing 3.6: Example Ruby code that would generate a cycle in the constraint graph.

The added edge from `b` to `a` could introduce problems for propagation. However, CPA accommodates for this by guarding the propagation process. It will continue as long as the type set of the visited nodes does not contain the types currently being propagated. I.e. if a type is added to `a`'s type set it will flow to `b`, then flow along all outgoing edges of `b` returning to `a`. Entering `a` it will stop, because `a`'s type set already includes the new type being propagated. Hence, the type will have flown along all outgoing edges, and the graph will be sound and complete. Following this rule it is guaranteed that when propagation stabilizes all nodes will have the correct types in their type set.

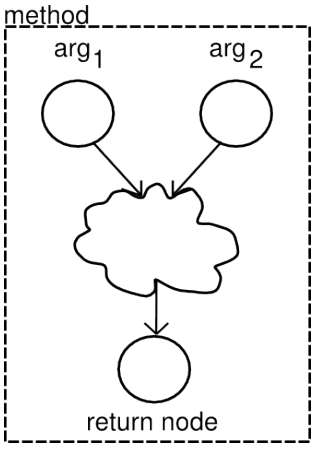


Figure 3.6: A template for a method with two formal arguments.

A program of reasonable utility would generate a large graph. Adding method calls, etc. would further increase the size of the graph. Hence, splitting the graph into different subgraphs makes it easier to comprehend and deal with. This is the purpose of **Templates**. Agesen [1] defines a template for a method *M* as the sub graph containing the nodes belonging to the definition of *M* and the edges originating from these nodes. Figure 3.6 illustrates a template for a method with two formal arguments. The cloud in Figure 3.6 illustrates the sub graph containing the nodes and edges of the code's method body. The Ruby code seen in Listing 3.7 could be the base of Figure 3.6.

```
1 def method(arg1, arg2)
2   #do something with arg1 and arg1
3   return arg1 + arg2
4 end
```

Listing 3.7: Sample Ruby code which could be the base of Figure 3.6

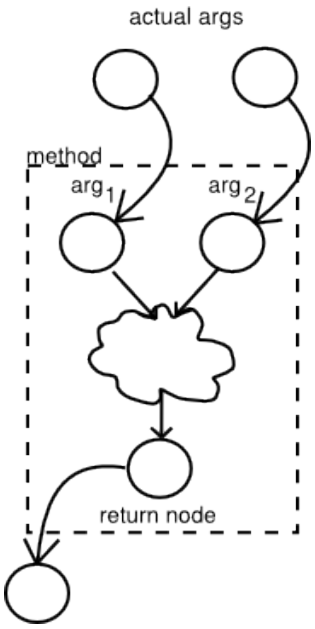


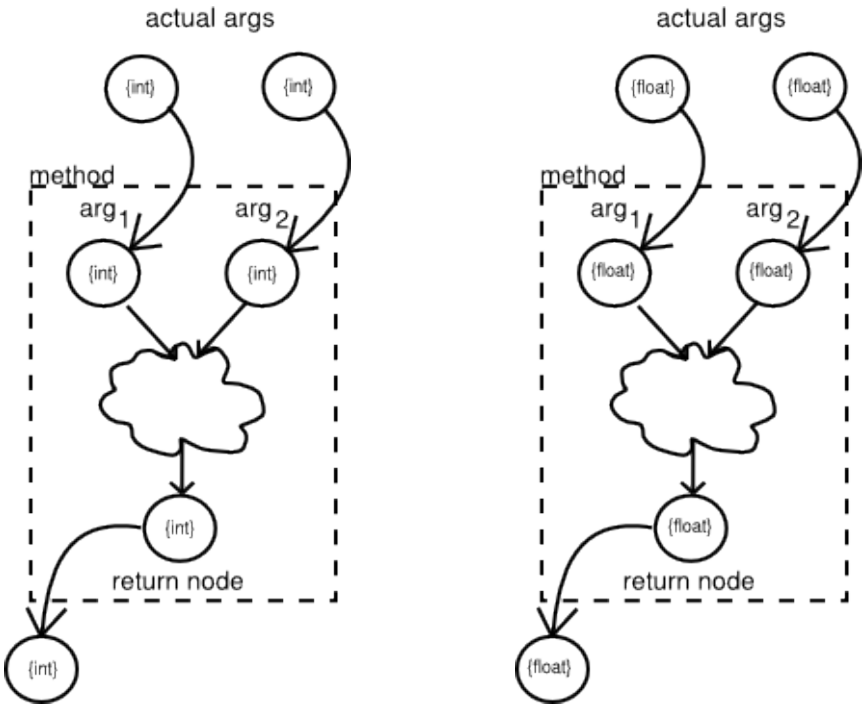
Figure 3.7: Connecting the formal arguments of a template with the actual arguments and connecting the return node.

In calling a method four steps occur:

1. Connect the actual arguments to the nodes representing the formal arguments in the template.
2. Connect the self (`this` reference) node.
3. Connect the result node in the template to the application of the methods result.
4. Propagate throughout the template.

Figure 3.7 illustrates the process of method calling. An edge is added between the nodes of the actual and formal arguments, likewise an edge is added between the templates return node and its use.

There is a relation between templates and methods but it is not necessarily one



(a) Initial call to a method results in actual arguments with type `int` to be propagated through the template

(b) Second call to the method, but now the actual arguments contain `floats`

Figure 3.8: Different calls to the same method will use a different template. This is guided by the types of the actual arguments.

to one. One method may have several templates associated. The number of templates for a method depends on how polymorphic the arguments are. With higher polymorphism comes more templates. Each template will be used to analyze a tuple of monomorphic types, i.e. a tuple with exactly one type from each of the polymorphic arguments.
The **template repository** contains all the templates used for type inference. The repository contains a list of templates used for each method in the program. Hence it contains templates for all methods in the program.
When the algorithm finds a method call it computes the cartesian product of the actual arguments types. This product contains tuples of monomorphic types. For each tuple a lookup is made in the repository for the method and a template that matches the tuples types. If none is found a new template is created, and the types are propagated. After propagation the template is saved in the repository. The next time a tuple with the same types needs a template it will reuse the one analyzed first. This reduces analysis time, and increases efficiency.
There are trade offs between how templates and methods are connected. Using

one template per method is very fast and does not require a lot of memory. However, it is also very imprecise, because types are mixed and contaminated. The other extreme with a template per method call is slow, and requires a lot of memory. It is however very precise, because types are never mixed. This implies that doing type inference using CPA is also a matter of weighing the number of templates thereby adjusting for the precision and efficiency desired.

Figure 3.8 illustrates this process on the concept of one method with two templates. First the actual arguments types are `ints` (Figure 3.8(a)), which results in a return node with type `int`. Later the method is called again but this time with `floats` as the types of the actual arguments (Figure 3.8(b)). This application results in a return node of `float`. Both templates are contained in the template repository for the method.

There is an analogy between concepts used in execution and type inference. Types are for type inference what values and objects are for execution. Likewise templates are comparable to activation records used during execution. This analogy helps present and understanding of the concepts in use, however, it is not universally true.
Taking the extreme presented above with using a template per method call, templates would be equivalent to activation records. This is due to the fact that the latter are used once and then thrown away. However, an efficient use of CPA would limit the number of templates per method. This would result in using one template per monomorphic set of types for the arguments. In this sense templates are reused where activation records are not. This also entails that templates are equivalent to a set of activation records, because during execution an unlimited amount might be created whereas during type inference only a limited number of templates would exist.

Besides the central algorithm presented above, Agesen [1] discusses the following aspects.

Dynamic dispatch resolution is the process of finding the possible methods that a given call may invoke. This is important because object oriented languages uses late binding and dynamism to achieve its characteristics.
Agesen [1] states that customizing CPA to work on another language requires a modification of the lookup rules for method calls. We will discuss this further in Chapter 5.

Dynamic Inheritance is a feature that allows an objects base class to change over time. This has implications for the object, because changing the base class may introduce different attributes, methods, etc. Likewise it has implications for the type inference algorithm. We will not discuss this part of the algorithm further,

since Ruby does not feature dynamic inheritance[2] [41]. We refer to Agesen [1], page 75 for a thorough discussion.

Inheritance is an important part of object oriented languages. It enables the creation type relations in the form of a hierarchy. Hence, it is important that CPA works for inheritance. Agesen [1], page 82 states that inheritance works out of the box in CPA because when analyzing methods the receiver is always in the context of a single class. Therefore if the method called is changed to a super class the receiver will change to resemble that class context.

Data Polymorphism means a variable can contain different types. Such examples are instance and class variables in Ruby. CPA does not deal with Data Polymorphism but Agesen [1] discusses different algorithms for achieving it.
How we deal with data polymorphism will be discussed later (see Section 5.4). Basically we let instance variables share types across class instances. So adding a type to an instance variable means it will be visible across all instances.

Blocks need special treatment because they capture the lexical scope in which the block was created and because they can perform non-local returns.

Blocks in CPA at type inference time are treated similarly to blocks at execution time. Therefore it is beneficial to remember these concepts during the following description.

CPA deals with blocks in two phases:

Phase 1 – Definition: When the type inferer encounters a block it creates a *closure object type* that binds the block with a lexical pointer to the template in which it is defined. This lexical pointer is important because it governs what is available when the block is executed, i.e. it encompasses the lexical scope of the block.
This is similar to the way blocks are implemented in programming languages. In this context it is necessary to keep the local variables coupled with the block, because they define the blocks world view when it is executed.

Phase 2 – Application: Invoking a block involves creating a new template for the block called, and copying the lexical pointer from the closure object to the newly allocated template. This way the template has access to the correct environment for the block; the environment as it was at the block definition time.

[2]Mimicking Dynamic Inheritance is possible through the use of Ruby delegation. Fulton [24], page 438 mentions it, but notes that Ruby classes can only have on parent.

Analyzing variable access in a type inference context can happen in two ways: **flow sensitive** or **flow insensitive**. The first takes into account the different types assigned to the variable and the type of the last assignment is the prevalent type. This is in contrast to the insensitive method, which takes the set of all the types of assignments made to the variable. Looking at Listing 3.8 illustrates the difference between the two methods. Using a flow sensitive method the type set of `x` in line 4 will be `Float`. Only the last assignment to `x` is preserved. Using a flow insensitive method the type set will contain `Fixnum`, `String`, and `Float`.

```
1 x = 4
2 x = "hello world"
3 x = 4.0
4 # What are the types of x?
```

Listing 3.8: Using a flow sensitive method the type set of `x` in line 4 will be `Float`. Only the last assignment to `x` is preserved. Using a flow insensitive method the type set will contain `Fixnum`, `String`, and `Float`.

Intuitively, a flow sensitive method improves precision, because at a given place only the last assigned type of the variable is preserved. However, it is also more complex than a flow insensitive method. Agesen [1] presents and discusses different methods of accomplishing a flow sensitive analysis, and we refer to his work for a further explanation.

In a prototype based language like Self there is no easy way of stating that one object is the same as another object. This is in contrast to a class based language like Ruby in which the class of an object works as a classifier[3]Working on the Self system Agesen [1] needed to analyze a large amount of objects. Therefore being able to divide the large amount of objects into smaller chunks is advantageous, because it can reduce analysis time. This division of objects is called **grouping**.

Agesen [1], page 105 presents six rules that govern into which group an object should be placed. As we focus on Ruby – a class based language – we will not discuss the grouping rules for Self. Instead we focus our attention on the rules for a class based language. Agesen [1] states that *"...classes may simplify grouping, in general they cannot fully replace grouping"*.
For a class based language Agesen [1] presents the following rules:

- Two objects belong to the same group if they are direct instances of the same class.
- Two objects belong to the same group if their contents [instance variables, ed.] pair wise belong to the same group.

[3]With respect to Ruby there are other issues as well pertaining to duck typing, however, for this discussion it is sufficient to note that the class does function as a classifier.

- Two list objects are in the same group if the sets of groups of their elements are equal.

Ultimately one aspect governs the grouping rules for a programming language: the type system's semantics. This stems from the fact that the type system states what constitutes equivalence on a meta level. For a discussion of how types are understood in Ruby see Section 3.1.1. Furthermore this aspect will be covered in a later chapter with an emphasis on how we have accomplished grouping (see Section 5.4).

A program may change an object significantly, which during type inference would require it to be placed in another group. This process is called **regrouping**. Agesen [1] does not present a method for doing so, but notes that it should be possible.
We will continue the talk on regrouping in a later chapter (see Section 5.4. However for now we can summarize that we do not add anything significantly to Agesen [1]'s discussions.

Recursion requires special treatment in CPA, because in the worst case it can cause non-termination. The problem is that a function calling itself recursively may require the creation of new templates for the function. The creation of a new template will be required if the lexical environment is different for the inside call. This can occur when calling the function with a block i.e. a closure. Agesen [1] calls this recursive customization, because of the continuing customization of the inside closure's lexical environment that sparks the creation of new templates.
Eliminating this requires the introduction of a cycle, so the inside call refers to the outside function, thereby breaking the recursive creation of new templates. Agesen [1] uses heuristics to discover a potential recursion and then introducing a cycle. We refer the reader to Agesen [1], chapter 5 where recursive customization is extensively discussed.

The type information discovered after running CPA can be used in a number of ways. Agesen [1] describes the process of *"Sifting out the gold"*, which has its background in the Self system. The purpose is to extract the objects required for an application, and leave the rest. In a sense it is like compacting the Self image, so only the application required is left.
Another application is a static checker that checks for flaws in the program. A part of this is checking for message not understood errors, an equally important aspect in Ruby programs. The second deals with browsing source code of object oriented systems, thereby increasing understandability and maintainability. The last application is for an optimizing compiler.
These are suggestions made by Agesen [1] of which some are implemented. However, this list is far from complete. The following chapters will elaborate

on our use of the discovered type information.

Related Work

This section briefly discusses projects that use CPA.

Starkiller [48]: The overall goal is to make Python programs run faster. Salib [48] uses CPA to create an optimizing compiler for Python. He lists a number of contributions he has made to CPA, which are:

Recursive Customization: Starkiller adds a number of silent arguments (out of scope variables) to the template. This helps CPA in choosing the right template.

Extension Type Description Language (ETDL): Allows type inference on foreign code written in C, C++, and Fortran. The ETDL is code written in Python that with respect to types mimicks what the foreign code does.

Tainted Containers: Given an initialized array containing (`"abc"`, `1`, `2.0`), the type inference system knows that the element with index 0 is a `String`. When it encounters a use of this element it will return this single type instead of the collective type set ⟨ `String`, `Fixnum`, `Float` ⟩. However if the container gets sorted the system is no longer aware of this. The container becomes tainted and the collective type set is returned.

Data Polymorphism: Starkiller deals with data polymorphism by splitting types of classes when different types are assigned to its instance variables. This was not part of Agesen [1]'s work.

Closures: Python does not allow non-local returns in closures, and therefore Starkiller handles closures differently.

Ultimately benchmarks show that StarKiller – although not complete – compiles code that performs almost as good as C code and significantly better than other Python compilers.

AnalyseJ[6]: A Java project for doing type analysis using the CPA. The last Concurrent Version System (CVS) commit for the project was in 2002.

Perl Request For Comments (RFC)[21]: describes an RFC for the Perl 6 language. Its premise is that *"types should be inferred whenever possible"*[21]. The preliminary idea is to use CPA as algorithmic method.
The RFC seems inactive as of late 2000. Further research into the RFC has not yielded an update on its current status.

Python TI[8]: A master thesis by Cannon [8] about an extension he developed to the Python compiler. His idea is to emit type specific information into the compiled byte-code, thereby gaining a performance increase. However, his goal of a 5% performance increase is not met, instead he achieves 1%. His system uses CPA coupled with iterative type analysis as described by Chambers and Ungar [12].

TI Ruby[33]: This project is a predecessor to the current report. It modifies CPA to work for Ruby. However, some parts of the Ruby language are missing (arrays, switch expressions, loops, singleton methods, exceptions, importing libraries, and Ruby core). TI Ruby states that building a type inference system for Ruby is possible.

In summary CPA has been used for TI on the following languages: Java, Python, Perl (envisioned), Ruby, besides Self for which it was developed.

3.4 Summary

This chapter laid the ground for an understanding of types and type inference. Important concepts which will be used throughout the report was defined by example. This included a discussion of how types can and are understood in Ruby. We defined the difference between data and parametric polymorphism, and explained type checking and general type inference. The chapter ended with a presentation of two algorithms for type inference: The Hindley-Milner algorithm and the Cartesian Product Algorithm (CPA). Understanding CPA is a prerequisite for this thesis, and the rest of the report documents the development of a type inference tool for Ruby using CPA.

CHAPTER 4

Problem Statement

This project has two purposes: first to develop a code analysis tool for Ruby, second to perform experiments on public Ruby code using the developed tool. This chapter sets the ground for the second part. To guide the development and help establish the types of experiments to be performed a set of hypotheses will be given. During the course of this thesis we will try to confirm these hypotheses, and the tool developed is a key factor in this.

Furthermore a set of design goals are presented that puts perspective and context on the first part of the purpose of the project.

4.1 Hypotheses

Hypothesis 1: Ruby programmers do not make type errors.

> In "The Development of Erlang" Armstrong [5] writes about his and a colleagues findings on performing a type check of the standard libraries in Erlang; they found no type errors. Preliminary he concludes that "good programmers don't make type errors". Furthermore he writes that their intuition on the types of certain libraries were proved correct. This illustrates that dynamic programmers may have a good understanding of the types in their programs.
>
> Cronqvist [15] writes about his experiences in developing a large industrial application ($>$ 2 million Lines of Code (LOC)) using Erlang. He describes the three parts of an Erlang development testing cycle. First, the programmers does Block Testing, which detects "most run-time errors". Furthermore it uncovers bugs that "in many cases would be found by the compiler of a statically typed language". Second, Function and System Testing is performed, during which all "corrections to code are documented in Trouble Reports (TR)". Analyzing these reports yielded 4 groups of errors of which the following 2 are of interest[1]: API mismatches and Typos.

[1] The remaining two deals with concurrency (race condition and wrong context errors), which is relevant for Erlang programmers and the articles context but not for a type inference system.

Interestingly Armstrong [5] notes, *"Many programs behave correctly despite the fact they are not well-typed."* This suggests that in some respects languages with implicit types are able to express concepts that are not easily expressed in a language with explicit types.

Hypothesis 2: Industry does not believe in Ruby. Can we help drive adoption of Ruby into industry by providing tool that can infer types for Ruby programs?

Traditionally industry uses statically typed languages. One of the reasons is that they help find typing errors, and the reasoning is that therefore they are safer. Because there is no concept of compile time type checks of most dynamically typed languages, industry is reluctant to adopt Ruby.

Erlang is a dynamically typed language used by the telecommunication equipment manufacturer Ericsson. Armstrong [5] writes about Ericsson's experiences with using such a language, and they are very positive. This leads to the conclusion that dynamically typed languages do have a place in industry. But does Ruby?

A tool that could perform a type check on Ruby code might help adoption of the Ruby language in industry. Therefore it would be interesting to see if a tool like this could help adoption in the longer run.

TIOBE Software [55] publishes a list once a month of the most popular programming languages. On the top ten list, of June 2007, 4 languages are statically typed and 6 languages dynamically typed. The top three languages are Java, C, and C++. Ruby comes in at number ten for June 2007.

Our assumption is that if a tool is available for inferring types, industry is more likely to adopt a dynamic language typed like Ruby.

Hypothesis 3: Programmers using Ruby are more productive.

Prechelt [42] performs a comparison between 7 different programming languages divided into scripting and conventional languages. The comparison showed that for a specific programming problem scripting languages were more productive than conventional languages. Conventional langauges are comparable to statically typed languages and Prechelt compares Java, C and C++. Scripting languages are comparable to dynamically typed languages, where he uses Python, Perl, Rexx and Tcl. Ruby belongs to the latter category.
Prechelt [42] writes, *"Designing and writing the program in Perl, Python, Rexx, or Tcl takes no more than half as much time as writing in C, C++, or Java and the resulting program is only half as long"*.

Cronqvist [15] writes about his results in developing large systems using Erlang. He notes that after the *"initial code-debug cycle"* the errors remaining are largely due to *logical or algorithmic errors"*. He suggests that this is due to the programmer not having to deal with low level issues but rather that he can focus on the problem to solve. This further increases productivity.

Hypothesis 4: Real life Ruby programs uses polymorphism restrictively.

Our assumption is that there is a correlation between how polymorphic a program is and how likely that the program contains type-errors. Answering this hypothesis will help answer hypothesis 1 if Ruby programmers make type errors. This hypothesis can be divided into the following two sub-hypothesis:

Hypothesis 4.1: Data polymorphic variables are used sparingly.

Ruby facilitates the use of variables to store different values. It is our hypothesis that even though it is possible, Ruby programmers uses this sparingly. Programmers do not use data-polymorphic variables, because they increase the complexity of the program, and obscure what value a variable store at a given time.

Hypothesis 4.2: Method calls exhibit limited polymorphism.

Even though Ruby programmers may define methods with parametric polymorphism, method calls often exhibit limited polymorphism. This hypothesis follows the pattern of Hypothesis 4.1 in that a high degree of polymorphism in method calls increases the complexity of the program.

Hypothesis 5: CPA can be retrofitted and used on the Ruby language.

CPA was developed for the Self system (see Section 3.3). Chapter 2 compared Ruby to the Self language and concluded that the two languages are similar in many ways. Their semantics (inherited from Smalltalk) are sufficiently similar to warrant the hypothesis that CPA can be used for Ruby. Furthermore, CPA has previously been used for Starkiller [48] – a static type inference for Python code (see Section 3.3). This suggests that modifying CPA to run in another language context is possible, i.e. CPA is not completely tied to the Self language.

The hypotheses presented above have varying scope, i.e. some are easier to

confirm than others. Some of the hypothesis have a scope that do not make it realistically that we can confirm them through the work presented in this thesis, they are demanding in both data collection and time. Other hypothesis are more limited in scope and can be confirmed or rejected with metrics of Ruby Programs that, given a type inference tool could be collected for a Ruby program. A discussion of our success of confirming each hypothesis is presented in Chapter 8

The primary aim for this thesis is to answer the hypothesis stated above. Answering these hypothesis will help put Ruby into a larger perspective. A number of secondary objectives will be presented in the coming section. The secondary objectives give a different perspective on the hypothesis, and help formulate specific goals, challenges, and requirements for this thesis and the developed tool. Furthermore these objectives will complement the hypothesis in conveying an understanding of the premise for this thesis.

4.2 Goals

Agesen [1] gives an implementation of a method for inferring types in a dynamic object oriented languages. We are interested in exploring this method, to examine if this method is a general approach or may only be used in the context of the original language, Self.

We wish to convey the experiences we have gained from implementing CPA for Ruby. Related to this we wish to validate that CPA works for Ruby and where it should be modified to work in a Ruby context.

Another goal is to extract metrics from the analysis of Ruby programs. These metrics will serve to confirm or reject the hypotheses presented above. The hypothesis will be discussed further in Chapter 8.

4.3 Challenges

We have identified three areas that we believe will prove a challenge for the type inference tool. The three areas are:

Conditional Control Flow: Conditional Control Flow constructs represent a challenge as the method developed by Agesen [1] is flow insensitive. As CPA is flow insensitive we do not know which of a conditionals branches we have to evaluate.

Continuations: Ruby has support for continuations with the use of the `callcc` method. What is of special interest here is that the execution state of a program is preserved. To infer the most concrete types the CPA must be aware of the call stack for methods and take this information into account.

Dynamic Programming: Ruby facilitates dynamic programming. Ruby's dynamic language features include the ability to add or update methods of instantiated objects. From Agesen [1] it is not immediately clear how this is to be handled. Another aspect is how the dynamic nature influence how types are to be understood.

4.4 Requirements

The following presents a set of requirements for the developed type inference tool. As per the goals these requirements have been chosen to support the hypotheses presented above.

Coverage: The tool must work on real life applications and not just constructed toy examples. This requirement stems from the second hypothesis, because if the tool does not work on real life application industry will never utilize it. Second, to be of use in answering if Ruby programmers make type errors (hypothesis 1) and the degree of polymorphism (hypothesis 4), the tool must work on real Ruby programs.

It also follows from the above that the tool must work on Ruby's Core and Standard Library to be of use.

Robust: The tool must not crash even if languages construct not implemented by the type inference tool are encountered. A tool that is not robust can not be expected to be adopted by industry. This is a second requirement in making industry adopt Ruby.

Precision: The inferred types must be as meaningful/specific as possible in a given context (see Chapter 3). The two algorithms discussed, Hindley-Milner and CPA, are different in the kind of types inferred for a program. In Hindley-Milner the most general types are inferred, whereas the most specific types are inferred using CPA. The more specific a type is the larger a set of errors can be caught by a type checker.

4.5 Quality Requirements

The following quality requirement definitions are from Firesmith [22]. The Quality Requirements provide a different view of the tool's goals as presented above. Therefore the individual goals presented above will be used in defining the meaning of the individual requirement.

Robustness: *representing the degree to which essential mission-critical services continue to be provided in spite of potentially harm-causing events or conditions* [22].

Given a piece of valid Ruby code – valid meaning parsable by the official Ruby interpreter – the tool must complete without errors. If language constructs are not supported is encountered, the tool should recover from it and complete and provide meaningful output.

Performance: *a timing characteristic* [22].

We do not have any specific time constraints on the developed program, but acknowledge that for it to be usable the running time must be coupled with the size of the project. This means that for small programs we want a running time in at most a few seconds. By a small program we mean a program with a few hundred lines of code.

Efficiency: *the degree to which something effectively uses (i.e., minimizes its consumption of) its computing and personnel resources* [22].

The tool should be efficient with respect to time and space, however, it should favor running in less time rather than using less space/memory.

Correctness: *specifies a minimum required amount of the quality factor correctness* [22].

Correctness of the developed tool can be divided into the following two sub-requirements.

Precision: *the dispersion of quantitative data* [22].

To be able to use the inferred types, the types must be as precise a possible. As a measure for the precision of an inferred type the size of the resulting typesets is used. The smaller the size of the typeset the more precise the is the inferred type.

Accuracy: *the magnitude of defects (i.e., the deviation of the actual or average*

measurements from their true value) in quantitative data [22].

We will require that the types inferred by the tool are the most specific types possible in the given context. If a variable of a program only is use as an `Integer`, the accuracy of the inferred type should be `Integer`.

4.6 Usage Scenarios

With a developed tool that can infer types for Ruby code, the following list contains examples of what the type information can be used for, or how it can be used.

Type Checker: With types available for a program it is natural to check if the types are used correctly through the program. Having type available enables catching potential `NoMethodError` exceptions. As it is discussed in Chapter 3 `NoMethodError`'s are the only type errors reported by the Ruby interpreter.

Integration with IDE: The developed type inference tool will make the type information available to a program but does not present this information to a user. There are different approaches to present the type information. Below are the different approaches that are under consideration listed.

- **Type Annotation in Source Code:** Type information can be annotated directly in the source code and presented in an IDE. This approach resemble the experience that users of explicitly typed languages have while retaining the capabilities of a dynamically typed language.
- **Tree View With Annotated Types:** Presenting the type information in a hierarchy closely mirroring the syntactic structure of the program, i.e. an Abstract Syntax Tree (AST) representation of the program.

Documentation: A benefit of presenting the type information at a source code level is that the types function as documentation. This effect could make the code's intentions clearer and help catch logical errors earlier in the development process.

4.7 Summary

This chapter established the formal requirements to this thesis. This was formulated as a set of hypotheses, which will help guide the development and experiments performed. Furthermore a set of requirements and goals for the thesis and tool was presented. This included quality requirements and usage scenarios for the tool.
The requirements formulated in this chapter will form the basis of the experiments described in Chapter 7 and the discussion in Chapter 8.

CHAPTER 5

Implementation of Ecstatic

This chapter describes our implementation of CPA called Ecstatic. It operates on and performs type inference off Ruby programs. The implementation is based on the presentation made in Chapter 3. The description serves to explain the experiences gained in implementing CPA and provide and implementation perspective on the work of Agesen [1].
We conclude the chapter by listing the capabilities and deficiencies of Ecstatic.

5.1 The Type Inference System

Ruby programs are executed by a Ruby interpreter. An interpreter typically has a built-in environment resembling a virtual machine that facilitates the execution of the program and a parser/code generator that converts the code into instructions that carry out the program. The environment of a Ruby interpreter contains the built-in features of the language and a symbol table to register and lookup defined classes, modules, methods, variables and so on. The parser typically parses the source code into an AST representation, and the code generator traverse this tree while generating instructions.

When doing type inference we do not want to execute Ruby programs, only analyze them. To do this, we have created a system similar to a Ruby interpreter that builds a constraint graph for type inference instead of generating instructions for execution. In analogy to an interpreter, the system has a "virtual machine" called *RubySim* (Section 5.1.2), a parser borrowed from a real interpreter (Section 5.1.3), and a *Controller* instead of a code generator (Section 5.1.4). Before going into a detailed description of these three central parts, we give an overview of the entire system in Section 5.1.1.

5.1.1 System Overview

This section presents a brief overview of the Ecstatic type inference system, which is shown in Figure 5.1. We do not expect the reader to fully understand the figure or the following description at this point, but we present it here to put

the different parts of the system into perspective. The reader may benefit from returning to this figure later on when the individual parts have been described in detail.

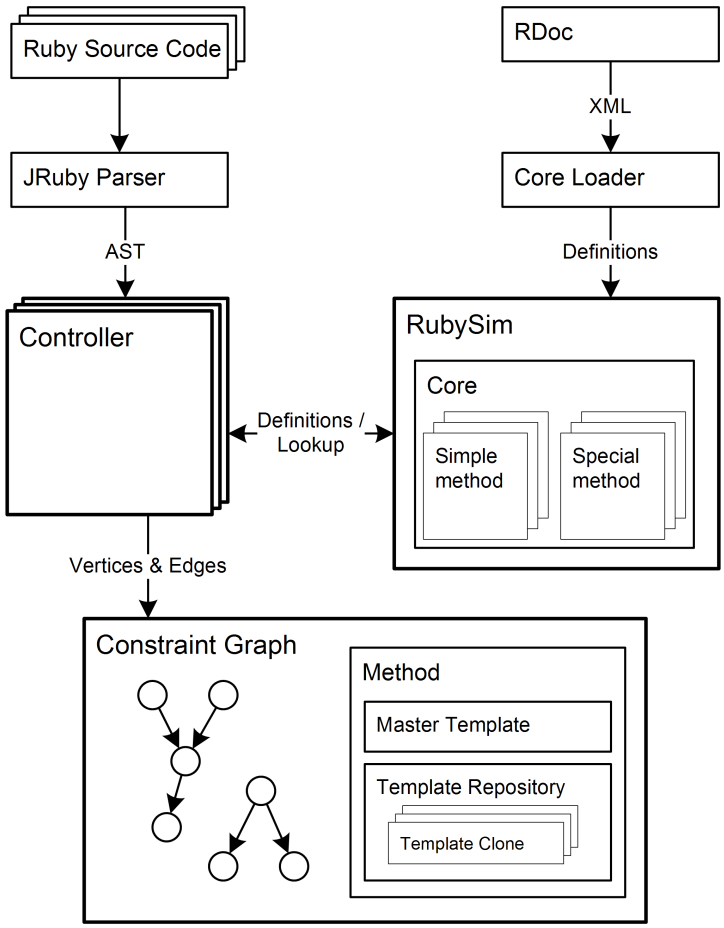


Figure 5.1: Overview of the Ecstatic type inference system.

Starting in the upper left corner, a Ruby program of one or more source files is fed to the JRuby parser. The parser produces an AST for each source file. These are passed on to the Controller, where the type inference process starts. The system creates one Controller per AST (or per source file).

Before the Controller starts traversing the AST, Rubysim is initialized. This is done by the Core Loader, which parses a number of XML files generated from RDoc (Ruby's documentation format). Based on these XML files, the Core Loader generates a simulation of the Ruby core consisting of both simple and special methods. The workings of the Core Loader is described in Section 5.1.2.

When Rubysim is initialized, the Controller begins to parse the AST. While do-

ing so, it uses RubySim to define and lookup classes, modules, methods, and variables. To do type inference, the Controller builds a Constraint Graph by adding vertices and edges to it. The vertices represent CPA's type variables and the edges represent constraints.

When the Controller encounters method definitions, it creates these methods on the constraint graph. These methods are described in detail in Section 5.2, but basically a method consists of one master template, one template repository, and a number of template clones. All templates are subgraphs on the constraint graph.

5.1.2 RubySim

RubySim is short for Ruby Simulator. The name comes from the idea that instead of executing Ruby programs, it only simulates the behaviour of programs. It is responsible for modelling the semantics of Ruby regarding objects, classes, modules, methods, scopes, variables and constants.

RubySim works like a virtual machine with a set of instructions for:

- *Opening and closing class, module, method, and block definitions.* These definitions require opening a new scope in which subsequent definitions and lookups are rooted. Closing a definition returns to the enclosing scope. It works like pushing and popping things on and off a stack.
- *Creating objects.* Objects are created by instantiating a previously defined class.
- *Defining variables and arguments.* Different types of variables are created differently. Local variables are added to the current local scope, while instance and class variables are added to the currently opened class or module definition. Arguments are added to the currently opened method or block definition.
- *Find previously defined variables.* Variables are found according to the scoping rules of Ruby.
- *Finding the target methods for messages (dynamic dispatch resolution).* When an object receives a message, a lookup is performed to find a method that matches the message.
- *Aliasing methods.* Ruby features the possibility of aliasing a method with a new name.

- *Including modules.* Modules can be included as mixins into a class or module definition.

These instructions are used by the Controller while analyzing a program, but also by the Core Loader to inject the core functionality into the system.

The Ruby Core

The core functionality of Ruby is implemented in C as a number of classes and modules. When programming Ruby, these classes and modules are available and treated the same way as classes and modules defined in Ruby code. Therefore they have to be incorporated into the type inference system in a way, so that classes and modules, whether implemented in C or Ruby, integrate seamlessly. Even though the source code of the core classes and modules is freely available, it cannot be parsed through the type inference system. The type inference system can only handle Ruby code and not C code. Instead we have implemented these core classes and modules directly in Java code.

The core contains 34 classes and 14 modules that together implement 1312 methods. All of these methods must be available in RubySim when type inference starts. Else, the type inference process will break if one of these methods are called. Implementing a simulation of all these methods in Java by hand is a huge task. We therefore face a dilemma: on the one hand, we need the methods; on the other hand, we have no desire to spend the time it takes to implement all 1312 methods manually. We would prefer to only spend a minimum of time on these core methods and then focus our effort on the rest of the type inference system. However, we realize that some of the core methods require an accurate simulation in RubySim, and therefore will require some effort. Based on these thoughts, we decided to create a solution that automatically generates an approximated implementation of the core, while simultaneously allowing for accurate simulation of selected parts.

Before explaining how we did this, we take a closer look at the methods in the core. From a type inference perspective, they can be divided into three groups:

1. *Fixed type methods* (without side effects). This group includes all methods with the following two properties: they have no side effects, and they always return a fixed type no matter the input they are given. Examples are, `to_s` that always returns a `String`, `to_a` that always returns an `Array`, and `length` that always returns an `Integer`.
2. *Variable type methods* (without side effects). This group includes all methods with the following two properties: they have no side effects, and their

return type depends on the type of the input they are given. For example, the return type of + implemented on `Fixnum` depends on the other part of the addition. `4 + 5` (`Fixnum + Fixnum`) yields a `Fixnum`, whereas `2 + 4.3` (`Fixnum + Float`) yields a `Float`.

3. *Side effect methods.* This group includes all methods that have side effects. Their return type may either be fixed or variable. Methods with side effect are important in Ruby, because they are responsible for implementing much of Ruby's semantics. Many of them modify objects or alter the execution environment in some way. For example, mixins are included by the method `include`. Class attributes are created by the methods `attr_reader` and `attr_writer`. In Section 5.1.4, we show how the method `require` is used to load additional source files.

From browsing through the core documentation, it seems that a vast majority of the methods belong to the first group. Fortunately, we have found a way to autogenerate the methods in the first group, which we will explain later.

The principle of importing the entire core into RubySim is as follows: All methods are autogenerated as if they were fixed type methods, except those methods for which we provide a special implementation. In this way, we ensure that all methods are available in RubySim. Having all the methods, we are able to run type inference on any program without it breaking because of a missing method. However, the result of the type inference at this point may be wrong if some special implementations are missing.

Because we can autogenerate the methods in group one correctly, we only need to provide special implementations for the methods in group two and three. Even though group two and three constitute a minority of the core methods, they still represent maybe hundreds of methods. The only way to be sure on this number is to manually check the behaviour of all 1312 methods. We do not want to do that, either. Instead, we have chosen a kind of stepwise refinement approach. By having dealt with Ruby for some time, we are naturally aware of some of the methods that definitely need a special implementation. We therefore started by creating special implementations for these methods. Apart from that, we made the system in such a way that it is easy to add more special methods when necessary.

The process of creating special implementations can happen incrementally as follows. Initially, we start out with special implementations for a few methods, while the rest of the core methods are treated as fixed type methods. We then run the type inferencer on a program and observe the result. If we identify a wrong behaviour, it might be because one or more side effect methods need a special implementation. We then create these implementations and run the type

inferencer again. If the behaviour is still not right, we repeat the process. When the code starts to behave correctly, we might instead identify some wrong type information. If one or more variable type methods seem to be the cause of this, we can create special implementations for those too.

By proceeding incrementally like described above, we can focus our effort on only creating special implementations for the most essential and most often used methods. Thereby, we avoid spending time on implementing methods that are unimportant or never used, which is a great advantage over having to implement all of them. Ideally, all methods in group two and three should have a special implementation to behave correctly. But since the developed type inference system is not yet complete, the stepwise refinement approach described above was considered the optimal compromise. Having said this, we now return to the discussion on how we autogenerate fixed type methods.

Ruby has a documentation format called RDoc [43] with which the core classes and modules are documented. Amongst other things, the documentation specifies the return type of all methods. This means that the documentation contains all the details needed to implement the fixed type methods in the type inference system. By this observation, we came up with the idea of autogenerating these methods based on the documentation. To do that, we have processed the documentation into an XML-format (see Appendix B) and by parsing this format we can import the core into RubySim. The documentation also describes all classes and modules together with their superclasses and mixins. This information is used to automatically build Ruby's class hierarchy into RubySim.

The process of importing the core into RubySim is as follows:

1. The XML documentation is parsed by the Core Loader.
2. For each class or module found, it creates this class or module in RubySim. If mixins are defined, these are also included.
3. For each method found, it does the following:
   By use of Java's reflection mechanisms, it checks if the Java package `tiruby.rubysim.core` contains a class with the same name as the method.
   (a) If it does, an instance of the class is created and stored in RubySim.
   (b) If not, an instance of the class `FixedTypeMethod` is created. The instance is initialized with the return type of the method and then stored in RubySim.

To summarize, all methods that have a special implementation become instances of their respective Java classes, while all other methods become instances of `FixedTypeMethod`. In this way, the Core Loader loads the entire Ruby core into

RubySim by autogenerating classes, modules and methods from RDoc documentation. At the same time, it allows for stepwise refinement of the core behaviour by implementing selected methods explicitly. This concludes the discussion of the Ruby core except from a couple of notes.

For some core methods, RDoc also indicates the types of the formal arguments. However, this type information is too inconsistent and inaccurate to be used as a basis for type inference and checking. Therefore, we do not check if the core methods are used with the correct types for parameters.

The Ruby core is not the only place where C code classes and modules are found. Programmers can create their own C implemented extensions for Ruby, for example to speed up performance or integrate with certain hardware. In theory, this user defined C code could be imported into the type inference system in a similar way as the core. We do not provide an API for that.

5.1.3 Parsing Ruby

Before talking about how Ruby programs are parsed and represented in the type inference system, we want to take a closer look at what defines a program. In this area, Ruby is radically different from Self. Self is an image-based system, whereas Ruby is source code-based. A Self image contains a heap of objects. Agesen explains that type inference in Self is done by analyzing these objects directly, but he is silent about how this is done. He defines a program as a chosen main method on a chosen object, and program execution as the computations that results from invoking this main method.

A Ruby program consists of one or more source files. Ruby programs are normally organized by having one of the source files recursively includes the rest. This means, that one source file includes other files that again include other files. This continues until all source files of the program have been included. The source file that starts the inclusion is called the main source file. We discuss how file inclusion is handled in Section 5.1.4.

Execution of a program is done by passing the main source file to a Ruby interpreter. The interpretation of a Ruby program starts in the body of a class definition—more specifically, the definition of a singleton class for an object called `main`, which is an instance of the core class `Object`. Because class definitions in Ruby are executed, the content of the main source file is processed procedurally line by line.

The first step in doing type inference for Ruby is to parse the source code. This process builds an AST from the code, which is a good representation of the code

to base the type inference system on.

Instead of writing a parser ourselves, we have chosen to use the one built into JRuby [14]. As cited from their website: *"JRuby is an 100% pure-Java implementation of the Ruby programming language."* It implements an interpreter for Ruby currently compatible to version 1.8.5 of the language. The JRuby project is open source and gives a good look into the inner workings of a Ruby interpreter. More importantly, the AST it produces is a Java data structure, and can easily be imported into a new Java project for the type inference system.

Parsing the Ruby code in Listing 5.1 gives the AST in Figure 5.2. The figure is a screen shot from a small tool we have created called AstBrowser. The names of the nodes in the AST are the same as the names of the corresponding Java classes.

A depth-first walkthrough of the AST is as follows: The `BlockNode` is an implicit container for the rest of the program. The `LocalAsgnNode` represents the assignment to `s` in line 1, which assigns the `StrNode` with the text `"World"`. The `NewlineNode` separates the expression in line 1 from the expression in line 2. The `FCallNode` represents a message with an implicit receiver, which means that the receiver is the Ruby `self`. In this case the receiver is an instance of Ruby's `Object` class, and the message sent is `puts` from line 2. The `ArrayNode` works as a list of arguments to the message. In this case there is one argument, the `CallNode`, which represents a message with explicit receiver. The receiver is `StrNode`, which has the text `"Hello "`. The message is named `+` and has one argument, `LocalVarNode`, which refers to the local variable `s` defined in line 1. When executed, Listing 5.1 results in concatenating `"World"` to `"Hello "` and printing it to the screen.

```
1 s = "World"
2 puts "Hello " + s
```

Listing 5.1: Ruby code

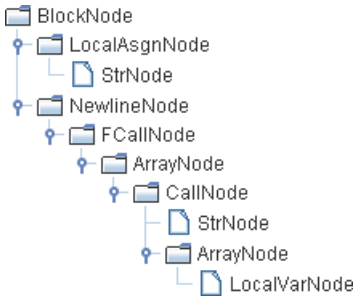


Figure 5.2: AST

5.1.4 The Controller

When the main source code file has been parsed, the control is left to the Controller for the rest of the type inference process. The Controller is handed the generated AST and is responsible for applying the CPA algorithm on it. It does that by traversing the AST and performing the three basic steps of CPA for each expression and variable found in the AST. Before the three steps are described in detail, we will investigate how the AST is traversed.

JRuby internally implements a visitor pattern to traverse the AST. Their implementation is naturally geared towards generating instructions for execution, and cannot as such be used directly for type inference. However, the visitor pattern is based on a Java interface that allows us to implement the visitor pattern differently. This is what the Controller does. The visitor pattern interface defines 107 methods, one for visiting each kind of AST node. The Controller implements these methods to perform type inference on the Ruby program instead of creating instructions for execution. The actions performed by the Controller will become clear as we go through the three basic steps of the CPA.

The first step of the CPA is to allocate type variables to all variables and expressions of a program. Basically, this is done by traversing the AST representation of the program to locate variables and expressions and allocate a type variable for each. For each AST node representing a variable or an expression, it would be natural to allocate the type variable directly on the AST node. However, because the AST nodes are generated by the JRuby parser (see Section 5.1.3), they cannot be modified without changing the parser. Instead, we create what we call a *vertex* object that has a reference to the AST node and contains a type variable. In this way, a type variable is associated with a variable or an expression through an object that references both of them (see Figure 5.3). The type variable is represented as a set and is initially empty. Vertex objects also serve as nodes on the constraint graph (actually, we use the word *vertex* for nodes on the constraint graph).

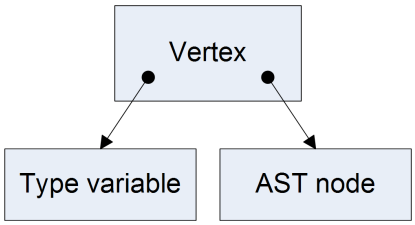


Figure 5.3: The structure of a vertex object. A vertex references both a type variable and an AST node, and thus associates a type variable with the variable or expression represented by the AST node.

In terms of the Controller as a visitor, it starts by visiting the root node of the program. The method for visiting the root node then specifies what other nodes to visit (typically, the child nodes of the root node). The Controller continues in this way until all nodes have been visited and the entire program have been analyzed. For each variable or expression it visits, it creates a vertex object as described above.

Step two of the CPA is to initialize the type variables. Most variables and expressions do not have an initial type, so most type variables remain empty after this step. The only ones affected are literals such as strings, integers, floats, arrays. This step is carried out simultaneously with step one. While traversing the AST as in step one, all nodes representing literals are located. When the Controller visits a literal, it first creates a vertex (step one) and then adds the type of the literal to the vertex's type variable.

The third step is the most complex of the three. As mentioned in Section 3.3, the step can be divided into two: creation of constraints, and propagation of type information. Constraints represent the flow of type information from one type variable to another. The question of which constraints should be added was touched upon in Section 3.3 and will not be further investigated here. Instead, we will show an example of how the Controller creates a constraint for an assignment expression. After the example, we will discuss how propagation is done.

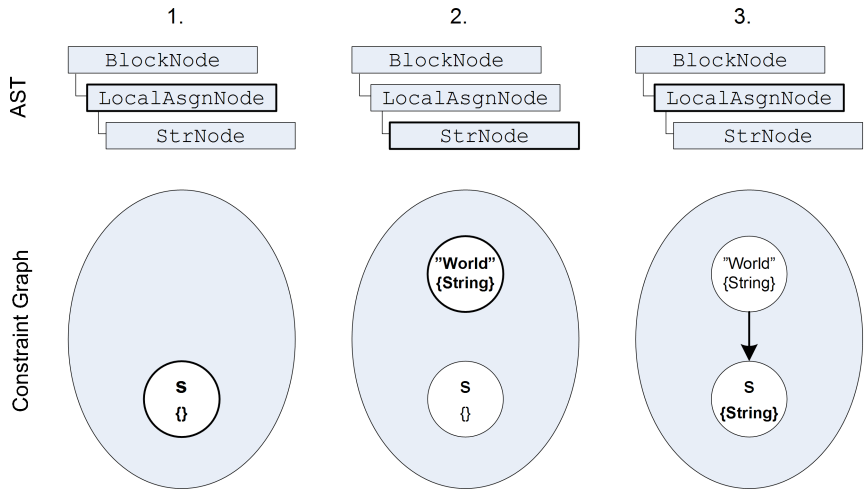


Figure 5.4: This figure shows how the Controller handles a local assignment in three steps.

In Listing 5.1, line 1, a string is assigned to a local variable. The corresponding AST in Figure 5.2 models this as a `LocalAsgnNode` that has a `StrNode` as child. The Controller handles this assignment in three steps as illustrated in Figure 5.4.

1. The Controller starts by visiting the `BlockNode` and then the `LocalAsgnNode`. In the `LocalAsgnNode`, the Controller creates a vertex for the local variable `s` (CPA step one). The type variable is left empty because the variable `s` has no initial type (CPA step two).
2. Knowing that an assignment expression has a value node, the Controller now visits the value node. Visiting the `StrNode`, the Controller now creates a vertex for the string (CPA step one) and initializes the type variable with the type `String` (CPA step two). No constraints need to be added here (CPA step three).
3. The method visiting `StrNode` exits by returning the string vertex back to the `LocalAsgnNode`. All visitor methods have the possibility of returning a vertex, otherwise they return `null`. The Controller is now back in the `LocalAsgnNode` where a directed edge (representing a constraint) is added from the returned vertex to the vertex of `s` (CPA step three). It is now time for the second part of step three: propagation.

Propagation

Agesen suggests propagating types eagerly to ensure type soundness. Eager propagation means, that as soon as an edge is added, type information should be propagated along it. When more and more edges are added to the constraint graph, the type information flows further and further through the graph. Type propagation continues until the graph stabilizes and no more types need to be propagated. In this section, we discuss how this is done in the Controller and the complications that arose.

In any single propagation step there is a source vertex and a target vertex. Type propagation is started when an edge is added between them. The type information is then propagated from the source vertex to the target vertex. If the type variable of the target vertex already contains the type information from the source vertex, propagation stops. No more needs to be done before another edge is added. If, however, the type variable of the target vertex changes, the new type information must be propagated further to the *children* of the target vertex. The children of a vertex are the vertices that have an incoming edge from that vertex. The type information is then propagated to each of the child vertices. If the type variables of any of the child vertices change, the process continues for their children until the situation stabilizes. Cycles in the constraint will naturally stop because the type variables eventually stop changing.

There is one problem with eager propagation when it comes to message-sends. Suppose we have created a call vertex representing a message that has two arguments. After that, the Controller goes on to visit the receiver and the arguments

and add edges from them to the call vertex. If it first visits the receiver, then after returning from the receiver, an edge is added from that to the call vertex and type information is propagated. When a call vertex receives new type information, its normal behaviour is to lookup targets for the message, propagate type information through the found templates, and add the result of the templates to its type variable. At this point, however, the edges from the arguments to the call vertex have not yet been added. The situation would therefore end in a missing arguments error when trying to invoke the method (i.e., propagate through the template). To avoid this, a call vertex has a ready-switch that can be switched off until all arguments are connected. As long as the ready-switch is off, the call vertex will not perform its normal action.

There is a second point in time during type inference where propagation is disabled. The establishment of constraints and propagation is the analysis-time equivalent of runtime execution. In this sense, it is important that eager propagation only occurs on parts of the code that should be executed immediately. As mentioned earlier, the body of a Ruby class is executed while the class is being defined, but the body of a method definition is not. Therefore, during method definition, the eager propagation is disabled. In this way, we can build a template that captures the initial state of a method before it is called. Section 5.2 explains how this template can be cloned to provide additional templates for analyzing message-sends monomorphically. The discussion of propagation through templates is continued in that section.

Handling `require`

As mentioned in Section 5.1.3, a Ruby program may consist of more than one source code file. Typically, a program is split up into small files containing some chunk of self-contained functionality. This is also the case with the Ruby standard library. To make use of these external files, the main source file must load them by sending the message `require` with the filename as argument. The corresponding `require` method is implemented in the core class `Module`. The method includes the external file into the current scope, except for local variables in the external file that are not propagated to the current scope. The `require` message can be sent at any point in the code, but typically it is sent at the beginning of a file. If an included file contains further `require` messages, additional files are included. However, a file already included once will not be included again.

To perform type inference on anything but the smallest Ruby programs, it is essential to facilitate the inclusion of additional files. The `require` method is implemented in the C core, but as mentioned in Section 5.1.2, the Core Loader makes it possible to create special implementations for selected methods. This is what we have done with the `require` method on class `Module`. When the method

is invoked, it first checks if the file to be included has been included previously. If not, the file is parsed by the JRuby parser and a new Controller is spawned to traverse the resulting AST. The old Controller hands the control over to the new Controller and waits until the new Controller is done. Then the control is handed back and the old Controller resumes the process.

During the switch between Controllers, the constraint graph and RubySim remains the same. Because all Controllers work on the same constraint graph and use the same RubySim, the state and context of the type inference process is preserved from one Controller to another. In other words, under the entire type inference process there is only one constraint graph and one instance of RubySim, and these are shared between controllers.

5.2 Methods and Templates

In Section 3.3, we explained how the CPA analyzes polymorphic message-sends monomorphically. It does so by computing the Cartesian product of the receiver type and all argument types. This generates a set of monomorphic tuples, each representing a monomorphic message-send.

Suppose that the lookup of a given message has resulted in the target method *m*. Agesen [1, page 58] now instructs that each monomorphic tuple must be propagated through a separate *m* template (a template for the method *m*). If an *m* template already exists for a given monomorphic tuple, it should be reused; if no such template exists, a new one should be created and used. Any newly created templates should then be stored in the template repository of *m* for later reuse. However, Agesen [1] does not explain *how* this should be done.

There are two things we do not know: how Agesen represents methods and how he creates templates. Our educated guess is, that methods are not represented in any other way than their raw form as objects in the Self image. How a template repository gets associated with a method remains unclear. If methods are "raw" objects, then these objects must be analyzed each time a template is created.

In Ecstatic we wanted to avoid re-analyzing the AST each time a new template is needed for a method. To do that, we use the concept of *master* templates and template *clones*. The two kinds are basically the same, both being subgraphs on the constraint graph, but they play different roles. When the type inferece system encounters a method definition, it builds a master template for that method. The master template then works as an intermediate representation of that method, and that method definition is never analyzed again. A master template captures the initial state of a method before being called, and type

information is never propagated through it.

As mentioned above, a new template is needed for each monomorphic use of a method. Because a method's master template is already on template form, it can easily be cloned into a new template, thus creating a template clone. A template clone is used for a single monomorphic send, and a method's template repository contains a clone for each monomorphic use of the method. In summary, a method has one master template, one template repository, and a number of template clones.

By cloning templates we avoid re-analyzing the AST each time a new template is needed. Instead, we represent methods as master templates, which must be clonable. A master template is a subgraph with a set of vertices (type variables) and edges (constraints). Cloning is done by first creating a new template as an empty subgraph. Then each vertex on the master template is cloned and added to the template clone. While doing this, we maintain a *vertex map* between all the original vertices and their corresponding clones. When a vertex is cloned, the clone is initialized with the same type variable and AST node reference as the original. In this way, the template clone holds the same initial state of a method as the master template does.

When all vertices are processed, the template clone contains the same number of vertices as the master template, but no edges. An edge connects a source vertex to a target vertex. For each edge in the master template, the source and target vertices are looked up in the previously created vertex map to find the two corresponding vertex clones. An edge is then added between the vertex clones. When all edges have been copied, the template clone contains the same number of edges as the master template.

So far, the process of cloning a template has been uncomplicated; it is done by cloning each vertex one by one and adding the corresponding edges. However, the process is complicated by vertices that are nontrivial to clone. For example, a *call vertex*, representing a message-send, internally maintains a *call context* of which other vertices are the receiver and the arguments of the message. After cloning a call vertex, the call context of the clone still references the uncloned vertices. As a consequence, we now have a vertex in the template clone that references vertices in the master template. To fix this, the vertices of a call context must be swapped with the corresponding cloned vertices using the vertex map. This process is illustrated in Figure 5.5.

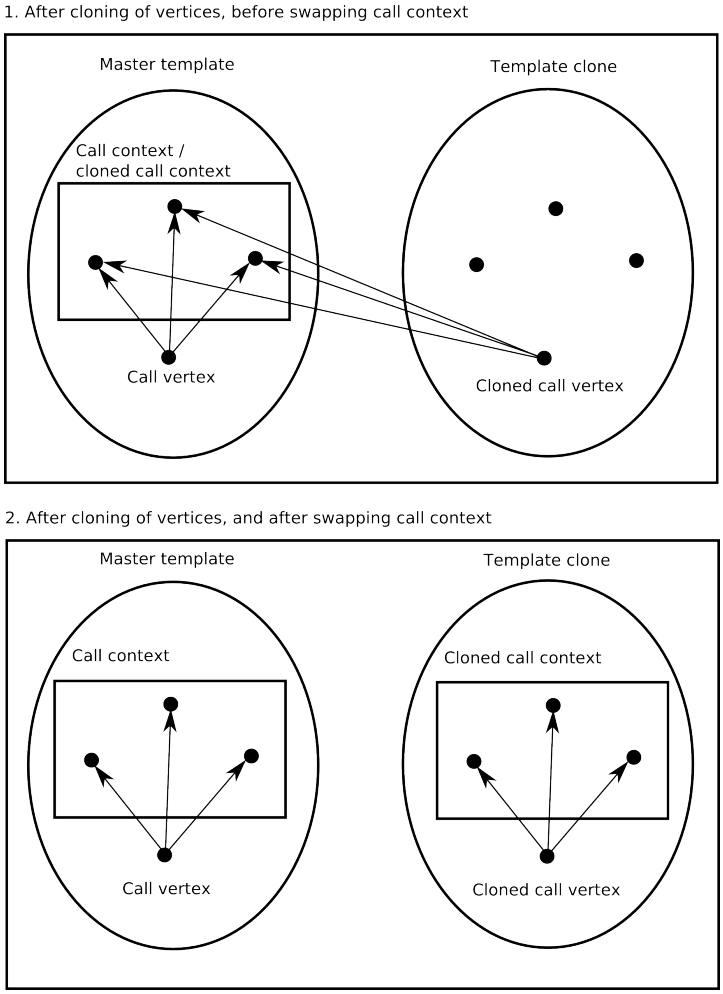


Figure 5.5: This figure shows the situation before and after swapping the vertices of a cloned call context.

5.2.1 Optional Arguments

Ruby offers the possibility of supplying default values to the formal arguments of a method definition. If a formal argument has a default value, it is *optional* for the caller of the method to supply this argument and thereby override the default value. Hence the name, optional arguments, which is something Self does not have.

We handle optional arguments through the use of master templates and template clones. If a method has an optional argument, the default value of the argument is naturally included in the master template. More specifically, an edge is added from the default value to the optional argument. At this moment, we do not know whether any later clones of the master template will use the default value or override it. As mentioned, type information is not propagated through master templates, and therefore, the default value does not "pollute" the type information in the template. When the method is called and a template clone is created, the clone is only used for that specific call of the method. If the caller supplies a value for the optional argument, the following occurs: The default value of the optional argument is simply removed from the template clone and so is the edge that connected it to the optional argument. Instead, the value specified by the caller is connected to the optional argument, just as with non-optional arguments. This process is illustrated in Figure 5.6. If the caller does not supply a value for the optional argument, no special action is taken because the default value is already connected. When the type information eventually is propagated through the template clone, it works correctly in either of the two cases.

5.2.2 Propagation Through Templates

When the Controller analyzes a method definition, it builds a master template for that method. Types are not propagated through master templates because they capture the initial state of a method before it is called. When a method is called, template clones are created, and the types of the method arguments must be propagated through the template clones. The challenge is then how to propagate type information through a template. Agesen does not explain how this is done in CPA; he just instructs that it must be done.

In Section 5.1.4, we explained the approach of eager propagation, where types are propagated immediately as edges are added. But in a template clone, edges are already added and propagation can therefore not follow the order in which edges are added. As a basis for discussing how propagation can be done, we present an example in Listing 5.2. The example shows the definition of a method

```
def sum(x = 1, y = 2)
  return x + y
end


sum(2.3, 3.2)
```

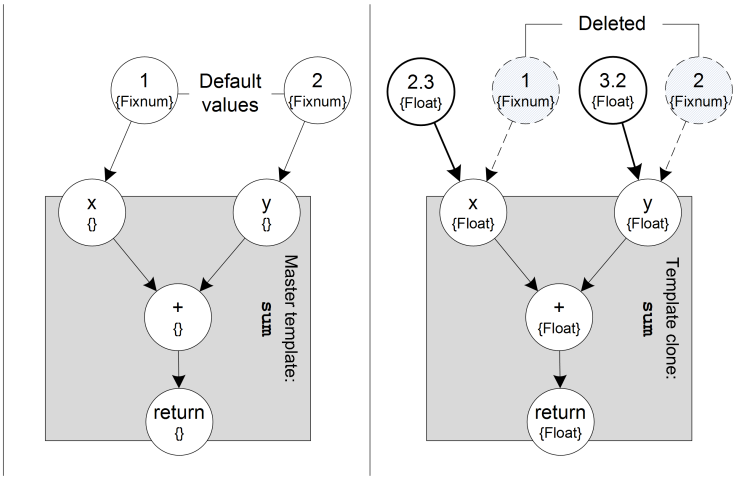


Figure 5.6: To the left is a method definition of `sum` with two optional arguments. This method is called with both arguments supplied. The middle part shows the master template of `sum` with the default values of the optional arguments connected. To the right is the template clone showing `sum` when called. The default values are deleted from the template. Instead, the actual arguments of the call are added, and the types are propagated.

`method1` and a message-send that targets the method in line 7. When the Controller processes the code, it starts by creating a master template for `method1`. When the method is invoked in line 7, the master template is cloned and the actual arguments `3` and `4` are connected to the formal arguments `x` and `y` of the template clone.. Furthermore, the `return` vertex of the template is connected to the variable `c`. Figure 5.7 shows the template clone of `method1` with the proper edges added. The type variables have been seeded with their initial types, but the type information has not yet been propagated. We have experimented with three different ways of propagating types through templates, which we will discuss here.

```
1 def method1(x, y)
2     $a = x + y
3     $b = "b"
4     return 3.3
5 end
6
7 c = method1(3, 4)
```

Listing 5.2: Definition and call of method1

1. *Propagate from the arguments and down.* The idea of this approach is to propagate types in the same order as the method would be executed at runtime. Propagation will start in the vertices `3` and `4`. The `Fixnum` types will flow

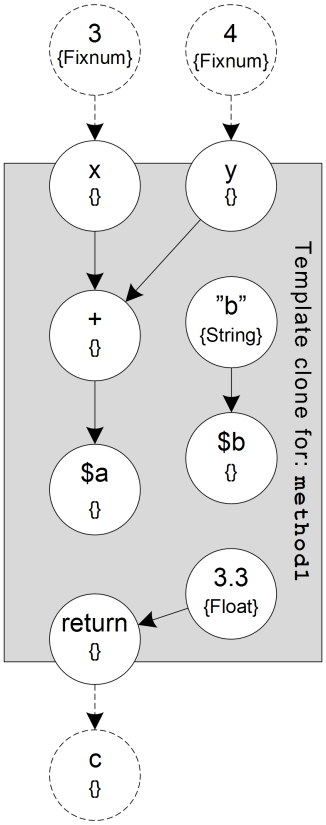


Figure 5.7: Template clone for method1 in Listing 5.2 just before propagation.

into `x` and `y` and down to the addition in `+`. When the addition message has been processed, the resulting type will flow into `$a`. Then the propagation will stop, because `$a` is not connected to the rest of the vertices. As a consequence, the `String` type will not be propagated into `$b`, and the `Float` type will not be propagated into `return` and from there to `c`. The type variable of `c` will thus remain empty, which is wrong because the method should return the type `Float`. Yielding a wrong result, we had to discard this approach to propagation. Furthermore, if a method has no arguments, this approach is not a valid solution.

2. *Reverse propagation from the return vertex and up.* We tested this approach as a way of ensuring that the result of a method is resolved correctly, thus solving the problem of the former approach. It proceeds as follows: Starting in vertex `c`, it resolves the "parents" of `c` – that is, the vertices that influence `c`. In this case, the only parent is `result`. It then continues by resolving the parents of the parents, and in this way crawling backwards up the graph. When it reaches a vertex that has no parents, it starts a normal propagation from this vertex, which then propagates all the way down to

the result. In our example, this happens in the vertex `3.3`, which has no parents. The `Float` type is then propagated down through `return` and into `c`. The propagation then stops with `c` having the correct type. However, no types have been propagated into the two global variables `$a` and `$b`. Because globals variables can be used in other contexts as well, this behaviour is wrong. We could try to apply the former approach also, which in our example would ensure that types are propagated into `$a`, but `$b` is still untouched by both approaches.

3. *Start propagation in all vertices.* This last approach is a kind of brute force approach that does not pay attention to the order of propagation. It iterates through the set of all vertices in the template and starts propagation in each vertex. In this way, we ensure that all types are propagated – even the `String` type into `$b`, which was not reached by the two former approaches.

Apparently, the third approach seems to work correctly, but a brute force approach is never a satisfying solution. There must be a more elegant way. At least, we believe that Agesen has solved the problem more elegantly. Or maybe he did not even experience this problem in the first place... Pondering this question, we realize that we might have misunderstood how Agesen handles templates. In the following, we discuss what we believe went wrong and how to fix it.

Agesen [1, page 42] writes as follows:

> The inference algorithm determines the send's type by propagating its actual argument types through the template(s) of the method(s) that the send may invoke.

Here, and a number of other places, he uses the wording: to propagate types "through" a template. Other places, he uses the word "into" instead of "through". His descriptions paint a picture of first having a complete template, and secondly propagating types through it. If this is the case, we do not know, but maybe the picture should be painted differently. We believe, we have found a different and very simple solution to this templates and propagation problem.

As mentioned in the beginning of Section 5.2, Agesen probably builds new templates directly from the method objects in the Self image. At the time when a new template needs to be created, the Cartesian product of the message arguments has already been computed. For the template to be created, we already know the monomorphic argument types that must be connected to it. Instead of first creating the template and then connecting the argument types, it could be done the other way around. To explain this, we switch to a Ruby context and

use the Controller as an example. Because the template is created at the time when the method is invoked (or executed), eager propagation can still be used and does not have to be disabled as we do during the creation of master templates. The Controller could start by creating vertices for the formal arguments of the method. It could then add edges from the actual arguments to the formal arguments and propagate the types immediately (eager propagation). The Controller should then continue through the method body and create the rest of the template while eagerly propagating types. When it reaches a return point in the method, the type of the return is already known (by eager propagation) and can be propagated back to the caller of the method. Summarizing this approach, the types are propagated through a template *while* it is created, instead of *first* creating the template and *then* propagating the types.

The approach described above does not incorporate the use of master template, because templates are created directly instead of being cloned. However, because of the problems with handling references when cloning templates, we recommend skipping the use of master templates in future work and instead use the approach described above.

5.3 Keeping Track of `self`

When a method is invoked and types are propagated through a template, it is often necessary to know the type of `self`. For example, if a template contains a message-send without an explicit receiver, the receiver of the message is `self`. Intuitively, one could say that the type of `self` in a method body is just the class that the method is defined on. But because of subtyping, this is not always the case. Let us say that the class `Sub` is a subclass of class `Super`, and `Super` implements the method `m`. If an object of type `Sub` receives the message `m`, the message is forwarded to `Super` because `Sub` does not implement a method with the name `m`. When the message reaches `Super`, the method `m` is invoked. The type of `self` inside the body of `m` is now `Sub`, the receiver of the message, instead of `Super`, the class of the method. As a consequence of this, we need some way of determining the type of `self` in the template of a method.

To keep track of the type of `self`, we initially tried to maintain a stack of the type of `self` inside RubySim. RubySim was then responsible for at any given moment during type inference to provide the correct type of `self`. This was quite a challenge, and just as we thought we had gotten a hold of it, some obscure call sequence proved us wrong. Eventually, we decided to try another approach.

We realized that `self` can be viewed as an extra formal argument of a method.

Correspondingly, the receiver of a message is an extra actual argument that should be connected to `self` on the invoked method. With this insight, we created an extra vertex on the templates as a "formal" `self` argument. The receiver as well as the actual arguments of a message can be polymorphic, and therefore the receiver must be included in the computation of the Cartesian product. In this way, the first argument in a monomorphic tuple is connected to `self` and the rest are connected as usual. When `self` is stored on the graph like this, we no longer have to lookup its type in RubySim and source of the problem with keeping track of `self` is removed.

5.4 Types and Grouping

In Section 3.3, we presented the following grouping rules for objects from Agesen [1]:

1. Two objects belong to the same group if they are direct instances of the same class.
2. Two objects belong to the same group if their contents [instance variables, ed.] pair-wise belong to the same group.

Based on the discussion of types in Ruby in Section 3.1.1, we use the immediate class of an object as its type. Coupling this with the first of the two rules, we do not distinguish between groups in types. In other words, we group objects by their type.

If two objects of the same type have pair-wise different types in their instance variables, Agesen suggests by the second rule above to put them in separate groups. We have chosen not to do this by the assumption that instance variables are rarely used polymorphically. I.e., we assume that two objects of the same type will rarely put objects of different types in the instance variables they have in common. In this way, we treat instance variables in the same way as class variables, which made the implementation much easier. In Section 7.2.2, we follow up on this assumption based on a number of experiments we have conducted.

5.5 The State of Ecstatic

Even though Ecstatic can handle a major part of the Ruby language, a few areas are left for future work. In this section, we list the capabilities and incapabilities

of Ecstatic.

Ecstatic can currently handle classes and inheritance, modules and mixins, methods (singleton, instance, and class methods), optional arguments, array arguments, parallel assignment, local variables, class variables, instance variables (treated as class variables), global variables, constants, conditional expressions, loop expressions, inclusion of additional source files (`require`), attribute creation, and import of the Ruby core. The entire Ruby core is automatically generated, and 42 core methods are provided with special implementations that simulate them correctly according to Ruby's semantics.

Ecstatic does not currently support:

- *Recursion.* When a method calls itself, the propagation mechanism enters an infinite loop. This should be trivial to fix by detecting the loop and breaking it. However, the related problem of recursive customization is non-trivial to fix. But based on Agesen [1] implementing it should be possible.
- *Most kinds of blocks.* In Chapter 8, we present a discussion of blocks in Ruby. There are four semantics for blocks, and only one of them is currently supported. The rest of the blocks are not trivial to implement. The lack of proper block support is a serious issue with Ecstatic because blocks are used very often in Ruby.
- *Block arguments.* A method may specify a special block argument as one of its formal arguments. When a block is passed to such a method, it is automatically converted to a `Proc` object, which can be accessed inside the method by the block argument. If all kinds of blocks were supported, this feature would be trivial to implement.
- *Some side effect methods in the Ruby core.* We have identified 42 core methods that needed special implementation, but there are probably more. By our stepwise refinement approach, it is easy to add more implementations when the need arises.

Finally, a few of the language constructs supported by Ecstatic are still buggy. Fixing this requires further testing and debugging.

A preliminary edition of the Ecstatic tool can be seen in Figure 5.8. The screenshot shows the source code of a small Ruby program called `SimpleMethod`. To the right, the program's AST generated by JRuby is seen. Figure 5.9 shows the same program, but with the types infered by Ecstatic annotated in the program code.

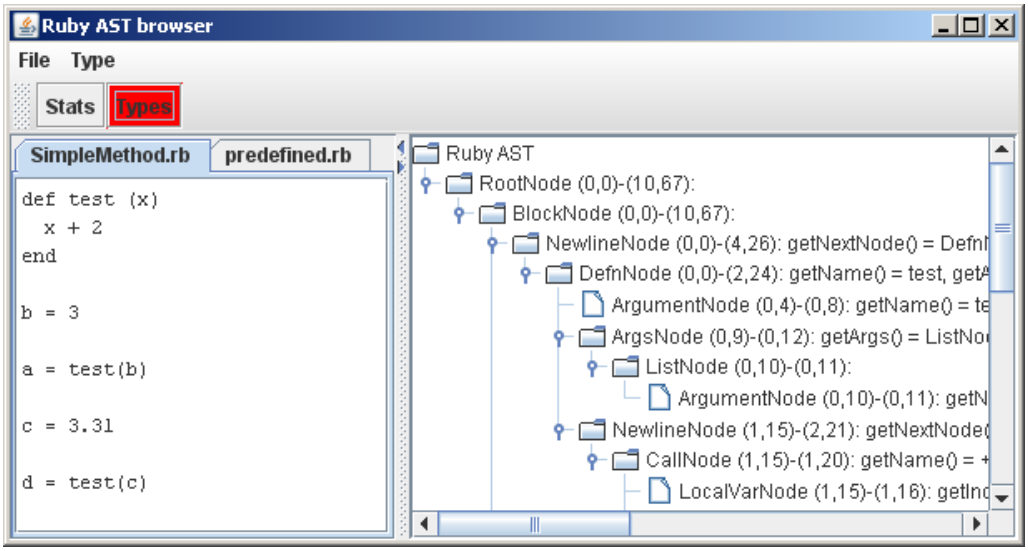


Figure 5.8: Ecstatic showing the source code and AST of a small Ruby program called `SimpleMethod`

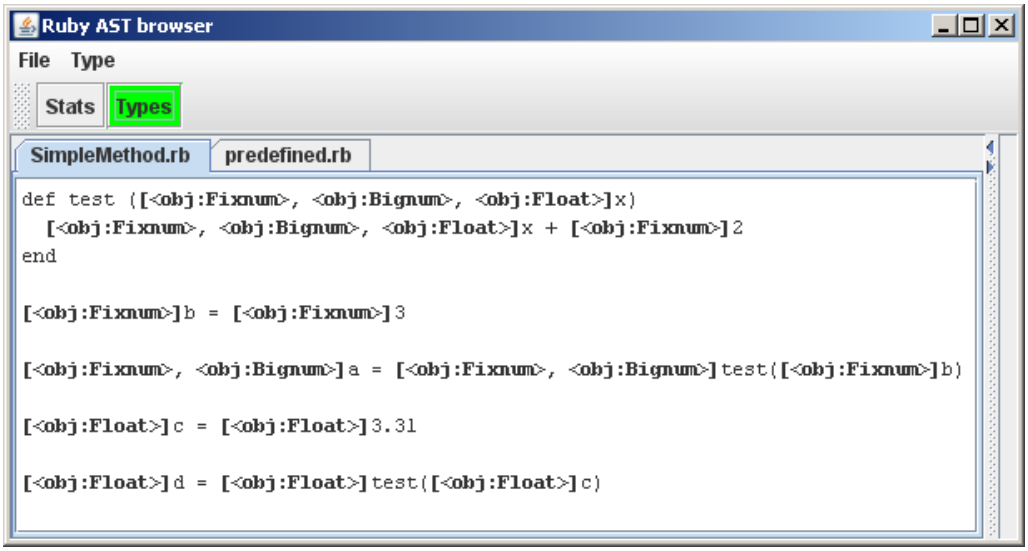


Figure 5.9: Ecstatic showing the source code of the `SimpleMethod` program with annotated types.

5.6 Summary

This chapter described the implementation of Ecstatic – a tool for type inference for Ruby. Ecstatic is based on CPA. A primary part of the implementation story has been to gather experience on how CPA can be implemented.
Ecstatic features the Ruby Simulator `RubySim` and a Controller implementing CPA and the constraint graph. We present a simple way to inject the Ruby core into the system to increase the precision of the type inference process.
In summary, Ecstatic supports a substantial portion of the Ruby language, however, work remains to be done.

CHAPTER 6

Testing Ecstatic

Testing is an important part of developing software. As size and complexity of a software system increases the need for automatisation of the testing process also increases. Having an automated testing system can help during development, by checking that a positive change does not mean that another previously fixed defect reoccurs.
The project on SW9 illustrated the need for a formalized testing process. It was often found that as development of the type inference system proceeded and more and more language features became supported previously supported constructs stopped working[33]. One of the purposes of the testing process is to remedy this, and ensure that changes made during development does not break previously implemented features.

A software system similar to the type inference system developed is a compiler. Both take source code, parse it, and produces an output. For a compiler this is executable code, for the type inference system it is type information.
Two groupings of compiler testing is validation and verification. The former checks that given an input the correct output is computed. The latter formally verifies that the software adheres to a specification, i.e. a language definition[68]. The two methods are similar to the difference between *black and white box* testing respectively. Using a validation based method you test the system as an opaque unit in which you cannot see the details of the implementation nor use this to your testing advantage. Verification is similar to a transparent box in which you know the details of the implementation and use this to formally prove the systems correctness.

From formal verification it follows that a formal apparatus for the system needs to be in place. This implies that the language needs to be specified in a rigid manner suitable for proofs. This is not the case for Ruby, which is solely specified in the compiler created by Yukihiro Matsumoto [19]. The Ruby community is working on creating a specification for the language [62], and Yukihiro Matsumoto has started the process of documenting Ruby in the context of the *Design Game*[26].
Ultimately, however, it means that formally verifying the type inference software system is currently infeasible. Therefore focus will be placed on validation.

This chapter documents the testing performed on the type inference system.

First a discussion and survey of methods for compiler validation and how it relates to type inference software will be performed. This presentation is largely based on Ada's and Pascal's compiler validation suites. This will form the basis for the testing effort, process and system in place for Ecstatic.

6.1 Research on Compiler Validation

In "The Ada Compiler Validation Capability" Goodenough [25] explains the compiler validation suite created for Ada. It is in essence a black box testing style, and hence cannot "detect all programming errors"[25, page 1]. Goodenough [25] poses a set of questions as to how and what should be tested, and the conclusion is to use *"many small tests"* that supports an *"evolutionary development of a test set"*[25, page 4]. A classification of tests is presented that divides the individual tests into groups; these are:

Class A: Tests that should compile, but the result may not be executable

Class B: Illegal programs that should be rejected by the compiler

Class C: Programs that compiles and runs

Class L: Illegal problems, but the illegality is detected at link time

Class D: Capacity tests, for example how many identifiers can a program use. Vendor specific and there is no specified lower limit defined in the standard.

Class E: Ambiguities in the standard. Each vendor creates a test that illustrates how they treat the ambiguity.

The process for running the tests is also described. It is emphasized that this must be specified and be tailored to automatic runs.

Tonndorf [56] presents the status of Ada's Conformity Assessments, which is an updated term for the validation suite presented in Goodenough [25]. Furthermore he discusses how the assessments can serve as a model for other contemporary programming languages (C/C++ and Java). He explains the history of the assessments and the roles of various standardization organizations. The test suites from two vendors for C and C++ is briefly analyzed and the difference between their and Ada's purpose is noted. Ada places "increased emphasis on negative tests (Class B tests)"[56, page 94], and the reason is that Ada has a strong focus on a formal testing procedure. Compiler vendors focus is primarily

on performing self tests[1] on their individual compilers. The survey of one vendors test suites for different languages follow a common pattern: each suite has a conforming and a testing part[56, page 95]. The first deals with how well the implementation conforms to the programming languages standard as defined by some party. The latter with actual testing of the compiler and defect finding.

In "Pascal Compiler Validation" a collection of articles is assembled from a conference in February 1982 [61]. It deals with the validation suite for Pascal, which is based on the work done on the Ada programming language. This includes articles on structure of the suite, test classification, and the validation process.

In summary the results of the research on compiler validation made above are:

Black Box: Validation as a testing practice for compilers is useful and practical. However, viewing the compiler as a black box, and hence not know anything about the implementation details, makes it difficult if not impossible to catch all errors. Thus, validation as such is not a 100% solution. [25]

Test Types: Properly testing using validation requires the use of both positive and negative tests. This implies that testing needs to focus both on what should work, but also on what should not work.

Classification: It is helpful to divide tests into a predefined classification. Ada does this, and this practice has been modified in the Pascal compiler validation suite. It helps in defining and narrowing the focus of a particular test. Furthermore it provides an overview of the test suite.

Small Test: The evaluated validation suites uses many small tests in favor of few big tests.

Automatic and Process: The validation suites all emphasize the value of a defined validation process and automatic execution of tests. This is especially important in their context since the tests are used in validation laboratories. Nevertheless, having a defined process and automization is beneficial.

This list of recommendations will serve as a guideline for the performed testing effort described in later sections.

[1] Tests used during development to ensure that the compiler continues to function and adheres to the vendors interpretation of the standard.

6.2 Relation to Type Inference

To use the results of the compiler validation discussion presented above, the relation between a compiler and a type inference system must be established. Both systems can act as a black box, and they both use source code as input and yield a specific output. A compiler outputs executable code, whereas the type inference system generates type information based on the input. In some respects the type inferer is a special case of a compiler.
The type inference software uses parts of a compiler (JRuby) to accomplish its tasks. Specifically the lexer and parser used to generate an AST. JRuby is extensively tested by its maintainers, so we assume it is correct. Therefore care must be exercised to avoid testing the lexing and parsing part of JRuby and instead focus on validating the type inference algorithm.

A key difference between compiler validation and its relation to type inference is the starting point. The compiler validation efforts start with a standard of the compilers target programming language. Based on this common standard a validation suite is created. For the Ruby type inference software this is different, simply because there is no standard for Ruby (at least not a described one). The validation suite created for the type inference software must therefore be based on intricacies in the CPA algorithm and the available information on the semantics of Ruby (see Thomas et al. [53], Ruby Community [47]).

6.3 Validation Suite

The validation suites goals are based on those recommended in Section 6.1.

Test Types: A wide range of tests will be created, this includes both code and type definitions that should work and code and type flows that shouldn't work.

Classification: A classification of tests will be defined and used to categorize each test. This will also help focus on a tests main purpose.

Small Tests: Focus will be put on creating many small tests each testing a specific part or feature, compared to creating few big tests.

Regression: The suite must support being used as a regression test tool during development. This will help ensure that newly implemented features does not break previously supported aspects.

The tests are grouped in two major groups each containing a number of sub groups:

Language or Library Features: These tests deal with specific language constructs or libraries. Examples are blocks and arrays respectively.
The purpose is to test singular aspects of the language or library.

This group has tests within the following sub-categories:

Instance Variables: Tests language constructs related to instance variables. Including attribute accessors (see Section 2.1.5).

Ranges: Tests language constructs related to ranges.

Blocks: Tests language constructs related to blocks.

CPA: Tests CPA intricacies such as propagation.

Sample Programs: This group's tests programs that utilize several of the features tested in the "Language or Library Features" group.
The purpose is to test the interaction between different elements of the language.

This group currently contains samples created during development. They are our exploratory tests of how Ruby works, and were used to get a better understanding of Ruby.

6.3.1 Test Suite Framework

Creating and executing the tests needs to be simple and quick, and preferably utilize an existing unit test framework. JUnit[37] fits the bar, and was hence chosen as the base.

In general a method is required that enables the possibility to check that certain parts of the tested program get the correct types. To accomplish this the *assert* pattern in JUnit was expanded. The desired functionality is to assert that certain identifiers has a type set that includes, exactly is, or excludes a specified set of types. Furthermore, it is convenient to assert that a typeset is empty.

The overall pattern of executing a unit test (represented by a class) is this (illustrated in Figure 6.1):

1. Given a Ruby file as input, run CPA on it.
2. Run a number of tests on the resulting constraint graph

(a) Each test executes a number of asserts to verify that identifiers have the correct type set.

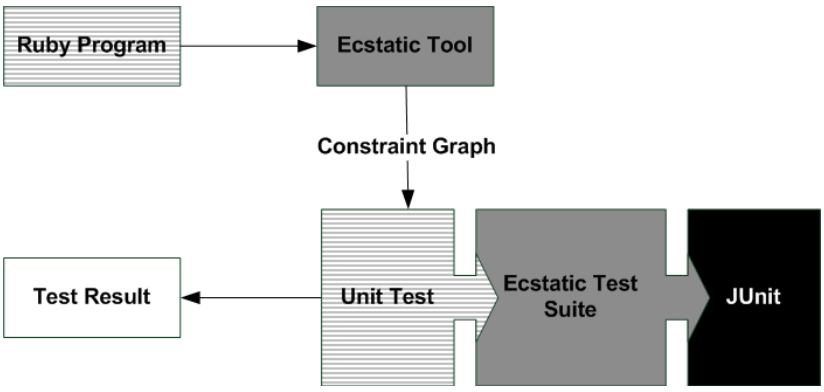


Figure 6.1: The process of executing a unit test. The shaded with horizontal lines gray boxes are supplied by the tester: a Ruby program and a unit test. The dark gray boxes are supplied by the tool: the Ecstatic tool and the test suite framework built on JUnit. The white box is the outcome of the test.

The assertions added are listed in Table 6.1. For each assertion a number of overloads exist, but they all include the ability to reference an identifier in the Ruby source code file. This is done on a line-and- column number basis, i.e. *"the identifier at line number 5, and column 7"*. The reason for this is based on the work flow of creating tests. First a Ruby program is written that utilizes some aspect of the language. Based on this a complimentary unit test is written. The unit test will have to reference points in the Ruby program to validate an aspect. The easiest way to do this is on an identifier basis, i.e. *"the identifier 'a' has this type"*. In order to unambiguously identify identifiers we use line and column numbers, because an identifier `a` might exist at several places in a Ruby program.

Assertion	Description
`typeSetIncludes()`	Asserts that the typeset includes the types supplied as an argument.
`typeSetExcludes()`	Asserts that the typeset excludes the types supplied as an argument.
`typeSetIsEmpty()`	Asserts that the typeset is empty.
`typeSetIs()`	Assets that the typeset is exactly the typeset supplied as an argument.

Table 6.1: The assertions added to JUnit to enable typeset testing. Each method takes a line and column number to identity the identifier in the Ruby source code file to test.

Figure 6.2 shows how the test suite is related to JUnit and the individual unit

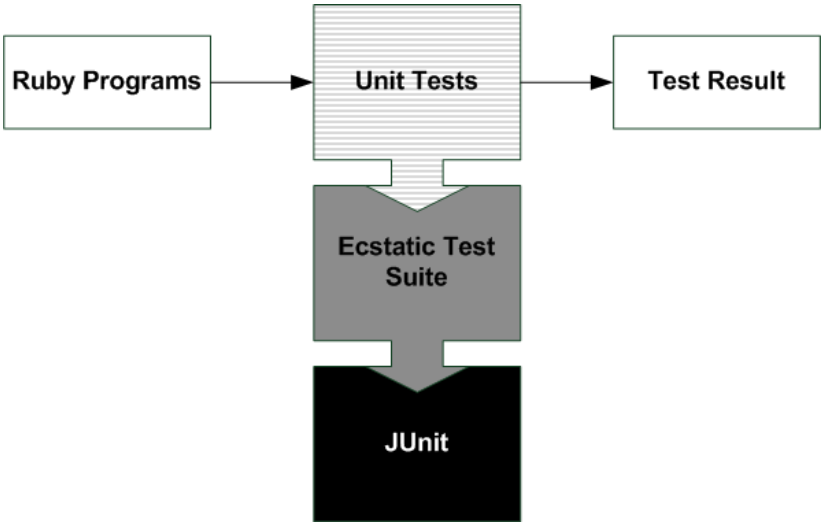


Figure 6.2: The relationship between the test suite, JUnit and the individual unit tests.

tests. A unit test takes a Ruby source file as input and outputs the test results.

Figure 6.3 shows a class diagram of the test suite. The `TypeInferenceTest` class executes the Ruby source file that includes code that should be tested. The `Assert` class contains the methods described above in Table 6.1, a reference to an instance of `TypeInferenceTest` used when executing the assertion methods. The `RubyFileTestBase` is the base class for all tests. It contains the static method `initTestForFile()`, which sets up the system for test by running `TypeInferenceTest`. It also includes instance methods to the assertion methods. Basically these delegate the call to the same method defined on the `Assert` class of which `RubyFileTestBase` holds a reference. This makes it easier for the test developer.
Creating a new test is as simple as subclassing `RubyFileTestBase`, and supplying a method (by convention called `setup()`), which calls `initTestForFile()` with the file path to the Ruby file under test. This method is decorated with the `@BeforeClass` JUnit annotation. This annotation ensures that this method will always be called before the individual test cases are run. Henceforth the developer creates individual test cases as per the usual JUnit way but utilizing the assertion methods defined in the `Assert` class.

An example unit test is seen in Appendix C. It is placed in the "instance variables"-group and it checks that attribute accessor methods work correctly.

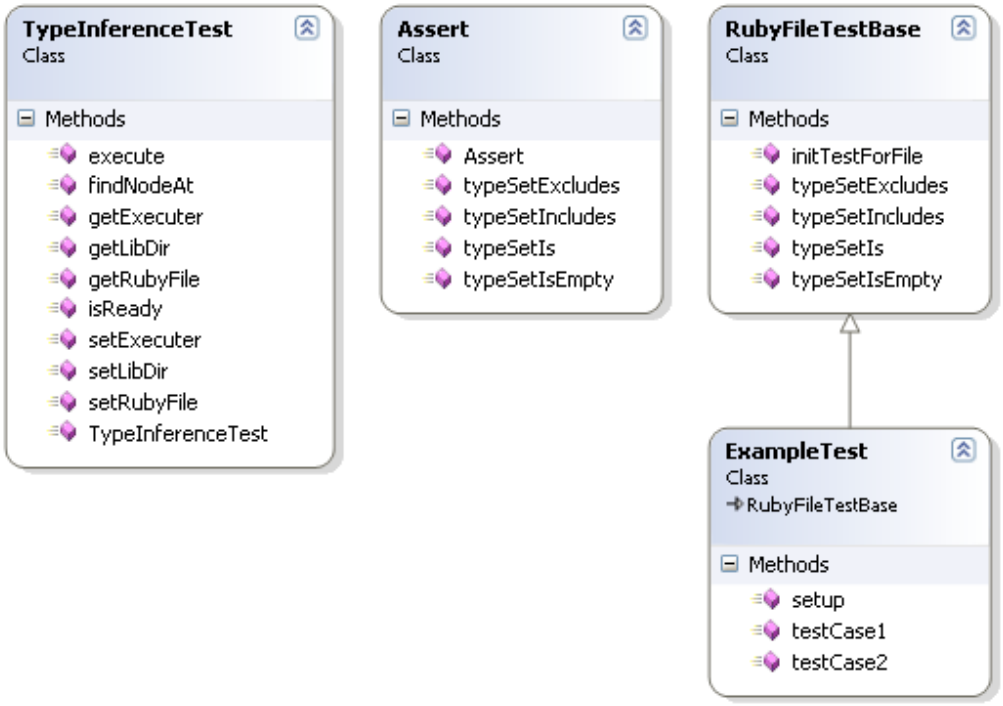


Figure 6.3: Diagram showing the individual parts of the test suite.

6.3.2 Results

Table 6.2 shows the results from running all the unit tests. It also gives an overview of the individual unit tests in each group. Each unit test has a number of singular tests inside. To increase readability these are omitted from the table, however, the full set of tests are available on the accompanying CD.

The following list reviews the tests in Table 6.2 that failed.

1. Recursion fails because it is not implemented (see Chapter 5, Section 5.5).

6.3.3 Reflection

Having a testing tool available has proved beneficial on many occasions. The issue discovered during SW9, which sparked the creation of the testing tool, has to some degree been remedied. During the development a test was created when a feature was found not working. This helped limit the possibility of re-introducing this defect later on. Furthermore, tests were created to support

Group	Test	Pass/Fail
Blocks		
	Blocks	✓
CPA		
	Template Propagation	✓
	Method Calling Method	✓
	Recursion	÷
Instance Variables		
	Attributes	✓
	Identifier Definition	✓
Ranges		
	Ranges Definition	✓
Samples		
	Example 1	✓
	Example 2	✓
	Module Definition	✓
	Simple Method	✓
	Arguments	✓
	Identity Function	✓
Total unit tests		13
Total tests		36
Passed		35
Failed		1

Table 6.2: Results from running the unit tests.

the development of a feature in a Test Driven way [67]. An example of this is the "Ranges"-tests. These were created before support for Ruby's Ranges was implemented. As such the tests helped guide the development, and served to validate when the feature was working correctly.
During development one person was responsible for creating tests. These tests were created from the documentation available on how Ruby works. This meant that tests were generally created with "no knowledge" of the developed tool. It is our belief that this helped increase the quality of the tests, because more intricate details were captured in tests. Contrast this with having the developers creating the tool also create the tests. In this scenario there is a risk of the developers creating tests that validate exactly what they have developed. Simply because they have knowledge of how it should work.

The following list reviews the issues presented in Section 6.3. Where appropriate we will present out opinion and suggest ideas for future work and improvements to the test suite.

Test Types: The tests created are primarily positive tests, i.e. does identifiers get the correct types. However, this is only one part of the required testing story. A desired extension to the tool is the ability to test for `NoMethodError`'s. This would enable testing of whether or not methods are defined, and hence enable both positive and negative tests of Ruby source code. Currently it is only possible to test for positive method calls, i.e. you can only test methods that are actually defined. Calling a non-defined method breaks a test. Implementing this assertion is analogous to the JUnit feature of checking that a specific exception is thrown. Hence, a better ability to create negative tests should be devised.

Classification: The classification used on the tests could have been better. A classification scheme more similar to that of Ada could be preferable. Our classification scheme could then be used as a sub categorization within these general classifications.

Small Tests: Focusing on creating small tests made it easier to produce a larger set of tests. It also helped limit the responsibility of each test. We were thereby able to locate and catch more focused defects.

Regression: The test suite and tests were used as a regression test tool. During implementation the tests were continuously executed to verify that everything still worked as supposed to. This proved very valuable and helped catch errors that might otherwise have crept in to the tool.

The current set of tests primarily exercise language and library features in smaller scale. However, there are few tests that deliberately and directly test aspects of CPA. Future work should remedy this to ensure a correct implementation of CPA and to alleviate that implemented features stop working.
Furthermore it would be desirable to increase the number of tests. 36 tests divided on 13 units is a low number. Especially for such a large and complex system.

6.4 Summary

This chapter investigated compiler validation and how that relates to type inference testing. The results formed the starting point for the testing effort performed on the Ecstatic tool. A framework for performing unit tests based on Ruby code was developed, and unit tests within different categories were developed. The results from running the tests are positive (only one test failed), but we need more tests to be more exhaustive.

In summary having a formalized testing process has proved very beneficial

through out the project's development.

CHAPTER 7

Experiments

This chapter examines Ecstatic by performing a number of experiments with it. To test Ecstatic 19 Ruby projects of varying complexity were chosen. The projects have all been found through the Ruby Application Archive (RAA) [45]. The experiments were performed to collect data for use in the next chapter. They will also help confirm or reject the hypotheses proposed in Chapter 4.

Table 7.1 lists the projects that Ecstatic have been tested with. The projects chosen are representative for the projects available through the RAA and in that respect represent real life programs.

The projects were selected on the premise that they do not depend on anything other than a standard Ruby environment (Ruby Core and the Standard Library). This requirement was imposed, because some projects in RAA depend on other programs and libraries that are not written in Ruby, and Ecstatic cannot handle that. The only notable exception to this requirement is Ruby Core, which is implemented in C.

The projects in RAA are somewhat limited in number. Most of the projects are best described as hobby projects. We found a limited amount of large scale Ruby programs in RAA. However, we still believe that running Ecstatic on the selected projects will yield valid information. This information will give an indication as to how large scale programs are composed.

Project Name	Description	LOC[a]
BMconverter	Convert bookmarks from different browsers of various formats	1207
Canna2skk	A translator program from a dictionary in Canna format to a dictionary in SKK format.	254
Clwiki	Simple Wiki written in Ruby	3386
Depends	A tool to determine the reverse dependencies of a Debian package	253
e	Extract Any Archive	122
FreeRIDE.rb	FreeRIDE is a Ruby Integrated Development Environment	2751
FTP_sync	Synchronize files on multiple machines via FTP server	492
ICalc	Simple IP Calculator for sub/sup networking	150
Markovnames	Given a list of names as input, produces new random variations in the same style	178
NaninHttpd	Simple HTTP daemon	491
Newsstats	Computes statistics for Usenet newsgroups	640
Qant	QAnt is a preprocessor of Ant build files	338
Quickey	Quickey is a little application for quick keyword entry	873
Roman	A test of a library that converts integers to roman numerals	104
ROOF	A simple object-oriented file system	1275
Setup	A generic installer for Ruby programs	1585
Slider	An application launch bar	618
Tiddy	Tiddy is a source code formatting and, or beautification software	682
Yawee	A Windows environment variable editor	126

Table 7.1: The projects used in the experiments with a short description and LOC.

[a]The Lines of Code (LOC) measure is based on the files in the project and excludes any external libraries. LOC includes blank lines and comments.

7.1 Data Collection

Ecstatic can provide statistics about the Ruby source code it is executed on. The data collected by the tool includes the following:

The Number of Files that the project is comprised of excluding external libraries.

The Number of Vertices on the constraint graph.

The Number of Edges on the constraint graph.

Number of Classes in the project.

Number of Methods in the project.

Vertices with an Empty Type Set on the constraint graph. The empty vertices of the templates are not counted because there are nodes on a template that are supposed to be empty. An example of this are the vertices that represent the formal arguments of a method. The empty vertices on the constraint graph can be used as a measure for the precision of the inferred types. However, empty vertices can also indicate that propagation on the constraint graph for some reason stops prematurely.

Methods: We collect the following list of information on methods.

Class and Method Name The method's class name and name.

The Number of Vertices on the method.

The Number of Edges on the method.

Number of Clones of the method. A methods number of clones can be used as a measure of how polymorphic message sends are. This is possible because a clone corresponds to a message send with monomorphic arguments. See Section 5.2 for a detailed discussion of the relationship between methods and clones.

Size of Type Sets the size of the vertices type set's. The type sets sizes can be used as a measure of how polymorphic Ruby programs are. The more polymorphism they exhibit the greater the size of the type sets. We have chosen a cutoff value of 5 for the size of the type sets. With more than 5 different types stored in a variable its use becomes increasingly confusing.

Running Time Ecstatic's total running time in seconds on the project.

Error Condition if the tool exits prematurely.

The individual output from running Ecstatic on each of the projects is listed in Appendix D.

7.2 Results

The projects have been divided into two different categories depending on the result of running Ecstatic on them.

Successful: Contains the projects that have types successfully inferred. The projects in this group are: BMConverter, Canna2skk, e, FreeRIDE, ICalc, Markovnames, Quickey, and Roman. The remainder of this chapter will only deal with projects in this category.

Fail: Contains the projects where the tool exits before any type information is obtained. Category 2 can be further subdivided into:

Fail.Tool: The error condition originates from the type inference tool. The projects of sub-category Fail.Tool is shown in Table 7.2 This sub-category represents shortcomings in the tool itself.

We further divide Fail.Tool into:

Fail.Tool.NotImplemented The projects in this category encountered a language construct that was not implemented.
CLWiki, FTP_Sync, Newsstats, QAnt, Setup, Tiddy, and Yawee belong in this category. They failed, because of an implicit conversion from a block to a `Proc` object. This use of blocks is not implemented in Ecstatic.

Fail.Tool.LogicError Projects in this category exit because of an unknown logical error in the tool. The reason for this exception needs to be examined further before a precise explanation for the error condition can be given.
The projects in this category are: Depends, ROOF, and Slider.

Fail.Subsystem: Projects with errors that originate from a subsystem of Ecstatic. For example errors that originate from JRuby.
Of the tested projects only a single one exhibited this error condition: NaninHttpd where JRuby throws a syntax exception.

7.2.1 Use of Variables

We collected the sizes of type sets of the vertices that are used to hold data. This means vertices that represent message sends are not counted.

It is reasonable to use the size of the type set as a measure for the degree of data polymorphism of a program. This is is illustrated in Listing 7.1.

Project Name	Error
CLWiki	RunTimeException -> Method not understood (ProcBlock)
Depends	NullPointerException -> null
FTP_Sync	RunTimeException -> Method not understood (ProcBlock)
Newsstats	RunTimeException -> Method not understood (ProcBlock)
QAnt	RunTimeException -> Method not understood (ProcBlock)
ROOF	ClassCastException -> Master cannot be cast to ConstraintGraph
Setup	RunTimeException -> Method not understood (ProcBlock)
Slider	ClassCastException -> Colon2Node cannot be cast to ConstNode
Tiddy	RunTimeException -> Method not understood (ProcBlock)
Yawee	RunTimeException -> Method not understood (ProcBlock)

Table 7.2: The projects of category Fail.Tool

```
1 a = 1
2 b = 1.0
3 b = a
4 c = "string"
5 c = b
```

Listing 7.1: Data polymorphic Ruby example

The sizes of the type sets of the variables are as follows: The variable `a` has a type set consisting of a single type `Integer`. Variable `b` has a type set of size two as it is given both a `Float` directly and the types that the variable `a` contains. Finally the variable `c`'s type set has size three as it is given both the types from the type set of variable `b` and the type `String`. We see that as the data polymorphic use of a variable increases the size of the type set does too.

The degree of data polymorphism for each project is shown in Table 7.3. A vertex may have an empty type set if the following conditions are met. Suppose we have an assignment of the form `variable = expression`. If the expression contains a message send, and this message send for some reason fails to have a type set inferred for it, both the expression and the variable will have empty type sets. Another probable cause of empty type sets is that propagation fails or

stops prematurely.

Project Name	**0**	**1**	**2**	**3**	4	**>5**
BMconverter	528	10597	1853	48	0	0
Canna2kk	71	524	30	15	0	0
e	9	189	6	30	0	0
FreeRIDE	191	645	25	4	0	0
ICalc	54	182	15	8	1	0
Markovnames	87	236	8	0	0	1
Quickey	23	39	0	0	0	0
Roman	32	582	12	0	0	0

Table 7.3: Data showing the distribution of the sizes of the typesets of the projects of the Successful category.

Table 7.4 lists the distribution of the sizes of the type sets of the projects vertices. It is worth noting that only a single project (e) uses data polymorphism to a greater extend.

7.2.2 Use of Instance Variables

Only three projects from the successful category uses instance variables. The three are: BMConverter, FreeRIDE, and Markovnames. Table 7.5 shows the number of instance variables in the three projects. Only BMConverter uses instance variables in a polymorphic way. All three instance variables of BMConverter has a type set of size two.

Project Name	$<= \mathbf{2}$	$> \mathbf{2}$
BMConverter	99.63%	0.37%
Canna2skk	97.66%	2.34%
e	87.18%	12.82%
FreeRIDE	99.54%	0.46%
ICalc	96.54%	3.46%
Markovnames	99.70%	0.3%
Quickey	100%	-
Roman	100%	-

Table 7.4: The projects percentages of vertices on the constraint graph that have a type set below and over size 2.

Project Name	Number of Instance Variables	Number of Instance Variable used Polymorphic
BMConverter	30	3
FreeRIDE	9	0
Markovnames	6	0

Table 7.5: The distribution of instance variables in the projects.

7.2.3 Degree of Parametric Polymorphism

Information about parametric polymorphism is collected by counting the number of clones for each method. A clone for a method corresponds to a monomorphic application of the method's arguments. This makes it a reasonable measure for the parametric polymorphism of message sends. If a method only has a single clone the method is not used in a polymorphic way. Table 7.6 shows the distribution between the number of methods and how many of them exhibit parametric polymorphism.

Project Name	Number of Methods	Number of Methods used Polymorphic
BMconverter	54	5
Canna2kk	2	0
e	0	0
FreeRIDE	10	0
ICalc	4	0
Markovnames	7	0
Quickey	1	0
Roman	5	1

Table 7.6: Data that shows the distribution of the number of clones per method.

7.2.4 Running Time

The running time of executing Ecstatic on each project is listed in Table 7.7.

7.2.5 `NoMethodErrors`

Further investigation of the `NoMethodError`s that occurred showed that they were caused by a missing type in a variables type set. The primary reason for the missing type is a lack of special implementation for methods in Ruby Core. For a discussion of this see Section 5.1.2.

Project Name	Average Running Time in Seconds
BMconverter	7.48
Canna2kk	1.62
e	1.57
FreeRIDE	1.68
ICalc	1.50
Markovnames	1.50
Quickey	1.37
Roman	1.48

Table 7.7: Running time of executing Ecstatic on each project. It is listed as an average of five consecutive runs.

The following projects contain `NoMethodError`s: BMConverter, Cann2skk, freeRIDE, and ICalc.
The errors all follow the same pattern, so we will only discuss it in the context of one project.

BMConverter: Has a `NoMethodError` in one of its required files `parsearg.rb`. An `Integer` type is missing from a typeset of a variable. This is due to the fact that a special implementation of the method `String.split` is missing.

7.3 Data Critique

This section performs a critique of the projects we have used in our experiments.

As mentioned the projects used in the experiments are characterized by being hobby projects. They are not real life and large applications per se. Furthermore, in some ways they illustrate the duality between Ruby as a scripting and object-oriented language. The projects uses instance variables sparingly if at all. This is an indication of the hobby nature and of using Ruby as a scripting language. Furthermore, the number of methods is quite small for all projects.
Although both applications and uses are perfectly valid use of Ruby, it is not necessarily an indication of large scale applications.

The projects were chosen from RAA. Their inclusion in the experiments were based on the reasons presented in the beginning of this chapter. We have not been able to find projects that were both large in size and did not utilize external libraries, which Ecstatic is unable to handle. We would therefore recommend that future work either find larger projects or remedy the issues involved in having Ecstatic run on them. This might entail the addition of capabilities for

handling external libraries.

7.4 Summary

This chapter described a number of experiments performed using Ecstatic. A number of projects were selected from the Ruby Application Archive (RAA) and a set of metrics were collected from running Ecstatic on each. The data collected will form the basis of a discussion in the next chapter.

CHAPTER 8

Discussion

Through the course of this thesis we have investigated and presented a number of themes. This shaped the definition of hypotheses and goals for the thesis in Chapter 4. Furthermore, we have discovered a number of issues that we wish to discuss further. This chapter discusses possible answers to the hypotheses, goals, and problems presented in Chapter 4.
In essence, this chapter contains our reflection on this thesis's work.

8.1 Experiences With the Ruby Language

During the course of this thesis we have gained a deeper understanding of the Ruby programming language. This section discusses our experiences gained with respect to Ruby as a language.

Ruby Community [46] describes Ruby as, *"A dynamic, open source programming language with a focus on simplicity and productivity. It has an elegant syntax that is natural to read and easy to write."*. On the surface Ruby is a very elegant, concise and natural language. This perception is conveyed initially in Chapter 2. However, when you dig into the language and approach the corners of it, the image becomes a bit blurry. In fact Yukihiro Matsumoto – the author of – says it concisely, *"Ruby is simple in appearance, but is very complex inside, just like our human body"* [29]. Ruby does truly look good on the surface, but underneath it is complex and conflicting.

We believe part of the issue is the lack of a specification for the language. Ruby is effectively what CRuby (the original interpreter written in C) accepts. There is no official written syntax or semantics. As mentioned in Chapter 6, there is an interest in the Ruby Community and from Yukihiro Matsumoto himself in standardizing the language, in being more transparent in what the language contains [62, 26]. However, the current situation continues to remain: what CRuby accepts, Ruby is.
The main problem with a lack of language specification is that issues with the language does not present themselves until they are implemented and in use. Conflicting semantics are not discovered until someone uses the language and finds the inconsistency. With an official specification and an official process for

changing or adding to the language, issues can be discovered and remedied before implementation. A prime example of these issues are blocks or closures. Cantrell [9] has a lengthy discussion by example on how blocks work in Ruby. From this discussion it is evident that blocks are far from simple and orthogonal in the language. Section 2.1.6 presents one such difference, namely between `Proc.new` and `lambda`, how they deal with `return` and whether their defining context needs to exist when called. Section 2.2.2 presented an issue with dynamic programming, namely that it appears to not be a first class citizen of the language. Even though Ruby allows programmers do almost anything at runtime (because any Ruby code can be evaluated dynamically), the evaluated code is not treated in the same way as code written directly in Ruby before runtime. Yegge [70] states that calls to injected methods are atomic, i.e. their inside workings are not exposed. Furthermore, the injected code is evaluated and then forgotten in the sense that the result of the evaluation is saved, but the actual code is not. So you cannot at a later time retrieve it, work with it and test and debug it.

Reading the above discussion, you might reach the conclusion that Ruby is complex, illogical, and inconsistent. However, in spite of its quirks it works. Programmers use it and love it. In fact they often become Ruby advocates telling everyone how delightful Ruby is. Martin Fowler, employed at Thoughtworks, announced at RailsConf 2007 that *"40% of our new business this year in the US is Ruby work"* [23]. O 'Reilly – a publisher of technical books – publishes a book sales list each quarter. Their statistics state that Ruby books outsell those of Python and Perl; Javascript and PHP outsell Ruby.
Both O 'Reilly's sales statistics and Fowler [23] indicate the popularity of Ruby. We do not have conclusive evidence as to why Ruby programmers love their language despite its quirks. Maybe even with the quirks programmers still feel more productive and comfortable compared to other languages.

8.1.1 Implementing Blocks

Blocks are an integral part of programming Ruby. They are used in a plethora of ways through out the Core and Standard Libraries, and their use is indeed considered a Ruby idiom. We described blocks briefly in Chapter 2. In the following we will discuss issues with blocks from an implementation point of view.

There are four types of blocks and closures in Ruby, and they can be expressed in seven ways [9]. The semantic difference between them are substantial. We already described how `Proc.new` and `lambda` differ with respect to return and existence of their defining context (Section 2.1.6). Furthermore, the semantics of a block construct in Ruby changes between version 1.8 and version 1.9, only

adding to the complexity[1].

Ruby blocks differ from Smalltalk and Self counterparts by their complexity. In Smalltalk and Self there are one block syntax and one block semantics. In contrast Ruby has seven block syntaxes and four block semantics. Obviously, the more complex the problem, the more implementation effort is required. Furthermore, this adds to the complexity of Ecstatic.

Ecstatic supports blocks passed via `{` and `}` with one parameter, and the use of the `yield` keyword. Two examples are presented in Section 2.1.6 in Listing 2.8 and Listing 2.9. This use of blocks is similar to method calls, and indeed this is how it is implemented. However, this practice does not readily apply itself to closures. Future work needs to remedy this, and treat closures correctly.

8.1.2 Dynamic Programming

In Chapter 4, we proposed an issue that we thought would pose a challenge for the tool, namely the concept of dynamic programming. Ecstatic does not support dynamic programming. Simply we do not handle it; they are effectively No Operation (NOP). In the following, we will discuss the issues involved in supporting dynamic programming. This discussion is based on the experiences gained during research of Ruby and implementing the tool.
The problem with dynamic programming is that it performs runtime modifications on the environment. Examples of this are the addition of methods to classes, or evaluating the contents of a string as valid and parseable Ruby code. In the following, we will restrict our discussion of dynamic programming to that of dynamically adding methods. There are three issues involved in supporting dynamic method programming: flow insensitivy, parsing a string, and adding methods to a context. We will examine each of these issues in detail in the following.

Flow Insensitivity: In Ruby, methods can be added at any place and time in the code. However, CPA works in a flow insensitive way, and hence would not be able to accommodate this. The issue deals with the point of definition and application. Before the method is added, calls to it should result in a `NoMethodError`. After it is added calls should of course work and no `NoMethodError` should occur . However, the distinction between when the call is made and how this is related to the point of method addition is not available when using a flow insensitive algorithm.

Parsing a String: `class_eval`, `module_eval`, and `instance_eval` differs from

[1] In v. 1.8 `proc` refers to `lambda`, in v. 1.9 it refers to `Proc.new` [9].

`eval` in one way: they support being called with a block instead of with a string. Because a block is readily available for parsing, we will initially refrain from including this in the discussion. We will refer to `class_eval`, `module_eval`, and `instance_eval` as context-eval's. Looking at the four `eval` methods that takes a string, and remembering the example from Section 2.1.5 in Listing 2.7, we see the difficulty: the new code is supplied in a string. First, this string may be difficult to parse and retrieve code from. Second, it may use concatenation to dynamically create the code. A sample use for this would be to decide at runtime what the name of the method should be as is the case in Listing 2.7. Therefore, extracting the code supplied to an `eval` method is potentially difficult.

If we turn our attention to the context-eval methods that take a block as parameter, the issue becomes simpler. Ultimately, because the block is readily parseable and therefore can be used directly in type inference.

Adding a Method to a Context: Adding a method to a class or an object is an integral part of a system that analyses or executes Ruby code. As described in Section 2.1, class definitions in Ruby are executed, and at any time you can open a definition and add methods to it. This is for example how singleton methods work. Therefore, an application analyzing Ruby code must support adding new methods to classes, modules, and instances to be of any use. Exemplified by singleton methods Ecstatic already supports adding methods to classes, modules, and instances. However, the code adding the method must be directly parseable and part of the code at analysis time. An example of this is seen in Listing 2.3. Ultimately, the issue of adding a method to a context is a minor one. This depends, however, on the method being readily available to add, i.e. not be in the form of a string.

Reviewing the issues presented above we see that the biggest issues in supporting dynamic addition of methods is: flow insensitivity and parsing a string. Adding a method to a class or instance is already supported by the Ecstatic. Parsing a string and extracting the information in a read and usable form is difficult. Overcoming the issue with flow insensitivity remains unclear.

Adding methods is not the only use of the `eval` methods. They can be used to evaluate a variable in another context, and thus be used to return instance variables. Still, the same considerations and issues exist as presented above. Especially the lack of ability to parse a string is paramount, because it severely limits what can be extracted from an `eval` call.

Salib [48] in the context of Python also have problems with handling dynamic code. His type inferencer does not handle it at all. He analysed the standard library of Python and found that `eval` constructs were used sparingly and its

use could often be rewritten with no loss of functionality.
We have not investigated how often `eval` is used in Ruby.

8.2 Discussion of the Hypotheses

This section will follow up on the hypotheses, goals, challenge, and requirements from Chapter 4, and the experiments in Chapter 7.

Hypothesis 1

The first hypothesis is the assumption that Ruby programmers do not make type errors.

In order to confirm or reject this hypothesis, we would need a type checking tool for Ruby that is 100% reliable. Furthermore, it requires running experiments on a vast number of real life Ruby projects of varying sizes. Because of the immature state of the Ecstatic tool and because of the nature of the experiments made, we cannot conclude definitely on this topic. However, based on the experience we have gained while working on type inference in Ruby and based on the experiments we have made, it seems that type errors in Ruby programs are few or none existing. The `NoMethodError`s reported by Ecstatic during the experiments are caused by the tool's incomplete state rather than the quality of the experiment projects. If the Ecstatic tool is completed and more experiments are conducted, we believe that this hypothesis is more likely to be confirmed than to be rejected.

This hypothesis can also be viewed as a mission statement for creating a tool like Ecstatic. The purpose of Ecstatic could be to help Ruby programmers avoid making type errors in the future by using the tool. Programmers could use the tool to check their program for errors before release or deployment.

Hypothesis 2

Hypothesis 2 examines if the presence of a type inference tool can help drive industry into accepting Ruby.

For a tool to help in this direction, it would be required to fully support all parts of the Ruby language, including the advanced and the inconsistent elements discussed earlier in this chapter (dynamic programming and blocks). And even

with the presence of a complete tool, only time would tell how the industry will respond. Since Ecstatic is not complete, we have not come closer to confirming this hypothesis. However, the experience we have gained working with Ruby tells us that the language would indeed benefit from tool support. Because Ruby has no language specification apart from the CRuby implementation, any additional support to help understand the language and the programs written in the language would be appreciated.

Hypothesis 3

Hypothesis 3 explore if Ruby programmers are more productive than programmers using other languages.

This is an aspect of Ruby that we have not looked into, and we leave it open for further research.

Hypothesis 4

This hypothesis is concerned with how polymorphic Ruby programmers write their programs. Our hypothesis is that polymorphism is only used very restrictively both in terms of data polymorphism and parametric polymorphism.

Hypothesis 4.1

This sub-hypothesis explores data polymorphism in Ruby programs. Our assumption is that even though data polymorphism is possible in Ruby programs, programmers restrict how polymorphic they write their programs.

In Section 7.2.1 and Section 7.2.2, we have examined the use of data polymorphism in Ruby programs.

Variables: We define that a variable is used sparingly with regards to polymorphism if the size of the variable's typeset do not exceed a size of two. This size is chosen from the subjective viewpoint that if variables are used to store more than two kind of values it limits the comprehensibility of the program.

Table 7.4 lists the distribution of type set sizes of the constraint graph. Our findings are that of all the examined projects, only the project `e` has a considerably high use of polymorphic variables.

Our assumption about the use of data polymorphism is thereby confirmed for ordinary variables in Ruby Programs. Next we will examine the usage of instance variables.

Instance Variables Three of the examined projects uses instance variables, the three are: BMConverter, FreeRIDE, and Markovnames. Of these, only the project BMConverter uses polymorphic instance variables. The BMConverter project defines 30 instance variable of which only 3 are used polymorphic. As it is the case with ordinary variables, it also the case that instance variables are only used in a polymorphic manner very sparingly. In Section 5.4 we made the assumption, that instance variables are rarely used polymorphically. This assumption has now been confirmed, which indicates that treating instance variables the same way as class variables only slighty lowers the precision of type inference.

Hypothesis 4.2

In Section 7.2.3, we examined the use of parametric polymorphism in Ruby programs. Of the 83 methods in the projects, only 7% have more than a single clone. A clone represents one monomorphic use of a method. Having more than one clone therefore means, that the method is used polymorphically. The method `SetExpression` from the BMConverter project is the method with the highest polymorphic use; it has 12 clones in total.

As a side-note, it is interesting to note that the methods used polymorphically are of modest complexity, i.e. they have a low number of vertices. The `setExpression` of the BMConverter only has 24 vertices and 29 edges. All methods that have more than a single clone are of very limited complexity.

Hypothesis 5

As it is explored in this thesis and with the development of Ecstatic, it is possible to use CPA for the Ruby Programming language. There are differences between the Self and Ruby as explored in Chapter 2, but these differences have not hindered the retrofitting of CPA to Ruby. Ruby presents a few new challenges, that Agesen [1] does not deal with. These include classes, modules and mixins, optional arguments and array arguments on methods, dynamic programming, and several semantics for blocks. The implementation of Ecstatic proves, that most of these challenges can also be handled by CPA. The only challenges open for future work is the implementation of blocks and dynamic programming.

8.3 Experiences Gained

In the second part of Chapter 4 we listed a series of goals, challenges, requirements, quality requirements, and usage scenarios for the developed tool and the thesis as a whole.

The purpose of these secondary objectives was to help put a perspective on the hypotheses and the thesis. The idea was that these objectives would help convey a better understanding of the premises of the thesis. In the following sections we will discuss if the secondary objectives were accomplished or not.

8.3.1 Goals

One of the goals for the project was to explore if it would be possible to take a type inference approach developed in Self and use it for Ruby. This have been positively accomplished. We have shown that it is possible to retrofit CPA to work in a Ruby context and make it work with reasonable success. This is further addressed in Section 8.2. However, there are still some quirks that needs ironing out. A problem that needs addressing is the vertices with empty typesets. An empty typeset of a vertex may as stated in Chapter 7 be the result of the propagation not proceeding as it should. Eliminating vertices with empty typesets would further strengthen our belief in the tool.

One of the concerns with adopting CPA was the differences between its original language Self and Ruby. As there are differences these would have to be addressed. We wanted to be as true to the original specification as possible, but recognized that some retrofitting would have to take place. If we have deviated from the original specification we have been explicit about it. One of these areas is in the handling of methods and templates. See Section 5.2 for an in-depth discussion about this.

We have succeeded with the goal of extracting metrics from type inferred programs. These metrics include how variables and methods are used. The collected metrics have been used in confirming or rejecting the hypotheses of Chapter 4.

8.3.2 Challenges

In the work with CPA we identified a set of challenges that we believed to be a concern in adapting CPA into a Ruby context. The challenges were conditional control flow, continuations, and dynamic programming.

Conditional control flow did not prove to be a challenge. Our concern was that as Self does not have traditional control structures this could present a problem in relation to Ruby. Even though Self does not have conditional control structures in a traditional sense, it has structures that are comparable to them. The solution was not to try and find out which branch should be evaluated. But rather evaluate all branches. Evaluating all the branches of a conditional may introduce imprecision in the inferred types. The problem is that a conditional has branches, which, when executed, is never evaluated. However, types will still be inferred for this branch. This means that types that are never present in the running program will be included in the inferred types.

Handling continuations have proved to be very difficult. The biggest problem is that Ecstatic needs to be aware of how the call stack was when the continuation was created. Ecstatic has no understanding of this and therefore continuations are not implemented. One possibility that could be explored in making Ecstatic aware of the call stack is to take a "snapshot" of `RubySim` when a continuation is created. When a `call` messages is sent to the continuation it must switch the current `RubySim` with the stored "snapshot".

Perhaps the biggest problem to overcome in providing a full type inference tool for the Ruby language is also one of the features that make Ruby an attractive language. This feature is Dynamic programming, which enables the programmer to add methods to classes or modules as a program is running. Ecstatic does not handle dynamic programming in any way and it must be considered an open question as to how this should be handled in the context of CPA. We have, however, discussed this issue in Section 8.1.2.

8.3.3 Requirements

One of the requirements for the developed tool was that it should run, not only on toy examples, but on real life programs. We have fulfilled this requirement as all the projects in consideration in Chapter 7 are real life programs. But as it is noted in Chapter 7 they are of limited complexity.

We have chosen not to fulfill the requirement that the developed tool should be robust. This deviation was performed, because as we consider the tool a work in progress and the lack of robustness helps us identify the areas that needs addressing to complete the tool.
The AST nodes that are currently not handled in the Controller, (see Chapter 5) throws an exception. It is, however, a relatively simple matter to make the tool more robust: instead of throwing an exception we could make an empty vertex and just add it to the constraint graph.

The types that the tool infers is as precise as we hoped they would be. We see from the experiments presented in Section 7.2.1 that the type sets of variables that represent data have a limited number of types. For most of the projects that have types inferred over 90% have at most two types in their typeset.

8.3.4 Quality Requirements

One of the quality requirements is robustness. As explained above we have chosen to deviate from this.

Section 7.2.4 lists the running times of a set of real life Ruby programs. The requirement placed on the running time was that it should be in seconds. This goal was achieved. There is of course a correlation between the running time and both the size and complexity of the program that the tool is inferring types for. The larger the program the more vertices need to be handled, which results in a longer running time. For ordinary Ruby programs we experience an asymptotic running time in the number of vertices times the number of edges on the constraint graph. For complex Ruby program the asymptotic running time approaches the number of vertices squared.

Efficiency is a quality requirement that we have not addressed. We assume that we have as much space available as is needed. For the time being no constraints have been put on the program neither time nor space wise.

The correctness quality requirement of the developed program is sub divided into a precision and an accuracy requirement. The precision of the inferred types is measured in the size of the typesets. The precision of the inferred types is discussed under Section 8.3.3. We had the requirement for the tool concerning the accuracy of the inferred types, that they should be as accurate as possible, i.e. having only a single type in a typeset is more accurate that having more.

8.3.5 Usage Scenarios

With a developed type inference tool in hand we have explored a number of areas of application for which the inferred types could be used.

One area of use for the inferred types is in a type checker that will examine the inferred types for say a variable and check it for consistent use throughout a program. The developed tool has some limited capabilities in this area and support some type checks. The type checks are limited to reporting about `NoMethodError`s.

Different approaches have been discussed as to how the inferred types for a program could be presented to a user of the tool. One of the things that was discussed early on was integrating the type inference tool with an Integrated Development Environment (IDE). This was considered but rejected at the time, since we wanted the focus of the project to be on the development of the tool and not on integrating with an IDE. Integrating in an IDE may be an important step in helping Ruby being adopted by industry.

A relation to the discussion about the integration with an IDE was how the inferred types for a program could be presented. Two different approaches was discussed: one approach was to annotate the types directly in the source code, presenting the user of a tool with a view that is similar in appearance to that of an explicitly typed languages. The other approach was to annotate the types in a structure that resembles the syntactic structure of the program. The approach that we have chosen for the tool is to annotate it directly in the source code. We have chosen this approaches as we believe that this is a view potential users of the program will be more familiar with.

8.4 Summary

This chapter discussed Ruby, its lack of an official specification and what this implies for language implementors, the problem with dynamic programming and preliminary issues to investigate, and the problems with Ruby blocks. We follow up on the hypotheses presented in Chapter 4 and present the current status of obtaining an answer for them. Finally, we discuss experiences gained and future usage scenarios for Ecstatic.

CHAPTER 9

Conclusion

To examine the field of type inference in relation to Ruby, we have developed a static code analysis tool called Ecstatic. It employs the Cartesian Product Algorithm (CPA) originally developed for use in the Self language. By comparison between Ruby and Self we conclude that they are sufficiently similar on a semantic level to warrant employing CPA on Ruby.

To motivate this thesis we presented five hypotheses, which we wanted to investigate by conducting experiments on Ruby programs.
The Ecstatic tool has enabled us to conduct experiments on several arbitrary Ruby programs found at the Ruby Application Archive (RAA). The purpose was to gain insights into how programmers use Ruby. The immaturity of Ecstatic coupled with a low complexity in the tested programs means, that we did not yield a definitive answer to the proposed hypotheses. However, through the experiments we gained more knowledge on two of the hypotheses. Our findings indicate a low degree of polymorphism being used in Ruby programs.

We end this conclusion by listing the major and minor contributions of this master's thesis.
The major contributions of this thesis are:

Ecstatic: We have implemented a tool that performs type inference on Ruby programs using the Cartesian Product Algorithm (CPA).

CPA Works on Ruby: We have implemented CPA to work on Ruby programs. The CPA was developed for use on the Self programming language, so it was not immediately apparent that it would work on Ruby.

Experiences in Implementing CPA: We have gained a number of insights and considerable experience in implementing CPA on Ruby. Some of these details and concepts are not readily available in Agesen [1]'s work on CPA.

Foreign Code Inclusion: We present a method for extracting type information from the Ruby Core's RDoc. This enables us to perform more precise type inference, because we are aware of the types of the built-in libraries.

Experiments on Ruby Code: With Ecstatic we have conducted a number of experiments on Ruby programs found at the RAA. This enables us to collect a set of statistics on how Ruby programs are written, including statistics

on data and parametric polymorphism.

Testing a TI System: Based on compiler validation, we present a method for testing the type inference system. The tests are based on Ruby source code samples and unit tests based on an extension to JUnit.

Comparing Ruby, Self, and Smalltalk: We perform a language comparison between Ruby, Self, and Smalltalk. We conclude that the three languages are very similar on a semantic level. Although the similarities between Ruby, Self, and Smalltalk are often presented, a more thorough comparison has not been done before.

The minor contributions of this thesis are:

Ideas for Future Research: We present four hypotheses regarding Ruby programs and programmers. Three of these are considerably broad, and are hence not confirmed or rejected in this thesis. However, they can be considered ideas for future work and research.

Types in Ruby: We present a suggestion on how to understand types in Ruby. There are different views on this, and we compare these views from a type inference angle.

Bibliography

[1] Ole Agesen. Concrete type inference: Delivering object-oriented applications. Technical report, Stanford University, Mountain View, CA, USA, January 1996. http://research.sun.com/techrep/1996/smli_tr-96-52.pdf.

[2] Ole Agesen and Urs Hölzle. Type feedback vs. concrete type inference: a comparison of optimization techniques for object-oriented languages. In *OOPSLA '95: Proceedings of the tenth annual conference on Object-oriented programming systems, languages, and applications*, pages 91–107, New York, NY, USA, 1995. ACM Press. ISBN 0-89791-703-0. doi: http://doi.acm.org/10.1145/217838.217847.

[3] Ole Agesen, Jens Palsberg, and Michael I. Schwartzbach. Type inference of self. In *ECOOP '93: Proceedings of the 7th European Conference on Object-Oriented Programming*, pages 247–267, London, UK, 1993. Springer-Verlag. ISBN 3-540-57120-5.

[4] Ole Agesen, Lars Bak, Craig Chambers, Bay-Wei Chang, Urs Hölzle, John Maloney, Randall B. Smith, David Ungar, and Mario Wolczko. *The Self 4.1 Programmer's Reference Manual*. Sun Microsystems, Inc., 901 San Antonio Road, Palo Alto, CA 94303 USA, May 2003. http://research.sun.com/self/release_4.0/Self-4.0/manuals/Self-4.1-Pgmers-Ref.pdf.

[5] Joe Armstrong. The development of erlang. In *ICFP '97: Proceedings of the second ACM SIGPLAN international conference on Functional programming*, pages 196–203, New York, NY, USA, 1997. ACM Press. ISBN 0-89791-918-1. doi: http://doi.acm.org/10.1145/258948.258967.

[6] Holger Baer. Analysej (program analysis collection). http://sourceforge.net/projects/programanalysis.

[7] Alan H. Borning and Daniel H. H. Ingalls. A type declaration and inference system for smalltalk. In *POPL '82: Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 133–141, New York, NY, USA, 1982. ACM Press. ISBN 0-89791-065-6. doi: http://doi.acm.org/10.1145/582153.582168.

[8] Brett Cannon. Localized type inference of atomic types in python. Master's thesis, California Polytechnic State University, June 2005. http://www.ocf.berkeley.edu/~bac/thesis.pdf.

[9] Paul Cantrell. Closures in ruby. http://innig.net/software/ruby/closures-in-ruby.rb.

[10] Luca Cardelli. Basic polymorphic typechecking. *Sci. Comput. Program.*, 8(2): 147–172, 1987. ISSN 0167-6423. doi: http://dx.doi.org/10.1016/0167-6423(87)90019-0.

[11] Luca Cardelli. Type systems. *ACM Comput. Surv.*, 28(1):263–264, 1996. ISSN 0360-0300. doi: http://doi.acm.org/10.1145/234313.234418. URL http://portal.acm.org/ft_gateway.cfm?id=234418&type=pdf&coll=ACM&dl=ACM&CFID=2566757&CFTOKEN=27972208.

[12] Craig Chambers and David Ungar. Interactive type analysis and extended message splitting; optimizing dynamically-typed object-oriented programs. In *PLDI '90: Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation*, pages 150–164, New York, NY, USA, 1990. ACM Press. ISBN 0-89791-364-7. doi: http://doi.acm.org/10.1145/93542.93562.

[13] Craig Chambers, David Ungar, Bay-Wei Chang, and Urs Hölzle. Parents are shared parts of objects: inheritance and encapsulation in self. *Lisp Symb. Comput.*, 4(3):207–222, 1991. ISSN 0892-4635. doi: http://dx.doi.org/10.1007/BF01806106.

[14] Codehaus. Jruby - java powered ruby implementation. URL http://jruby.codehaus.org. http://jruby.codehaus.org, Last seen December 2006.

[15] Mats Cronqvist. Troubleshooting a large erlang system. In *ERLANG '04: Proceedings of the 2004 ACM SIGPLAN workshop on Erlang*, pages 11–15, New York, NY, USA, 2004. ACM Press. ISBN 1-58113-918-7. doi: http://doi.acm.org/10.1145/1022471.1022474.

[16] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *POPL '82: Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 207–212, New York, NY, USA, 1982. ACM Press. ISBN 0-89791-065-6. doi: http://doi.acm.org/10.1145/582153.582176.

[17] Rick DeNatale. Chain, chain, chain. Blog, June 2007. http://talklikeaduck.denhaven2.com/articles/2007/06/02/chain-chain-chain.

[18] Rick DeNatale. comp.lang.ruby: so lost on singleton, metaclass and virtual class. Google Groups, June 2007. http://groups.google.com/group/comp.lang.ruby/msg/ff99dccab14b2dd9.

[19] Pat Eyler. Jruby, what's in it for us. Blog, September 2006. http://on-ruby.blogspot.com/2006/09/jruby-whats-in-it-for-us.html.

[20] Ruby FAQ. How does ruby compare with python?, November 2000. http://faq.rubygarden.org/entry/show/14, Last seen May 2007.

[21] Steven Fink. Rfc: Perl6 type inference. RFC, Website, August . http://dev.perl.org/perl6/rfc/4.html.

[22] Donald Firesmith. Open process framework. Website. http://www.opfro.org/.

[23] Martin Fowler. Railsconf 2007, May 2007. http://martinfowler.com/bliki/RailsConf2007.html.

[24] Hal Fulton. *The Ruby Way, Second Edition: Solutions and Techniques in Ruby Programming (2nd Edition) (Addison-Wesley Professional Ruby Series)*. Addison-Wesley Professional, 2006. ISBN 0672328844.

[25] John B. Goodenough. The ada compiler validation capability. In *SIGPLAN '80: Proceeding of the ACM-SIGPLAN symposium on Ada programming language*, pages 1–8, New York, NY, USA, 1980. ACM Press. ISBN 0-89791-030-3. doi: http://doi.acm.org/10.1145/948632.948634.

[26] Austin HaloStatue. Ruby conference 2006 - matznote, day 2 (saturday, 21 october 2006). Blog, October 2006. http://halostatue.ca/2006/10/24/ruby-conference-2006-matznote-day-2-saturday-21-october-2006/.

[27] Michael R. Hansen and Hans Rischel. *Introduction to Programming Using Sml*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999. ISBN 0201398206.

[28] David Heinemeier Hansson. Ruby on rails. http://rubyonrails.org/.

[29] http://www.ruby lang.org. About ruby. URL http://www.ruby-lang.org/en/about. http://www.ruby-lang.org/en/about, Last seen June 2007.

[30] Mark P. Jones. Functional programming with overloading and higher-order polymorphism. In *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques-Tutorial Text*, pages 97–136, London, UK, 1995. Springer-Verlag. ISBN 3-540-59451-5.

[31] Kristian Kristensen. Rdoc: Xml generate method list. Usenet, March 2007. http://groups.google.com/group/comp.lang.ruby/browse_thread/thread/aa11baddc7c4b5d2/c61fcf2b588f3d6f?lnk=gst&q=kristian+kristensen&rnum=1#c61fcf2b588f3d6f.

[32] Andrew Kuchling. Lj interviews guido van rossum. *Linux J.*, 1998(55es):4, 1998. ISSN 1075-3583.

[33] Martin Madsen and Peter Sørensen. Type inference in ruby. Technical report, Aalborg University, January 2007.

[34] Yukihiro Matsumoto. The ruby programming language, June 2000. http://www.informit.com/articles/article.asp?p=18225&rl=1, Last seen May 2007.

[35] Microsoft Corp. Windows powershell, 2007. http://www.microsoft.com/windowsserver2003/technologies/management/powershell/default.mspx.

[36] Oscar Nierstrasz. Dynamic object-oriented programming with smalltalk: 7. understanding classes and metaclasses. Slides. http://www.iam.unibe.ch/~scg/Teaching/Smalltalk/Slides/07Metaclasses.pdf.

[37] Object Mentor, Inc. Junit, testing ressoruces for extreme programming. Website, 2007. http://www.junit.org.

[38] Jens Palsberg and Michael I. Schwartzbach. Object-oriented type inference. In Norman Meyrowitz, editor, *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, volume 26, New York, NY, 1991. ACM Press. URL citeseer.ist.psu.edu/palsberg91objectoriented.html.

[39] G. Phillips and T. Shepard. Static typing without explicit types. Referred in Agesen [1] but not available to the authors, 1994.

[40] John Plevyak and Andrew A. Chien. Precise concrete type inference for object-oriented languages. In *OOPSLA '94: Proceedings of the ninth annual conference on Object-oriented programming systems, language, and applications*, pages 324–340, New York, NY, USA, 1994. ACM Press. ISBN 0-89791-688-3. doi: http://doi.acm.org/10.1145/191080.191130.

[41] Sidu Ponnappa. Aren't dynamic inheritence changes possible in ruby classes? Weblog, May 2007. http://diningtablecoder.blogspot.com/2007/05/arent-dynamic-inheritence-changes.html.

[42] Lutz Prechelt. An empirical comparison of seven programming languages. *Computer*, 33(10):23–29, 2000. ISSN 0018-9162. doi: http://dx.doi.org/10.1109/2.876288.

[43] RDoc. Rdoc - documentation from ruby source files. Website. http://rdoc.sourceforge.net, Last seen June 2007.

[44] J. A. Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12(1):23–41, 1965. ISSN 0004-5411. doi: http://doi.acm.org/10.1145/321250.321253.

[45] Ruby Application Archive. Ruby application archive. http://raa.ruby-lang.org/.

[46] Ruby Community. Ruby programming language. URL http://www.ruby-lang.org/. http://www.ruby-lang.org, Last seen June 2007.

[47] Ruby Community. Ruby-doc.org - help and documentation for the ruby programming language., 2007. http://www.ruby-doc.org/.

[48] Michael Salib. Starkiller: A static type inferencer and compiler for python, 2004. URL http://salib.com/writings/thesis/thesis.pdf.

[49] Ted Samson. Sun picks up a gem in jruby, September 2006. http://weblog.infoworld.com/techwatch/archives/007818.html.

[50] S. Somasegar. Mix 07 - silverlight shines brighter!, April 2007. http://blogs.msdn.com/somasegar/archive/2007/04/30/mix-07-silverlight-shines-brighter.aspx.

[51] Guy Steele. Dynamic languages wizards series – panel on language design. Panel Discussion, 2001. http://www.ai.mit.edu/projects/dynlangs/wizards-panels.html.

[52] Bruce Stewart. An interview with the creator of ruby, November 2001. http://www.linuxdevcenter.com/pub/a/linux/2001/11/29/ruby.html, Last seen May 2007.

[53] Dave Thomas, Chad Fowler, and Andy Hunt. *Programming Ruby - The Pragmatic Programmer's Guide*. Pragmatic Bookshelf, second edition, October 2004.

[54] Kresten Krab Thorup. Paa konference. Weblog, June 2007. http://www.version2.dk/artikel/2741.

[55] TIOBE Software. Tpci - tiobe programming community index - june 2007. URL http://www.tiobe.com/tpci.htm. http://www.tiobe.com/tpci.htm, Last seen June 2007.

[56] Michael Tonndorf. Ada conformity assessments: a model for other programming languages? In *SIGAda '99: Proceedings of the 1999 annual ACM SIGAda international conference on Ada*, pages 89–99, New York, NY, USA,

1999. ACM Press. ISBN 1-58113-127-5. doi: http://doi.acm.org/10.1145/319294.319310.

[57] David Ungar, Craig Chambers, Bay-Wei Chang, and Urs Hölzle. Organizing programs without classes. *Lisp Symb. Comput.*, 4(3):223–242, 1991. ISSN 0892-4635. doi: http://dx.doi.org/10.1007/BF01806107.

[58] Bill Venners. The making of python, January 2003. http://www.artima.com/intv/pythonP.html, Last seen May 2007.

[59] Bill Venners. The philosophy of ruby, September 2003. http://www.artima.com/intv/rubyP.html, Last seen May 2007.

[60] Jason Voegele. Programming language comparison, December 2005. http://www.jvoegele.com/software/langcomp.html.

[61] Brian A. Wichmann and Z. J. Ciechanowicz, editors. *Pascal Compiler Validation*. John Wiley & Sons, Inc., New York, NY, USA, 1983. ISBN 0471901334.

[62] Wiki. Rubyspec. http://www.headius.com/rubyspec/index.php/Main_Page.

[63] Wikipedia. Closure (computer science), June 2007. http://en.wikipedia.org/wiki/Closure_%28computer_science%29.

[64] Wikipedia. Duck test, June 2007. http://en.wikipedia.org/wiki/Duck_test.

[65] Wikipedia. Garbage collection, May 2007. http://en.wikipedia.org/wiki/Garbage_collection_%28computer_science%29.

[66] Wikipedia. Self programming language, June 2007. http://en.wikipedia.org/wiki/Self_programming_language.

[67] Wikipedia. Test-driven development. Wiki, May 2007. http://en.wikipedia.org/wiki/Test-driven_development.

[68] Wikipedia. Formal verification. http://en.wikipedia.org/wiki/Formal_verification.

[69] Mario Wolczko. *Prototype-Based Programming: Concepts, Languages and Applications*, chapter Self Includes: Smalltalk. Springer, 1999.

[70] Steve Yegge. Digging into ruby symbols, January 2006. http://steve.yegge.googlepages.com/digging-into-ruby-symbols.

APPENDIX A

Acronyms

IDE Integrated Development Environment

LOC Lines of Code

CPA Cartesian Product Algorithm

AST Abstract Syntax Tree

XML eXtended Markup Language

MI Multiple Inheritance

HTML Hyper Text Markup Language

TI Type Inference

CVS Concurrent Version System

RFC Request For Comments

ETDL Extension Type Description Language

RAA Ruby Application Archive

LIFO Last In, First Out

MIT Massachusetts Institute of Technology

NOP No Operation

JVM Java Virtual Machine

CLR Common Language Runtime

APPENDIX B

RDoc Extractor

RDoc is the tool used to document Ruby code. It is for Ruby what JavaDoc is for Java. It supports the extraction of comments and information from the code and the generation of files that includes this information (such as Hyper Text Markup Language (HTML), eXtended Markup Language (XML), etc.). The RDoc distributed with Ruby includes an XML generator, however, it has problems generating the methods of classes and modules. A post to `comp.lang.ruby` did not yield any results [31], so a modified template was created. This template is based on the multi-file HTML template. It is called *xml_from_html.rb* and is included in the *tools*-folder. Running RDoc with this template selected yields the same folder structure as running it with the unmodified HTML template, however, the individual HTML pages contains our XML instead. Each of these files is run through a pre-processor written in Ruby called `rdoc_xml.rb`. It rearranges and formats the XML making it more usable for the Java implementation.

Each file includes where applicable the name of a Ruby class, its included modules, its parent(s) and a list of methods. Each method definition includes its type (class or instance), visibility (public, private), and a list of valid call sequences. This last bit is important for the type inferer, because it yields the arguments types. In some cases these types are straight-forward (*int or nil*, in other cases they require more processing. An example of the generated XML is seen in Listing B.1.

```
1     <tiruby>
2       <class-module-list>
3     <Class name='Fixnum'>
4             <classmod-info>
5       <infiles>
6         <infile>numeric.c</infile>
7       </infiles>
8      <superclass>Integer</superclass>
9        </classmod-info>
10    </Class>
11  </class-module-list>
12      <included-module-list>
13        <included-module name='Precision' href='Precision.html'/>
14      </included-module-list>
15      <method-list>
16<method name='%' id='#M000713'>
17          <Instance-method visibility='Public' id='#M000713'>
18
19          <call-seq-list>
20              <call-seq>
21                <left>fix % other</left>
```

```
22                <right>Numeric</right>
23              </call-seq>
24              <call-seq>
25                <left>fix.modulo(other)</left>
26                <right>Numeric</right>
27              </call-seq>
28            </call-seq-list>
29        </Instance-method>
30        </method>
31        <method name='+' id='#M000708'>
32          <Instance-method visibility='Public' id='#M000708'>
33
34          <call-seq-list>
35              <call-seq>
36                <left>fix + numeric</left>
37                <right>numeric_result</right>
38              </call-seq>
39            </call-seq-list>
40        </Instance-method>
41        </method>
42<method name='&lt;' id='#M000723'>
43          <Instance-method visibility='Public' id='#M000723'>
44
45          <call-seq-list>
46              <call-seq>
47                <left>fix &lt; other</left>
48                <right>true or false</right>
49              </call-seq>
50            </call-seq-list>
51        </Instance-method>
52        </method>
53<method name='&lt;=&gt;' id='#M000720'>
54          <Instance-method visibility='Public' id='#M000720'>
55
56          <call-seq-list>
57              <call-seq>
58                <left>fix &lt;=&gt; numeric</left>
59                <right>-1, 0, +1</right>
60              </call-seq>
61            </call-seq-list>
62        </Instance-method>
63        </method>
64      </method-list>
65    </tiruby>
```

Listing B.1: Our generated XML from RDoc of the `Fixnum` class. This is fed to the Core Loader.

Running the Scripts

The following script fragment illustrates how to run the scripts using PowerShell [35]:

```
1 rdoc --fmt html --all -T xml_from_html --op <OUTPUT_DIR>
2 cd <OUTPUT_DIR>/classes
```

```
3 dir *.html | rename -item -NewName {$_.Name.replace(".html", ".xml")}
4 dir *.xml| foreach { ruby -w <PATH_TO_SCRIPT>/rdoc_xml.rb $_.Name ("tiruby
      -" + $_.Name)}
```

APPENDIX C

Sample Unit Test

This appendix shows an example unit test from the "instance variable"-group. It tests that the attribute accessor methods works properly. Listing C.1 shows the Ruby program used as input for the test. Listing C.2 shows the unit test that tests the types in the Ruby program.

```
1  class Test
2      attr_writer :write
3      attr_reader :read
4      attr_accessor :both
5
6      def initialize
7          @write = "hej"
8          @read = 5
9          @both = 1.0
10     end
11
12 end
13 x = Test.new
14 initRead = x.read # should be Fixnum
15 initWrite = x.write # MethodMissing error
16 initBoth = x.both #should be Float
17
18 x.write = 5 # @write should now be (String, Fixnum)
19 x.read = "hep" # Methodmissing
20 x.both = "test" # @both should now be (String, Float)
21
22 postRead = x.read # should be Fixnum
23 postWrite = x.write # MethodMissing error
24 postBoth = x.both #should be (String, Float)
25
26 puts "Done"
```

Listing C.1: A Ruby program using instance variables. Used as input for the unit test in Listing C.2.

```
1  package instancevars;
2
3  import org.junit.BeforeClass;
4  import org.junit.Test;
5
6  import testsuite.RubyFileTestBase;
7
8  public class AttributesTest extends RubyFileTestBase {
9
10     @BeforeClass
11     public static void setup()
12     {
13         initTestForFile("test/instancevars/AttributesTest.rb");
```

```
14      }
15
16      @Test
17      public void initialReadHaveCorrectTypeSet()
18      {
19          typeSetIs(14, 1, new String[] {"Fixnum"});
20          typeSetIs(16, 1, new String[] {"Float"});
21      }
22
23      @Test
24      public void readAfterChangeHaveCorrectTypeSet()
25      {
26          typeSetIs(22, 1, new String[] {"Fixnum"});
27          typeSetIs(24, 1, new String[] {"String", "Float"});
28      }
29
30      @Test
31      public void missingMethodsGetNilClass()
32      {
33          typeSetIs(15, 1, new String[] {"NilClass"});
34          typeSetIs(19, 1, new String[] {"NilClass"});
35          typeSetIs(23, 1, new String[] {"NilClass"});
36      }
37
38  }
```

Listing C.2: A sample unit test that uses Listing C.1 as input. It validates the types of the instance variables.

APPENDIX D

Experiments

D.1 BMConverter

Number of Files	4
Number of Vertices	16280
Number of Edges	18455
Number of Classes	13
Number of Methods	54
Number of Vertices with Empty TypeSets	991

Table D.1: Generel information about BMConverter

Method name	Number of Vertices	Number of Edges	Number of Clo
<null>::getopts	108	118	1
<null>::printUsageAndExit	7	6	1
<null>::setParenthesis	13	16	8
<null>::setOrAnd	13	16	8
<null>::setExpression	24	29	12
<null>::parseArgs	56	60	1
<null>::usage	18	17	0
<null>::htoutformat	3	2	0
<null>::htinformat	3	2	0
<Formatter>::addindentation	16	15	0
<Formatter>::formatUrl	4	2	0
<Formatter>::formatFolder	4	2	0
<Formatter>::formatSeperator	4	2	0
<Formatter>::formatFolderend	4	2	0
<Formatter>::getHeader	3	2	0
<Parser>::initialize	5	2	0
<Parser>::parse	4	2	0
<FireFox08Formatter>::formatUrl	70	77	0
<FireFox08Formatter>::formatFolder	37	41	0
<FireFox08Formatter>::formatFolderend	5	4	0
<FireFox08Formatter>::formatSeperator	5	4	0
<FireFox08Formatter>::getHeader	9	7	0
<OperaHotlist2Formatter>::formatUrl	76	83	0
<OperaHotlist2Formatter>::formatFolder	59	64	0
<OperaHotlist2Formatter>::formatFolderend	4	1	0
<OperaHotlist2Formatter>::formatSeperator	3	0	0
<OperaHotlist2Formatter>::getHeader	3	1	0
<FireFox08Parser>::initialize	8	4	4
<FireFox08Parser>::parse	10	7	0
<OperaHotlist2Parser>::initialize	8	4	0
<OperaHotlist2Parser>::parse	10	7	0
<Bookmarkstack>::initialize	17	9	4
<Bookmarkstack>::size	3	1	0
<Bookmarkstack>::empty?	3	1	0
<Bookmarkstack>::top	4	2	0
<Bookmarkstack>::push	54	53	0
<Bookmarkstack>::pop	47	49	0

Table D.3: Data about the methods of BMConverter

Method name	Number of Vertices	Number of Edges	Number of Clones
<Bookmarkstack>::debugstate	9	7	0
<Folder>::initialize	22	11	0
<Url>::initialize	30	15	0
<Folderend>::initialize	3	1	0
<Headerend>::initialize	2	0	0
<Seperator>::initialize	3	1	0
<Outputwriter>::initialize	6	4	0
<Outputwriter>::println	18	15	0
<Outputwriter>::screen	11	10	1
<Outputwriter>::debug	11	10	0
<Statistics>::initialize	8	4	1
<Statistics>::incUrl	4	3	0
<Statistics>::incFolder	8	7	0
<Statistics>::decFolder	4	3	0
<Status>::initialize	3	1	1
<Status>::set	14	12	0
<Status>::get	3	1	0

Table D.5: Data about the methods of BMConverter

Instance Variable Name	Typeset size
@stack	2
@linenumber	1
@elements	1
@max_size	1
@statistics	2
@formatter	1
@output	1
@firsturl	1
@state	2
@size	0
@name	0
@depth	0
@id	0
@createdtimestamp	0
@lastmodifiedtimestamp	0
@ispersonalfolder	0
@expanded	0
@description	0
@inpersonalfolder	0
@personalfolderposition	0
@url	0
@shortcuturl	0
@nickname	0
@createdtimestamp	0
@lastvisitedtimestamp	0
@lastmodifiedtimestamp	0
@icon	0
@charset	0
@personalfolderposition	0
@numoffolders	1

Table D.6: Data about the instance variables of BMConverter

Typeset size	Number of Vertices with Size
0	528
1	10597
2	1853
3	48
4	0
>5	0

Table D.7: Sizes of the typesets of BMConverter vertices that represent data, call vertices not counted

Run	Runnig Time (in seconds)
1	7.36
2	7.44
3	7.42
4	7.67
5	7.51
Average	7.48

Table D.8: Running times for the tool, inferring types for the program BMConverter

D.2 Canna2Skk

Number of Files	3
Number of Vertices	1002
Number of Edges	1137
Number of Classes	0
Number of Methods	2
Number of Vertices with Empty TypeSets	132

Table D.9: Generel information about Canna2Skk

Method name	Number of Vertices	Number of Edges	Number of Clones
<null>::getopts	108	118	1
<null>::hiragana?	13	10	0

Table D.11: Data about the methods of Canna2Skk

Typeset size	Number of Vertices with Size
0	71
1	524
2	30
3	15
4	0
>5	0

Table D.12: Sizes of the typesets of Canna2Skk vertices that represent data, call vertices not counted

Run	Runnig Time (in seconds)
1	1.68
2	1.72
3	1.52
4	1.61
5	1.57
Average	1.62

Table D.13: Running times for the tool, inferring types for the program Canna2Skk

Number of Files	2
Number of Vertices	283
Number of Edges	245
Number of Classes	0
Number of Methods	0
Number of Vertices with Empty TypeSets	10

Table D.14: Generel information about e

D.3 e

Typeset size	Number of Vertices with Size
0	9
1	189
2	6
3	30
4	0
>5	0

Table D.15: Sizes of the typesets of e vertices that represent data, call vertices not counted

Run	Runnig Time (in seconds)
1	1.49
2	1.49
3	1.79
4	1.58
5	1.48
Average	1.57

Table D.16: Running times for the tool, inferring types for the program e

Number of Files	4
Number of Vertices	1219
Number of Edges	1218
Number of Classes	8
Number of Methods	6
Number of Vertices with Empty TypeSets	321

Table D.17: Generel information about FreeRIDE

D.4 FreeRIDE

Method name	Number of Vertices	Number of Edges	Number of Clones
<GetoptLong>::initialize	34	24	1
<GetoptLong>::ordering=	30	28	0
<GetoptLong>::set_options	17	12	0
<GetoptLong>::terminate	27	20	0
<GetoptLong>::terminated?	5	3	0
<GetoptLong>::set_error	20	14	0
<GetoptLong>::error_message	3	1	0
<GetoptLong>::get	206	224	0
<GetoptLong>::each	2	1	1
<null>::usage	14	13	1

Table D.19: Data about the methods of FreeRIDE

Instance Variable Name	Typeset size
@ordering	1
@canonical_names	1
@argument_flags	1
@quiet	0
@status	1
@error	1
@error_message	1
@rest_singles	1
@non_option_arguments	1

Table D.20: Data about the instance variables of FreeRIDE

Typeset size	Number of Vertices with Size
0	191
1	645
2	25
3	4
4	0
>5	0

Table D.21: Sizes of the typesets of FreeRIDE vertices that represent data, call vertices not counted

Run	Runnig Time (in seconds)
1	1.64
2	1.58
3	1.65
4	1.76
5	1.76
Average	1.68

Table D.22: Running times for the tool, inferring types for the program FreeRIDE

Number of Files	2
Number of Vertices	424
Number of Edges	493
Number of Classes	0
Number of Methods	4
Number of Vertices with Empty TypeSets	122

Table D.23: Generel information about ICalc

D.5 ICalc

Method name	Number of Vertices	Number of Edges	Number of Clones
<ICALC>::to_b	38	41	0
<ICALC>::to_d	43	45	0
<ICALC>::to_i	51	56	0
<ICALC>::to_p	46	52	0

Table D.25: Data about the methods of ICalc

Typeset size	Number of Vertices with Size
0	54
1	182
2	15
3	8
4	1
>5	0

Table D.26: Sizes of the typesets of ICalc vertices that represent data, call vertices not counted

Run	Runnig Time (in seconds)
1	1.51
2	1.53
3	1.46
4	1.55
5	1.45
Average	1.50

Table D.27: Running times for the tool, inferring types for the program ICalc

Number of Files	2
Number of Vertices	515
Number of Edges	524
Number of Classes	3
Number of Methods	7
Number of Vertices with Empty TypeSets	160

Table D.28: Generel information about Markovnames

D.6 Markovnames

Method name	Number of Vertices	Number of Edges	Number of Clones
<Array>::random	5	6	1
<String>::wrap	33	34	1
<MarkovNameGenerator>::initialize	13	7	1
<MarkovNameGenerator>::read	6	3	0
<MarkovNameGenerator>::input	10	8	0
<MarkovNameGenerator>::name	13	9	1
<null>::usage	24	23	1

Table D.30: Data about the methods of Markovnames

Instance Variable Name	Typeset size
@chains	1
@input_set	1
@randomness	1
@ngram_size	1
@progress	1
@chains	1

Table D.31: Data about the instance variables of Markovnames

Typeset size	Number of Vertices with Size
0	87
1	236
2	8
3	0
4	0
>5	1

Table D.32: Sizes of the typesets of Markovnames vertices that represent data, call vertices not counted

Run	Runnig Time (in seconds)
1	1.52
2	1.46
3	1.52
4	1.57
5	1.44
Average	1.50

Table D.33: Running times for the tool, inferring types for the program Markovnames

Number of Files	2
Number of Vertices	85
Number of Edges	69
Number of Classes	0
Number of Methods	1
Number of Vertices with Empty TypeSets	36

Table D.34: Generel information about Quickey

D.7 Quickey

Method name	Number of Vertices	Number of Edges	Number of Clones
<null>::main	21	17	1

Table D.36: Data about the methods of Quickey

Typeset size	Number of Vertices with Size
0	23
1	39
2	0
3	0
4	0
>5	0

Table D.37: Sizes of the typesets of Quickey vertices that represent data, call vertices not counted

Run	Runnig Time (in seconds)
1	1.35
2	1.50
3	1.40
4	1.30
5	1.30
Average	1.37

Table D.38: Running times for the tool, inferring types for the program Quickey

Number of Files	3
Number of Vertices	788
Number of Edges	567
Number of Classes	2
Number of Methods	5
Number of Vertices with Empty TypeSets	56

Table D.39: Generel information about Roman

D.8 Roman

Method name	Number of Vertices	Number of Edges	Number of Clones
<Roman>::to_int	20	16	0
<Roman>::to_roman	30	26	1
<Roman>::reverse_hash	6	3	2
<Roman>::method_missing	4	3	0
<Integer>::to_roman	6	5	1

Table D.41: Data about the methods of Roman

Typeset size	Number of Vertices with Size
0	32
1	582
2	12
3	0
4	0
>5	0

Table D.42: Sizes of the typesets of Roman vertices that represent data, call vertices not counted

Run	Runnig Time (in seconds)
1	1.47
2	1.45
3	1.51
4	1.44
5	1.52
Average	1.48

Table D.43: Running times for the tool, inferring types for the program Roman

D.9 Projects with Error Conditions

Project Name	Error Condition
CLwiki	RuntimeException -> Method not yet implemented
Depends	NullPointerException -> null
FTP_Sync	RuntimeException -> Method not yet implemented
naninhttpd	SyntaxException -> Invalid char 177775 in expression
newsstats	RuntimeException -> Method not yet implemented
Qant	RuntimeException -> Method not yet implemented
ROOF	ClassCastException -> Master cannot be cast to ConstraintGraph
Setup	RuntimeException -> Method not yet implemented
Slider	ClassCastException -> Colon2Node cannot be cast to ConstNode
Sync_citydesk	RuntimeException -> Method not yet implemented
Tiddy	RuntimeException -> Method not yet implemented
Yawee	RuntimeException -> Method not yet implemented

Table D.44: Project that fail to have types inferred for them