ANALIST: A TOOL FOR IMPROVED STATIC TYPE ANALYSIS FOR RUBY

TWAN COENRAAD

UNIVERSITY OF TWENTE.

Master's Thesis

Master of Science
Formal Methods and Tools for Verification
Faculty of Electrical Engineering, Mathematics and Computer Science
University of Twente

September 2017 – February 2018

Twan Coenraad: *Analist: A Tool for Improved Static Type Analysis for Ruby*, Master's Thesis, © September 2017 – February 2018

SUPERVISORS:
prof. dr. M. Huisman
dr. A. Fehnker
ir. I. van Hurne (Moneybird)
J.C.G. Weeink BSc (Moneybird)

LOCATION:
Enschede, the Netherlands

TIME FRAME:
September 2017 – February 2018

The *best things* happen by chance.

— Dory

ABSTRACT

Dynamically typed languages pose both inherent advantages and disadvantages towards developers. The lack of a static typing system results in having lots of freedom during development, at the cost of having to deal with typical run-time errors, like *type* errors, *argument* errors and *no method* errors. Earlier research has been conducted to deal with this type uncertainty, e. g., by developing analytic tools that can validate statically a dynamically typed code base. However, most of the time these tools give many false positives, therefore, they are not helpful for a developer to use in a real-world scenario. In this research a tool named Analist is developed for the Ruby language, focused on using a pragmatic approach. This means that wherever assumptions (e. g., when derived from the database schema file) are quite safe to be made, this is done. By design, this cannot be as complete as previously developed tools, yet it turns out to be a promising way of preventing programmer errors to occur as both synthetical benchmarks and an experiment with developers confirm. This balance makes Analist a tool that is useful for developers. In future work one can consider to add more Ruby type definitions to Analist to make it even more useful, as for this research the span of what can be analyzed correctly was clearly limited.

I've never done that before, so I'm sure I can do it!

— Pippi Longstocking

ACKNOWLEDGMENTS

I thank my supervisors from the University of Twente Marieke and Ansgar for their time reading (and re-reading) my earlier versions of this master's thesis. In particular I thank Marieke for our countless meetings, talking about both serious and light-hearted business.

Next, I thank both Ivo and Jeroen from Moneybird for their day to day supervision at Moneybird, for their useful insights during our meetings and in between. Furthermore, I thank all my colleagues, co-graduates and fellow interns at Moneybird that made my graduation project very pleasant to do. In particular I learned a lot from Thomas, with whom I have tinkered a lot finding a good approach for Analist.

At last, I thank Thomas, Wietze and Jip for their proof reading and other useful graduate advice. I thank my girlfriend Joyce for being patient and understanding, even when working during late hours.

CONTENTS

ACRONYMS AND DEFINITIONS

AST	Abstract Syntax Tree
DRuby	Dynamic Ruby, http://www.cs.umd.edu/projects/PL/druby/
ERB	Embedded Ruby, https://apidock.com/ruby/ERB
gem	Gems are the Ruby version of prepared libarires that provide some specific functionality, https://rubygems.org
HTML	HyperText Markup Language
HTTP	Hypertext Transfer Protocol
linter	A plugin that gives feedback about the code that is written, commonly inline and standalone
LOC	Lines of Code
metaprogramming	Metaprogramming is programming a (meta) program that, when executed, results in a new program that again can be executed
Rails	Ruby on Rails, http://rubyonrails.org/
RPython	Restricted Python, https://rpython.readthedocs.io/
Rubocop	Rubocop, https://github.com/bbatsov/rubocop/
Ruby	Ruby, https://www.ruby-lang.org/
SQL	Structured Query Language
XSS	Cross-site scripting, https://www.owasp.org/index.php/Cross-site_Scripting_(XSS)

Part I

INTRODUCING ANALIST

In this part, the research domain is explored, the problems and solutions in similar research are explained and conclusions are drawn with respect to what key features are for Analist and what problems should be avoided.

1 INTRODUCTION

In this chapter, an introduction is given about the type checking problem Analist tries to solve. Section 1.1 tells what the high-level background is of the problem. Sections 1.2 and 1.3 introduce both the programming language and the framework commonly used together. Section 1.4 sets forth what the company Moneybird is, and why they are interested in a solution for this problem. Then, Section 1.5 explains what contribution is given to the scientific world. At last, the research question (Section 1.6) and structure of this thesis (Section 1.7) are set out.

1.1 PROBLEM STATEMENT

1.1.1 *Background*

Programming languages can be roughly divided into two groups. On one hand there exist *dynamically typed languages* in which every variable is only bound to an object and can be re-assigned when desired. On the other hand, there exist *statically typed languages* in which variables are bound to both a type and an object. Once declared, that variable can only be assigned objects of the defined type.

Dynamically typed languages therefore offer large flexibility during development since no type checks are performed before the code is executed. Only at run-time these languages try to handle a method call on an object and give errors when that turns out to be impossible. This also makes it possible to have objects of different types with a similar interface, sharing some methods, without having to hard-code this behavior. These interfaces are only implicitly given and not enforced in any way.

Statically typed languages instead, give more guarantees with regard to types and as a result to their available methods. In statically typed languages code is typically compiled or type inferred and afterward type checked before it is executed, giving a developer type errors in this phase of development. As a direct result, a developer is required to define all types explicitly, or at least in a very strict manner.

The flexibility of having types that can be adapted without the formal administration, is at the cost of having less certainty with regard to the correct use of types, making type errors in dynamically typed languages more likely to occur. This is what we want to address in this research.

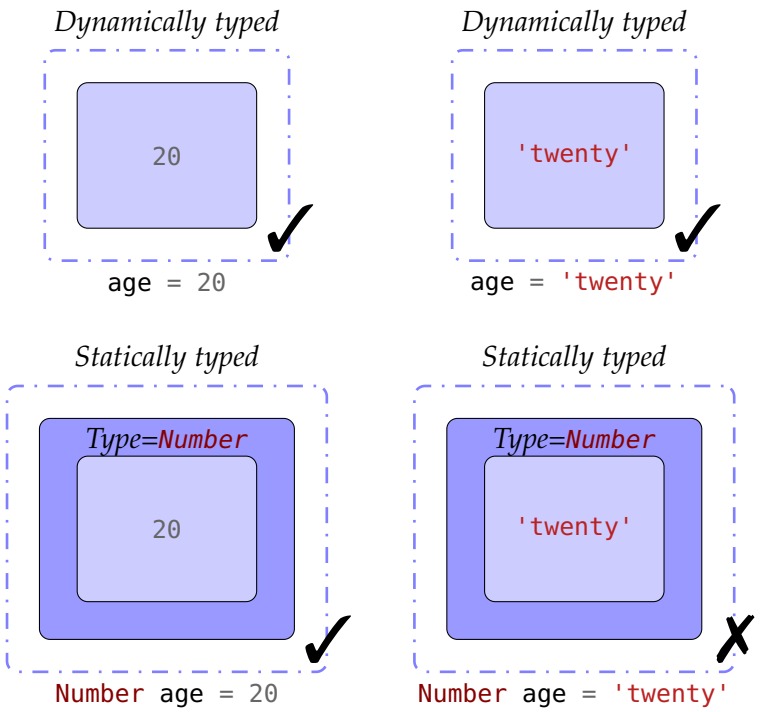


Figure 1.1: Dynamically typed vs statically typed variables

1.1.1.1 *An explanatory example*

To better understand what the difference is between the two types of languages, an example has been put forward below to illustrate its working. Refer to Figure 1.1 for a visualization of the example below.

Imagine you want to save the *age* of someone who is 20 years old. When you save it in a *dynamically typed* language into a variable like `age = 'banana'`, then this is valid syntax, although it is clear that this is not what was intended. When putting it like `age = 'twenty'`, it is not at a glance visible that this is *probably* not what is intended, although there is no way to check this. For instance, you cannot calculate someone's birth year with the word 'twenty'. However, you can when saving it like a number, simply with `age = 20`. This also shows an advantage of dynamically typed languages as it is easy to switch types. In *statically typed* languages such a type error cannot occur, as it is necessary to lock the type of a variable, e. g., `Number age = 20`. `Number age = 'twenty'` is thereafter simply not allowed by the *type checker* of the statically typed language. This restricts a developer to have only one type for every variable and stick to it throughout the program.

1.1.2 *The problem*

This research is trying to bring the profits of having static types to the *dynamically typed* language Ruby, used in conjunction with Ruby on Rails (Rails). It introduces a type checking tool that can be used to do static type checking similar to the work done by a type inference system, commonly part of a compiler. The tool exploits all the knowledge available, derived from the project environment. In that way, the flexibility of Ruby can be kept, while also having more certainties that on run-time no type errors occur.

The most important requirements to the analyzer are that it is reasonably fast, works out-of-the-box and gives almost no *false positives*. Ideally, a developers team should be able to add the tool as a step of the build process. This research focuses on errors that are both a result of the dynamic nature of Ruby and are commonly made by developers. Default Ruby and Rails behavior is expected, therefore it is explicitly not tried to have full coverage of all possible exceptions that possible can be programmed. In other words, the tool allows abnormal Ruby code to pass falsely.

1.1.3 *Requirements*

We want to build a static analysis tool with the properties as listed below. The properties are based on the result of similar research as can be found in Chapter 2.

- It *must* do a static analysis focusing on type checking for Ruby, in particular on a project that uses Rails as its framework, both for their latest released versions.
- It *must* show only relevant errors, thus only when it is *almost* certain that it is a programmer's mistake and will result in run-time errors.
- It *should* be flow-sensitive, context-sensitive and interprocedural (see Section 2.5.3 for an explanation of these terms). It was found in earlier research that this may improve results significantly.
- It *should* take advantage of any supported gems that are available within the project. For instance, Moneybird uses the `mutations` gem, that enforces run-time validation for models and gives an outline for what the data model should look like, including semi-automatic coercion (implicit casting) from one value type to another. The database schema as defined in source code tells how the object's fields are defined. Rails models tell what relations they have and can be used as an anchor for what kind of object is to be expected when referring to it.

- It *should* be adapted to work with Rails out-of-the-box, as is common for gems that support Rails. This means that simply adding the program and running it would be enough to give decent results with sane, yet opinionated default settings.
- It *should* be possible to use it within an automatic building process.
- It *should* be possible to configure what kind of errors and warnings are given, to make the tool as compatible to a developer's style as possible.
- It *should* be fast enough to be able to run the program on a Ruby file after each save, or a shorter time frame.
- It *could* preferably have a way of saving an initialized run-time state. This is suggested by DRuby and implemented in RPython. This might make it easier to deal with variables that dependent on the run-time environment.

1.2 RUBY, THE PROGRAMMING LANGUAGE

Ruby is a *programming language* that is dynamically typed. Although the hype on Ruby is over [14], it is still widely used in conjunction with the Rails web development framework. What developers like about Ruby is its very simple, almost English-like syntax. A typical example illustrating this is the following in which an operation is performed repetitively:

```
1 35.times do
2   puts 'Hello world!'
3 end
4 # Hello world!
5 # Hello world!
6 # ...
```

Also, for most common operations on objects, a native Ruby method is available, keeping Ruby code very clean and easy to read. This makes Ruby a powerful language during development and easy to learn, with a clean and concise syntax. Also, within the Ruby community many gems exist that take care of most common programmer challenges. Next to the disadvantages that are common for dynamically typed languages (see Section 1.1.1.1), also the ability to extend and overwrite anything at run-time is often considered a weak spot.

1.3 RUBY ON RAILS, THE FRAMEWORK

Ruby is often used together with the Rails *web development framework*. Noteworthy Rails applications include Airbnb (marketplace for accommodation rent), DigiD (e-identity provider for the Dutch government) and Moneybird (accounting software). Rails is appreciated for taking care of most common problems that occur in web development, e. g., manipulating objects in a database, handling HTTP requests and rendering templates. Rails' philosophy is using *convention over configuration*, meaning that Rails' default values will probably be sane with regard to what is commonly done when dealing with a certain problem.

The main drawback of using Rails as web development framework is that it does not scale that well under heavy use. This is mainly due to Ruby, as it is considered not the best performing language. Next, the convention over configuration philosophy also comes with a lot of *magic methods* and *configuration options* that have no clear origin or defined behavior.

1.4 MONEYBIRD

This research was commissioned by Moneybird. Moneybird was founded in 2008 as an online software service for sending invoices from entrepreneurs to their customers. Nowadays, that service has evolved into a full-featured bookkeeping service. It is used by over 150.000[1] entrepreneurs, mainly in the Netherlands.

Moneybird's main application and its corresponding microservices are all built in Ruby using Rails, which was in 2008 a popular choice for building web applications [16]. Nowadays, there is a large code base for its main application (168K LOC in Ruby files, counted using `cloc`[2]). In 2017, about 2.5K files were changed, with 36K insertions and 100K deletions[3].

The motivation for this research is that Moneybird is curious to see whether type checking is possible on a reliable level in real-world projects, which are written in dynamically typed languages, like their own. At this moment it frequently happens that very similar types are mixed or interchanged when refactoring (e. g., see Listing 1), which results in run-time errors. When these simple errors can be prevented by running Analist, less critical errors will remain in source code when an application is eventually in production. It is known that for an application like Moneybird, about 2-70 errors per 1K LOC can be expected [13].

1 As stated on https://www.moneybird.nl/, accessed on January 23th, 2018
2 https://github.com/AlDanial/cloc,limitingtowardsapplicationcodesolely, accessed on January 23th, 2018
3 Measured using `git diff --shortstat`

```
1 -   <span class="input-append-nobg"><%=
  ↪   administration.show_tax_number_icon %> </span>
2 +   <span class="input-append-nobg"><%=
  ↪   administration.decorate.show_tax_number_icon %> </span>
```

Listing 1: Example of a refactoring error (in red). It would be detected (partially) by ANALIST. *Note: without the call to* `decorate`*, the* `administration` *object is missing methods, exactly what is causing a bug here*

During mid-December 2017 up until mid-January 2018 about 30 *type* errors, about 260 *argument* errors and about 1260 *no method errors*[4] were raised in their main application[5]. These errors specifically are subject of this research.

1.5 CONTRIBUTION

This research tries to build a static type checker for a dynamically typed language with a *pragmatic* approach, in contrast to other researchers that take a theoretical approach.

The contribution of this research is mainly the design of a type checker, a proof of concept in a full-fledged implementation including code editor tool support and an evaluation of this approach to show to what extent such type checkers can be more helpful than those with a true theoretical approach.

1.6 RESEARCH QUESTION

The research question we want to answer is:

To what extent is it possible to create ANALIST*, a static type analysis tool for Ruby on type checking that conforms to the requirements as put forward in Section 1.1.3?*

Subquestions that arise are:

- To what extent is there benefit from using information that is being exposed by some gems?
- To what extent do developers have benefit from using ANALIST?

1.7 STRUCTURE

The report is structured as follows. In Chapter 2 related work is reviewed and compared exhaustively, to end with all lessons that were

4 *No method errors* were only counted when the object exists, yet the method called was not – i. e., method calls for `NilClass` were ignored

5 All numbers originate from Moneybird's error logging service

learned from earlier research. Chapter 3 gives an abstract overview of how ANALIST is built up and how it works. Chapter 4 explains how the actual implementation was done, including all technical details. Validation and performance of ANALIST are shown in Chapter 5. At last, conclusions are drawn and future work is listed in Chapter 6.

2 RELATED WORK

In this chapter related work regarding type checking in dynamically typed languages is discussed. Several research on the use of type checkers for dynamically typed languages has been conducted, such as for PHP with Phantm (Section 2.2.1), Pixy (Section 2.2.2) and WeVerca (Section 2.2.3), Python with RPython (Section 2.3.1) and also for Ruby, in a dialect language called DRuby (Section 2.4.1).

2.1 FEATURE DEFINITIONS

To start with, definitions of features type checkers typically have are given.

2.1.1 *Flow-sensitivity*

Flow-sensitivity means that the order in which statements are executed matters. This is of interest as it is possible to reuse a variable in a dynamically typed language with a different type. All code examples are written in Ruby.

```
1 a = 1
2 a = a + 1 # 2
3
4 a = 'ab'
5 a + '1' # 'ab1'
```

Listing 2: Example of flow-sensitive code

With *flow-sensitive* analysis, it is possible to make this snippet pass, because then the analyzer is aware of the dynamic type change on line 4. Without, it could also be typed with `number`, `string` and give warnings.

2.1.1.1 *Context-sensitivity*

Context-sensitivity means that when a method is analyzed, the specific context in which it is called is taken into account within the method's body [3]. In contrast, when context-insensitive, it would be analyzed as a method on its own, resulting in a less concise result.

```ruby
def func(arg)
  if arg
    return 2
  else
    return false
  end
end
```

Listing 3: Example of context-sensitive code

In the listing above, depending on the input `func` either returns a number (`2`) or a boolean (`false`). A context-sensitive analysis takes this into account, whereas a context-insensitive analysis would suggest that either a number or a boolean is returned, losing precision.

2.1.2 *Interprocedural support*

Interprocedural support means that method invocation is handled in a correct way. This is of great concern with regards to scoping of variables and whether nested calculations are visible to the outer world.

```ruby
def func(b)
  b = 2
end

b = 3
func(b)

b # 3
```

Listing 4: Example of code that requires support for handling procedures

The return value of the code in the listing above will depend on the programming language: either it is still `3` (*pass-by-value*) or it becomes `2` (*pass-by-reference*). In case of Ruby the former is done.

2.1.3 *Path-sensitivity*

Path-sensitivity means that the branches a program takes depending on a certain state at run-time matters. This is of interest, e. g., when a method has multiple return types.

```ruby
bool = true

if bool
  value = 2
else
  value = false
end

value + 3 # 5
value + 3 # NoMethodError: undefined method `+' for
 ↪  false:FalseClass
```

Listing 5: Example of code that is subject to support for multiple paths

With *path-sensitive* analysis handling it is possible to let pass the snippet above. In that case, the analyzer is aware that the type of `value` is `number`. Without, it can also be typed with `boolean` and give warnings when an analyzer is uncertain.

2.1.4 *Supports object-oriented design*

Supporting object-oriented design simply means that support for classes and objects is added. As most applications are built with this or a similar principle, it is valuable when ANALIST supports it.

2.1.5 *Evaluation patterns support*

Supporting evaluation patterns means that dynamically generated and evaluated code is supported.

```ruby
def func(func_name)
  eval "def #{func_name}(arg); 3 + arg; end"
end

func('abc')
abc(3) # 6
```

Listing 6: Example of an evaluation pattern

In the snippet above, `func` defines dynamically a method after giving it a method name. Therefore, after running `func('abc')` the method `:abc` is created, that expects exactly 1 argument. When this pattern is not supported, especially programs that depend on metaprogramming are hard to analyze correctly.

2.2 PHP

PHP is a dynamically typed language with a focus on web development, that was in the early internet days very popular for self-learning web programmers. A reason for this is that it is very simple to combine HTML with PHP, using PHP simply as templating engine that could interact with a database, fetch and manipulate stored data and return the rendered web page. A lot of programming tutorials exist for PHP that are written by other self-learning programmers. These tutorials were therefore mostly of poor quality and as a result vulnerabilities in PHP websites were very common, including *SQL injections*[1] and *XSS* attacks. All research below focus mainly on this last aspect: finding of *tainted* (unsafe) data in the source code of an application.

2.2.1 *Phantm*

Phantm [12] tries to improve type analysis in PHP by focusing on the gap that arises by the necessary approximation for keeping the performance at a reasonable level and absence of *environment-specific information* at run-time. It takes a hybrid approach to circumvent this problem by running the program as usual and then *capture the program state* at a point where most set-up configuration has finished. This program state is then used to do a static analysis. An example (in Ruby) is depicted below:

```
1 debug = ENV['DEBUG']
2 puts 'Starting program in debug mode' if debug
```

If Phantm captures program state after starting the program, the value of `ENV` containing all environment variables will be known, including what files are actually loaded. Then, it is possible to *prune code paths* that depend on this. In the example above, the second line can simply be ignored when it is known that the debug flag is not set on production.

A library was created to enable developers to mark a certain point in the code to collect the current state, which stores keys and values into a *state file*. During this process, only simple values like scalar values and arrays are taken into account.

A flow-sensitive approach is used, not only to deal with variable type changes correctly but also to follow values of associative arrays, which are commonly used as configuration options objects. Phantm has detailed information about built-in functions available, which can be extended with user-defined functions that can be annotated using PHP's documentation features to improve its results. Addi-

1 https://www.owasp.org/index.php/SQL_Injection, accessed on July 19th, 2017

tional warnings are emitted when uninitialized variables or uninitialized array entries are referred. This is done to take care of PHP's `register_globals` [2] that was enabled by default in earlier versions of PHP, which resulted in a large source of vulnerabilities.

When the analysis is performed, the following steps are taken:

1. A *concrete state* is captured as a map of variable names to their values and a heap that contains object references to object states. These object states are mappings of fields to their values. This concrete state is what is saved in the described state file.

2. An *abstraction function* is applied, putting the concrete state in an abstracted form. It takes individual variables values and abstracts them into certain classes, e. g., integers, strings, and maps. When a value is *known*, for example when it is used as the index of an array, that value is abstracted. When a concrete value is *unknown*, it is marked as such. The set of concrete objects contains possible real-world memory locations in the heap, whereas the set of abstract objects contains a set of program points where objects can be created. Special care is taken of undefined references versus nullified variables. From a PHP perspective, they behave the same, however, the former is most likely unintentional and should eventually be warned for.

3. When transforming the abstracted form by applying transfer functions for each consecutive statement, the following features are highlighted:
   - Any time information can be derived from a variable's type, *type refinement* is applied. Types are refined by computing the new lattice that is the meet of the current type lattice and the expected variable types.
   - With that type information available, *conditional filtering* can be applied. This is an extension to type refinement in which the value of a variable is used to predict which branch of an `if` statement is taken. This makes it possible to find methods returning `false` on errors and else some value, which is a common pattern when querying for values in PHP. When it is found that a certain `if` statement is impossible to fulfill, unreachable code is detected.
   - *Termination is enforced* by setting hard limits on array depth and by ensuring that any time a new type is introduced, it is equally, or wider than any type that is known before. The researchers state that this approach works well in practice.

2 `register_globals` turns any GET and POST field in an HTTP request into a variable, making it very simple to inject user input into an application.

4. When the analysis reaches its fixed point, all types are extracted, insofar possible. It will then make a final pass over the control flow graph of the program and give all errors detected. All type information that is available after type refinement is added to the type mismatches. The level of detail in the report can be configured.

2.2.1.1 *Conclusion*

Phantm found a useful new insight on type refinement by saving run-time state. The path pruning that is made possible due to this refinement is also a good finding. Manually modeling the built-in functions and supporting the expansion of this modeling to user-defined classes is a great way of making a widely applicable analyzer. Their choice to capture a state just after initialization seems to be a helpful approach overcoming typical problems with static analysis on dynamically typed languages. This resembles what RPython (see Section 2.3.1) does in some way, namely the two-phase process of doing a code analysis, by first performing dynamic processing and only then performing an analysis. Alias analysis as Pixy supports is omitted, but it is not considered to be any large flaw. Also dealing with executing dynamically created code (using `eval`) is not mentioned, but also this is not much of a great deal.

2.2.2 *Pixy*

Pixy [9] takes a flow-senstive, interprocedural and context-sensitive data flow approach, focusing on finding taint-style (unsafe input) vulnerabilities. When given with a PHP program, an analysis looks like this:

1. *Constructing an abstraction* is done by PhpParser[3], a tool made specially for Pixy. It is a combination of a lexical analyzer, parser program, and specification files. The specification files are part of the PHP interpreter. During this step, the source code is transformed into objects that can easily be traversed through.

2. *Deriving an intermediate representation* afterwards gives a linearized form of the plain PHP script, similar to three-address codes [4]. This linearized form flattens out all possible loop patterns that exist, e. g., `for` and `while` and turn every function into a simple control flow graph. Global scoped code that does not belong to any function is put in a `main` function.

3 https://github.com/oliverklee/phpparser

4 A three-address code commonly consists of an assignment and a binary computation, e. g., `a = true OR false`.

a) Variables, constants and literals are turned into *place* abstractions. These abstractions are used to store more concise information, when available.

b) Functions are turned into three control flow graph nodes, namely a *call preparation*, a *calling*, and a *returning call* node.

3. During *alias analysis* so-called *alias groups* are created that have identifiers all pointing to the same memory location, e. g., `a = &b`, results in an alias group of `(a, b)`. When no aliases have been defined for a variable, the corresponding identifier is put alone in an alias group. When it is uncertain what alias is built, for example when two `if` branches result in different aliases, so-called *may-aliases* are created. When certain what exact alias is built, *must-aliases* are created. When resolving a lattice based on these may-aliases and must-aliases, may-aliases are ordered above must-aliases, which results in a loss of precision, as *may-aliases* contain multiple alternatives. When defining *transfer functions*[5], most statements remain untouched, with the exception of a few. E. g., *reference assignments*, `a = &b` are processed by removing `a` from all alias groups and adding `a` everywhere `b` already is. For *interprocedural transfer function calls*, a problem mentioned specifically is that it can be hard to determine statically how deep a recursive function call will be, e. g., when such a function ends conditionally. To scope variables correctly when calling functions recursively, *call preparation* and *call return* nodes store and restore *alias information* and the value it currently holds. To simplify function parameter handling with regard to function calls, those values are treated the same way.

4. During *literal analysis* it is determined at every program point what all literals can hold. It is performed to make it possible to make analysis more concise, e. g., by pruning *unreachable* code paths, or making variable array indices fixed. Information gathered during the alias analysis is incorporated here. For all variables and constants that exist in the analyzed program, a lattice is defined, refined and then resolved. The top element of this lattice refers to Ω, meaning *absolutely nothing* is known up until this point. For the *transfer function* it is defined that on assignments, depending on whether a variable is considered a simple variable[6], an array, or array element and whether it is a *may-alias* or *must-alias* as determined in the alias analysis, strong or weak updates or strong or weak overlaps are applied. Updates are performed on simple variables, whereas overlaps are performed on arrays and array elements. In a *strong update*, a vari-

5 A transfer function defines how data is transformed when it flows through a node.

6 Simple variables are in this paper considered any variable, not an array, nor an array element.

able is just overwritten with the right-hand side's value. In the case of a *weak update* all aliases of the left-hand side are bound to the least upper bound between the literal that such an alias already holds and the right-hand side of the assignment. *Strong* and *weak overlaps* are handled similarly, with the difference that actions are performed on the array elements within. If an array index contains a non-literal, like `$a[$i]`, it is mapped to Ω. When dealing with *unary operations*, e. g., `$a = -1`, or *binary operations*, e. g., `$a = 1 + 2`, first such calculations are performed and then the result is assigned as explained before. All *built-in functions* are mapped to Ω, thus knowing nothing about its internal behavior. *Reference assignments* nodes result in a bare overwrite, as the authors restrict themselves to simple variables with regards to reference assignments. It is assumed that *referenced* variables are not redirected to other variables within a method call. All aforementioned actions result in loss of precision, although the authors report that actual analyses are not really influenced by this.

5. At last, during *taint analysis* maps variables to be either *tainted* or *untainted*. A conservative approach is used, meaning that only when it is certain that a value is safe, it is considered *untainted*, else it is *tainted*. This results in having array elements with non-literal indices considered tainted, as they are mapped to Ω during literal analysis. An exception for this pessimistic approach is made for newly created arrays, that are explicitly flagged to be clean. Within the *transfer function* it is defined that sanitization can be achieved by both typecasting and using sanitizing PHP functions. Where applicable, the clean array flag is passed on. In contrast to what is done in the literal analysis, built-in functions are in this analysis modeled more faithful to reduce the appearances of false positives.

After that all analyzes have finished, for each sensitive sink, e. g., a place that is shown to the user or used within an SQL statement, it is considered whether that sink can have tainted input variables. When that is the case, a warning is displayed.

2.2.2.1 *Conclusion*

Pixy focuses mainly on doing a correct alias analysis. Albeit a sophisticated solution, most code programmed will not have many aliases, if any, because it is considered bad practice. Moreover, associative arrays[7] and objects are not supported correctly when used in alias analysis according to [5]. Next, the lack of having support for classes is a major flaw, given that most real-world applications will have classes

7 Associative arrays are known as hash maps, or dictionaries in other languages

and objects to separate concerns better. Type inference is not considered at all. An asset is the approach being flow-sensitive, interprocedural and context-sensitive, but altogether, Pixy cannot be considered production-ready.

2.2.3 *WeVerca*

WeVerca [5][6] tries to bring a framework to do a full-featured analysis on PHP. It captures the internal behavior as correct as possible, by resolving method calls, *include* statements and getting values out of an object. WeVerca focuses on the difficult interplay between value analysis and heap analysis, which come into play when dealing with for example associative arrays. When dealing with arrays, a variable can point to a specific value within such an array, while that array can be indexed by another variable as well. That variable and its (primitive) value are captured by the *value analysis*, whereas the array is caught during the *heap analysis*. By splitting the analysis into two parts and first performing an abstract analysis followed by performing an end-user analysis, it is thought that it is easier to extract any information out of a dynamically typed language. As a proof of concept, a *taint analysis* as end-user analysis is implemented.

WeVerca outlines the first phase as follows:

1. In the first step, the *control flow* is saved into an *intermediate representation*. This representation is a graph, in which each node contains a code statement. The graph consists of nodes with *value* and *non-value* nodes and its edges are *flow* edges that represent control flow between program instructions, in which *value* edges are used to connect value-using nodes (e. g., operators) with value-containing nodes (e. g., operands). They are connected when there is a mutual dependency. Each node belongs to an analysis state in the *data representation*. Nodes mutate from one state into another as defined in the *transfer function*. Most transfer definitions do not change the analysis state and just compute values or are *value getters*. Any information gathered here is saved in the *data representation*. The information is not added to the analysis state, so succeeding nodes do not know anything regarding this data.

2. To build the intermediate representation, an entry script is built for the program to be analyzed and then it is gradually analyzed. It processes caller nodes, e. g., functions, methods, and constructors, that are expanded on the go, following the control flow. When a caller node is evaluated, the analysis state that is known up until that point is used to proceed with. Than *extensions* follow, that handle actual to formal parameter binding

and on returning to the calling method, *extension-sinks* nodes are placed.

3. To begin with the analysis, a *declaration analysis* is performed in which a *declaration state* is built. A *declaration state* is a set of classes, functions, constants and operators.

4. Thereafter a *heap analysis* is done, in which arrays, array indices, object fields and variables are approximated. The *summary* heap identifiers summarizes all heap identifiers that could be updated by assignments that have *statically* no information attached. When heap identifiers need to be made distinguishable, e. g., when a previously statically unknown target is statically known after processing a statement, a new heap identifier is created and all states after this use this new so-called materialized heap identifier. As heap identifiers are tracked by the value analysis, this forms an interplay with the heap analysis, in which updates have to be sent back and forth.

5. This *value analysis* consists of a first and second phase, where the first phase uses values that compute accessed control flow and structures and the second phase deals with heap identifiers. The first phase is therefore independent of the *heap analysis*. To make the height of the lattice finite and guarantee termination of analysis, the size of all sets is limited by a constant.

For the *transfer functions* both *strong* and *weak* updates are defined, that respectively update a heap identifier to a new value, or update the heap identifier to either contain the new value, or the original one.

To make the *taint analysis* work, the WeVerca framework and its results are used to follow tainted values from sources, e. g., user input, to sinks, e. g., print statements. The researchers do not exactly show what is done here, but show in their short evaluation that their method works, albeit only based on a comparison on just 2 projects. More on this is found in Section 2.5.1.1.

2.2.3.1 *Conclusion*

WeVerca shows a flexible approach on performing a static analysis, by splitting the process in an abstract analysis and a concrete analysis. It is hard to say whether the flexibility turns any end-user analysis into a simple plug and play solution as suggested. Next, WeVerca seems to cover any feature you can wish for, except for being flow-sensitive (refer to Section 2.1 for a full list of type checker features). WeVerca points out that this results in some extra false positives, albeit at a low rate, especially with regards to the examples they show. Similar to Phantm, no support is mentioned for aliases and `eval`'ed code, but this is still not much of an issue.

2.3 PYTHON

Python is a general purpose dynamically typed language, that is known to be very expressive. This means that most problems one can think of, are in just a few lines of Python code solved. Python tries to be a language that is applicable in functional, procedural and object-oriented fields. Python is both used in desktop development as for web development. According to the IEEE, it is the first top programming language [7].

2.3.1 *RPython*

RPython [1] (*Restricted Python*) tries to be a more robust and interoperable alternative to Python while preserving the flexibility the Python language brings. RPython tries to be interoperable by focussing on being compatible with the Java and .NET run-times. This is a whole different approach of doing a purely static analysis on a dynamically typed language. It can nevertheless give good insights in what one can do to take advantage of a more static typed dynamic language. RPython is a strict subset of the Python language, therefore any RPython program is also a valid Python program. RPython forces some restrictions to make the transformation to a statically typed language doable:

1. Python is a dynamic language in which type information is bound to objects, not to methods, variables or return values. RPython forbids that this type information results in incompatible types, e. g., a method must always return a value of the same type. In practice, they found this not much of a hurdle, as most Python code already adheres to this.

2. Class definitions may not be altered dynamically by adding or removing methods and fields. According to the researchers, this is a serious limitation, though special care is taken to make typical Python patterns still possible, without exactly explaining what this means.

3. Instead of being dynamically typed, only predefined primitive types can be used, like integers, booleans, and strings, together with container types like tuples, lists, and dictionaries. In user-defined classes, it is not necessary to explicitly define types, as they are automatically inferred. For example, the following is supported:

```
1 class Example
2   def __init__(self, arg1, arg2)
3     self.var1 = arg1
4     self.var2 = arg2
```

```
5
6   def run
7     example = Example("String", 35)
```

In this example, it is derived automatically that `var1` contains a string type, and `var2` a number.

4. RPython only supports single inheritance, whereas Python supports multiple inheritances. To compensate, they support mixins, which can be seen as classes marked as `mixin` and that get inlined when invoked. Mixins do not interfere with the inheritance hierarchy and methods defined in classes take precedence over mixed in methods. The order is relevant in this.

5. Just as in Python both *classes* and *methods* are treated as *first-order citizens*, e. g., they can be passed around when invoking methods.

Compiling and executing an RPython program is done in an atypical way by not parsing the source code alone, but by:

1. *Initialization*, the set-up process in which Python dynamic features can extensively be used. It is this phase in which normal Python can be used together with all Pythonic patterns that exist. For instance, this makes it possible to do metaprogramming and evaluate dynamically created Python code.

2. *Translation*, the process in which an initialized program is analyzed. During this process, types are inferred and stored, and types are checked to be not contradictory. After this, compiled programs are generated, usable for the Java or .NET run-time environment.

3. *Run*, running the output of the translation phase.

Note that a lot of Python powerful dynamic features are nowadays also possible within the Java or .NET run-time. However, as Python was built with this powerful expressiveness in mind, it performs far better at it.

Currently, because the main entry point needs to be supplied and depending on that a class or function can react differently, RPython does not support type checking in a composable manner. For the same reason, it is hard to have separate compilation. Also, generic structures[8] are not supported, though the authors think that this can improve RPython expressive power in the future.

8 https://en.wikipedia.org/wiki/Generic_programming, accessed on 30 January 2018

2.3.1.1 *Conclusion*

RPython is a whole different solution to mix dynamically typed languages with static types. They focus on making a statically typed language, as a derivative from a popular dynamically typed one.

This approach has multiple advantages, such as letting running project code on multiple run-time environments, to see which one performs and adapts best to the specific tasks. This is all possible while keeping commonly used Pythonic patterns intact. Also, the choice to have an initialization phase that allows full dynamic Python is considered a good compromise between the expressiveness this results in, versus the difficulty, this brings dynamic programming to do type checking. Disadvantages are that an existing code base can be hard to transform and that developers that use forbidden patterns extensively, will have to adapt their code style and cannot use their favorite Python packages at all time.

2.4 RUBY

See Section 1.2 for a brief introduction to Ruby, a dynamically typed language.

2.4.1 *DRuby*

2.4.1.1 *Overview*

DRuby [4] aims to bring static typing to the dynamic typed Ruby language. It does this by trying to make programmer's life as easy as possible, as is common in the Ruby community. In principle, static type inference is automatically done wherever possible. However, when this results in imprecise results, it is possible to add annotations to give static types to dynamic code. These *annotations* then are also validated at runtime. To make the result useful with not too many false positives, the developers have carefully considered to what extent the analysis should be strict. Therefore, they tolerate some lack of precision resulting in certain programs being marked erroneously as valid or invalid. In particular, there is support for doing a flow-sensitive analysis on local variables, thus reuse of local variables that are first typed `Array` and later typed as `String` is supported. On the contrary, Ruby's metaprogramming capabilities are not fully supported as its behavior is hard to grasp correctly.

We now summarize DRuby's main features:

- In Ruby, everything is an object. This is done internally in C, which hides the actual internal type from Ruby source code. Therefore, it is necessary to annotate built-in classes and their

methods with type definitions, consisting of the method name, its input type and its returning type. DRuby provides this.

- Next to the basic types, also **intersection types** are created, which are methods that can belong to multiple classes and depending on the class they belong to, can even have different returning types. It is noted that automated type inference for intersection types is not working, although annotations work fine. See Listing 7 for an example.

```
1 'a'.include?('a') # include?: (string) -> boolean
2 'a'.include?(1) # include?: (fixnum) -> boolean
```

Listing 7: Code example of intersection types

- When having a method that belongs possibly to multiple classes, a **union type**, dual to the intersection type, is formed. See Listing 8 for an example. A crucial difference between the intersection and union type is that in case of an intersection type a method must be defined with equivalent types in both intersecting class types, whereas for a union type only one of them must match the type. Both types exist to perform static type checking.

```
1 ['a'].concat(['b']) # ['a', 'b']
2 'a'.concat('b') # 'ab'
3
4 chance = Random.new.rand > 0.5 # Random.new.rand gives a
  ↪ float between 0 and 1
5 x = chance ? ['a'] : 'a' # x and y are either both
6 y = chance ? ['b'] : 'b' # arrays or strings and therefore
  ↪ concatable
7 x.concat(y) # either ['a', 'b'] or 'ab'
```

Listing 8: Code example of `concat` as union type

- A common idiom in Ruby is the use of *optional arguments* and *varargs*, a varying number of arguments. Both are supported in DRuby.
- *Parametric polymorphism* is supported to make it simpler to express certain patterns, e.g. the `Object.clone` method that returns its own type and referring to the identity object `self`.
- As *mixins* cannot be checked statically, run-time constraints are added to make sure that any contract resulting from these mixins is checked.

- In case a method returns a tuple with various types, a *tuple type* is used.
- When methods take a parameter list as input and return that list as output, a special notation is used, promoting it to a *first-class citizen* in DRuby.
- *Constants* are resolved statically and used to construct the class hierarchy.

When performing *type inference*, a structure called *object type* is used to do bookkeeping on the collection of methods and their type definition. A *constraint-based analysis* is conducted. In the first stage, all mutual dependencies are obtained and turned into constraints. This set of constraints is then refined by applying rewriting rules repetitively. On any inconsistencies detected, e. g., when looking for methods on classes (taking superclasses and mixins into account) that turn out to be undefined, errors are raised. When no errors occur, a valid typing has been found. Tuple types are treated as arrays.

If union types occur on the right-hand side, for example $type_1 \leq type_2$ or $type_3$, then this resolves correctly when $type_1$ is equal to either $type_2$ or $type_3$. For intersection types a similar approach exists. For this, run-time checks are added.

To verify any *manually annotated* method, its parameters and return values must be checked on run-time. At this moment, many structures are supported. However, making it feature complete remains future work.

DRuby can be used as a drop-in replacement for Ruby, that can be enabled to verify these run-time checks. A file provided by DRuby that contains all base types is sideloaded.

2.4.1.2 *Conclusion*

Like what is common in the Ruby community, DRuby is all about taking care of what is typical. It takes care of the most used features and focuses on creating an analyzer that is usable in general. Also, the fact that it tries to resolve most inference itself, but is given the opportunity to give annotations is very much Ruby-like. They note that adding an initialization phase like RPython does, can overcome bootstrapping issues that are the result of configuration and environment. DRuby lacks tests on real-world sized applications, so it is unknown how their solution would perform well when it is given such a code base.

2.5 COMPARING PAST RESEARCH

In the next sections all presented solutions are compared with each other. Almost every study performed an experiment in which they

show their stronger and weaker points, or even compare themselves to other similar solutions.

2.5.1 *Experimental benchmark results*

2.5.1.1 *WeVerca*

WeVerca shows in their own report that in comparison to Pixy and Phantm they performs significantly better. For the small excerpt (648 LOC), taken from the MYBLOGGIE[9] application, WeVerca finds all errors (according to themselves), where Pixy finds only 69% of the errors and Phantm only 23%. Also, with regards to the false-positives rate, WeVerca scores better, with only a rate of 19%, where Pixy and Phantm score 44% and 93% respectively. They conclude that Phantm cannot be considered useful, given that competing programs have higher false-positive rates. When looking at the running time however, WeVerca is significantly slower, especially on larger projects. We conclude that precision comes atthe price of time.

2.5.1.2 *Pixy*

Pixy shows in their report that they have found on modules (9k LOC each) of the `PhpNuke`[10] project a significant amount (24) known issues with about as many false positives (30). On other smaller projects also vulnerabilities are found, with one vulnerability per false positive as well. Pixy also found some previously unknown vulnerabilities, with about the same false positive rate. Nothing is reported on run time, nor is there a comparison with other projects on the same code base.

2.5.1.3 *Phantm*

Phantm shows in their report that they find in the `WebMail`[11] project (4k LOC) 43 problems, ranging from bugs to annotation errors, with about 11 seconds runtime. Nothing is said about false positives. Larger projects, like DokuWiki[12] (31k LOC) take 244s of run time. For this, 76 errors are reported. They consider this a reasonable running time. Using state information, gathered after the initial boot phase, 12% of all errors are dropped for DokuWiki and 12% of all errorsfor WebMail. For the former, a few methods are highlighted, in which the error reduction is over 50% of the cases. Increased runtime because of the state information was about one second according to the report.

9 https://sourceforge.net/projects/mybloggie/, accessed on January 14th, 2018
10 https://www.phpnuke.org/, accessed on 15th January, 2018
11 This project is not publicly disclosed by the authors of Phantm
12 https://www.dokuwiki.org, accessed on 15th January, 2018

2.5.1.4 *RPython*

RPython writers' claim that it is hard to say something about performance with regards to benchmarks they ran. Besides that, they state RPython performs on a scale between complete statically typed languages like C# and Java, and complete dynamically typed languages like IronPython and Jython. This means that a performance gain on Python can be achieved by implementing RPython, while Python features are partly retained.

2.5.1.5 *DRuby*

DRuby was run on 18 benchmark Ruby programs, with all programs under a thousand lines of code. In these small programs, it turned out that the number of errors found in comparison to the number of false positives is quite high. In most cases the amount of false positives is as high or even higher than the error count. The run time of most benchmarks was below 7 seconds, which is reasonably quick, albeit that all benchmarks were performed on a small code base. Some warnings were given on a code inspection level. This was in particular for unused variables and when omitting parenthesis. Both could lead to unintended code behaviour and were therefore warned for.

2.5.2 *Limitations and points of improvement*

Based on experiences by the researchers themselves, the following list of limitations and points of improvement was composed.

- *WeVerca*
  1. WeVerca is imprecise with regards to the definitions for *built-in functions*.
  2. Current analysis is done *path-insensitively*, which would preferably be path-sensitive.
  3. Due to the assumption of a *non-relational value domain*[13], false positives occur. When values are assumed to relate, it is easier to come to a more tightened bound.
- *Pixy*
  1. Pixy does not support *object-oriented features*, which is nowadays a very common pattern when developing software.
  2. Pixy handles *reference assignments* only partially. When dealing with simple variables it expects that reference parameters are not redirected to other variables inside the function.

13 Values in a *relational* value domain are values that relate to each other, thus for example for the *age* domain values between 0 and 150 will be viable alternatives.

3. `include` statements are not scanned automatically, as the paths to be included are not necessarily known in static analysis, e.g. when their paths are evaluated dynamically.
4. *Input sanitization errors are reported sometimes falsely*, e.g. when the sanitized value itself is encapsulated in a safe environment.

- *Phantm* does not report on any defects that their program has. It does not show whether false positives occur, making it harder to compare it to others. Also, nothing is said about the quality of the given warnings. Based on the results of WeVerca we can conclude that the amount of false positives is large in one example with a ratio of 93%.

- *RPython*
  1. *Independent compilation* is not supported yet. At this moment a known entry point is needed to compile a RPython program. This gives probably less accurate results than a full static analysis, as paths are likely to differ through a program depending on their origin.
  2. *Interoperability* with regards to mixing C# or Java and RPython programs is not completed yet. This is partially done for RPython accessing C# or Java, but not the other way around.
  3. Instead of making a method copy for every type that is found, *expressive power* can be increased by making methods generic and as a result more compact.

- *DRuby*
  1. Some *standard library functions are not supported* in DRuby due to their nature. This is for example the case for `Array.flatten`, for which no finite intersection type can be defined, as any *n-dimensional* array is transformed into a *1-dimensional* array.
  2. Special Ruby language behaviour makes it hard to cover all its behaviour during static analysis. This is a problem when classes are re-opened at run-time, which can lead to changed or removal of methods, the so-called *monkey patching*. *DRuby assumes that no dynamic changes are made*.
  3. It is possible to give objects more methods than the class describes by adding at run-time methods to the so-called *eigenclass* of an object. DRuby is unable to handle this behaviour.
  4. Ruby supports *reflection and dynamic evaluation* that makes it possible to metaprogram in Ruby. This is not supported

by DRuby. It is suggested that an approach as taken by RPython by precomputing this metaprogramming behaviour would be a solution, but is left as future work. During evaluation this was done manually as workaround.

5. Simpler *types* are now checked on run-time, e.g. when run through a test. However, object types and individual expressions are not checked yet. Also static verification of these annotations is seen as future work.
6. There is *no support for dynamically composed file names*, e.g. when including such file names. During experiments this was solved by replacing filenames by hand. Also, as the checks are only run when methods are actually invoked, it was necessary to build a custom script that invoked all test methods that depend on the assumption that that was done by the test runner.
7. At this moment *only local variables are analyzed flow-sensitive*.

2.5.3 *Feature comparison*

A feature comparison table is made to compare all features and can be found in Table 2.1.

Table 2.1: Comparing features of type checker on dynamically typed languages

feature	WeVerca	Pixy	Phantm	RPython	DRuby
flow-sensitive	✓	✓	✓	✓	✗ [1]
path-sensitive	✗	✓	✓ [2]	✓ [3]	✓ [4]
interprocedural	✓	✓	✓	✓	✓
supports object-oriented design	✓	✗	✓	✓	✓
supports evaluation patterns	✓	✗	✓ [5]	✓ [6]	✗ [7]

[1] DRuby only considers local variables flow-sensitively
[2] Phantm supports this partially as Conditional filtering, meaning that implied statements that come from control structures are used
[3] RPython forbids the use of multiple return types
[4] DRuby uses run-time checks when its uncertain about what different path may result in
[5] Phantm supports evaluation patterns partially, by being able to collect a (dynamic) program state as a starting point
[6] RPython supports evaluation patterns partially, as it has a full dynamic Python initialization phase, after which the allowed syntactic correct structures way more restricted
[7] DRuby accepts type hinting as a manual, though useful alternative

2.6 LESSONS LEARNT

Based on own experiences and what others have researched, we have compiled a list of properties we would like our code analyzer to have, as can be found in Section 1.1.3. The common pitfall for all solutions already researched is that they focus on a solution for general purpose, resulting in a static analyzer with limited usability. This due to of the unlimited possibilities of programming languages.

By limiting our scope to Ruby applications used in cooperation with commonly used gems, like Rails, we think that we have found a compromise with wide usability, but gives better and faster results with regard to the limitations named above.

Part II

IMPLEMENTING ANALIST

In this part, it is explained how Analist's was designed on a high level and implemented in detail. We also show how a plugin for the Atom editor was developed.

3 ABSTRACT OVERVIEW OF ANALIST

In this chapter an abstract overview of Analist is given. Section 3.1 explains where the name comes from. Next, the program flow of Analist is explained in Section 3.2, from preparation to checking. In Chapter 4 the technical details are given.

3.1 NAMING AND LOGO

The tool that is built during this research is named Analist. Its name is a derivative of the English word for analyst, or simply the Dutch word analist with the same meaning. The logo is a flask in which a red liquid bottles, indicating the experimental state of Analist. It is designed by Freepik[1].

3.2 PROGRAM FLOW

The program takes a Rails or Ruby project folder as input and transforms it multiple times, extracting and adding type information during these passes, ultimately returning in errors and warnings to display to a developer. See Figure 3.1 for an overview.

3.2.1 *Preparation*

First, some preparation is done to make the annotating and checking pass easier to perform. During the *initialization*, the project-specific configuration for Analist is read and applied, giving the possibility to declare global types and exclude files.

Next, when *exploring* the project, all Ruby files are transformed into a recursively defined manipulable and *explorable structure*, the Abstract Syntax Tree (AST). Where applicable, this structure is extended with Ruby code originating from HTML template files that are implicitly loaded by Rails. This is necessary, as it is also possible to execute Ruby code from HTML template files.

Thereafter an *indexing* pass is performed in which both *classes* and *method* mappings from the identifier to the corresponding AST are derived from project code and database schema. All mappings are put into a single *resources* object. This object is used for lookups during the annotating pass. Note that all maps are based on the project code

1 https://www.freepik.com/free-vector/scientist-boy-with-lab-objects_849316.htm

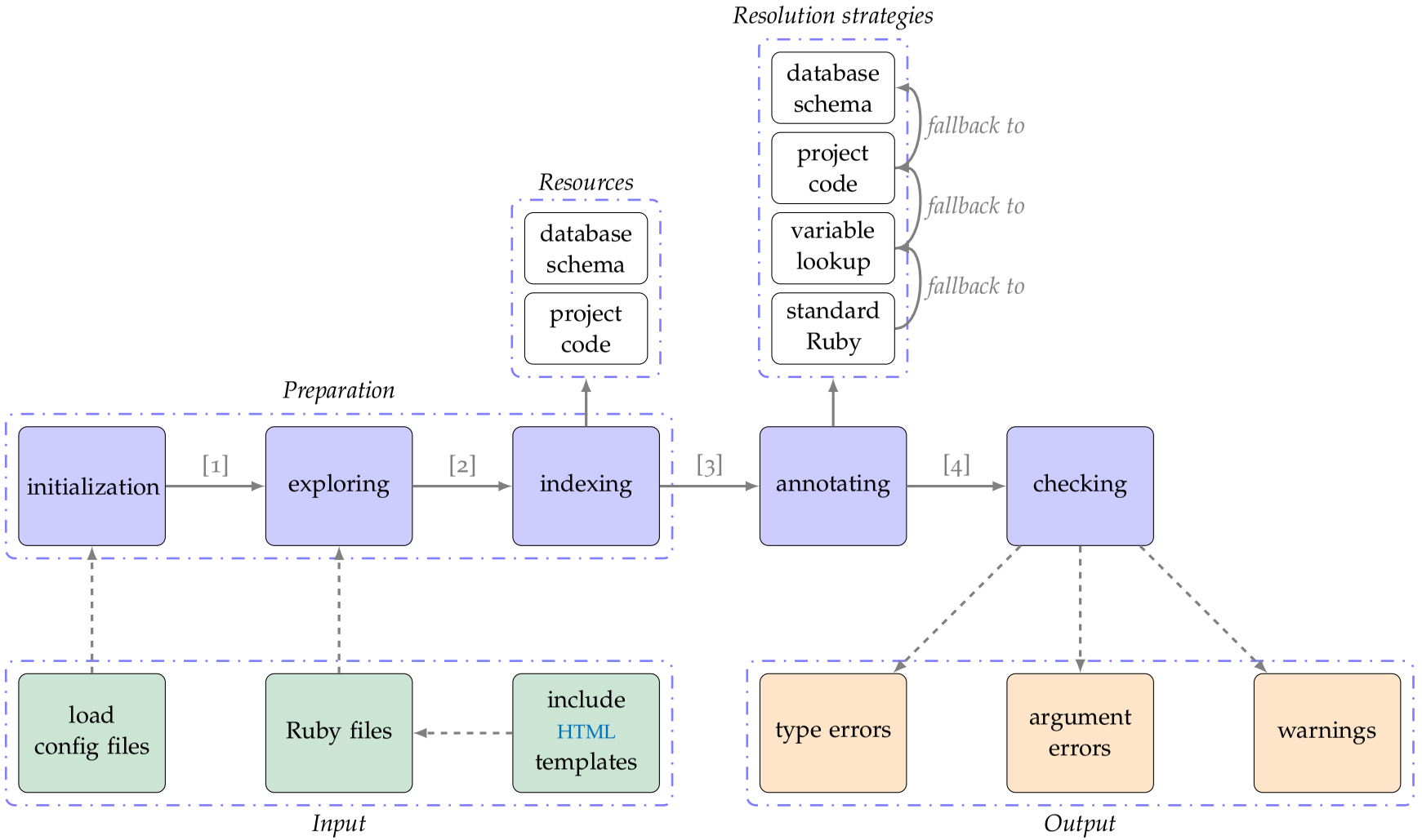


Figure 3.1: Flow chart for Analist
Legend: [1]: filenames, [2]: extended ASTs, [3]: extended ASTs with resources, [4]: annotated ASTs

and not auto loaded[2], making this approach work even when not all Rails conventions are followed.

3.2.2 *Annotating*

The most important pass of ANALIST is the *annotating* pass. During this, any *node* in the AST is (recursively) transformed into an *annotated node*. That annotated node signals what types are *expected* based on a statement, possibly with *multiple* valid alternatives. In the next pass it is compared to its actual annotation.

3.2.2.1 *Resolution strategies*

For most types of nodes, e. g., primitive types, blocks or variable assignments the annotating strategy is defined in code. Variable assignments are saved into a symbol table as a tuple of a variable and the corresponding annotation for later reference. As developers can create their own methods and classes, there is a need to handle method and object calls more carefully. There are multiple resolving strategies to find an appropriate annotation format for a node. In resolution order:

1. *Standard Ruby*, use a pre-defined annotation as supplied with ANALIST, based on the standard library of Ruby.
2. *Variable lookup*, copy the annotation from the symbol table if the annotation is known already.
3. *Project code*, try to obtain a type based on the class and method that is called and retrieve it using the *resources* object. This method is resolved on its own and its returning type is used.
4. *Database schema*, try to obtain a type based on the schema that is used by Rails.

Note that that support for *variable assignments* is available and *(nested) scopes* are taken into account. Also *object initialization* and *method calls on objects* are supported. Therefore ANALIST supports both *flow-sensitive* and *interprocedural* analysis. Furthermore ANALIST supports *object-oriented design* in the sense of Section 2.5.3. However, it is *not* supporting *path-sensitive* analysis, neither is there support for *evaluation patterns*.

3.2.2.2 *Unknown types*

In case a node cannot be resolved using the strategy above, it is annotated with a special unknown annotation type. This is used in the checking pass to ignore that node when presenting any errors.

2 Auto loading within Ruby means loading files only when asked for it, often used for keeping loading time small

3.2.2.3 *Annotation format*

Annotations are stored as a tuple, consisting of three parts. All three parts contain *signatures*, a tuple of a class type and an indicator whether they refer to an instance or a collection.

- *receiver type*, the signature of the receiving object. For example, when looking at `puts "lorem"`, the receiver will be **`nil`**, as `puts` is called on no object. In case of `User.first`, `User` will be the receiver, as `first` is called on it.
- *argument types*, the signature of the arguments. When multiple argument types are valid, they can be given as a set containing all possible valid alternatives. It is possible to give a special *any arguments* type, when the argument types can vary much. This makes it possible to deal with methods that support many types of input.
- *return type*, the signature that is returned after the statement is executed.

Note that both argument types and return type can depend on the *receiver*. Therefore, it is possible to use the receiver type and give different argument types and return type depending on the receiver. This can be compared to the *intersection type* that is defined in DRuby, see Section 2.4.1.1.

3.2.3 *Checking*

During the *checking* pass, all annotated nodes are recursively checked if their annotations comply with the derivative of the nested children's annotations. Any mismatch, when not marked to be ignored, is presented to the developer. An example of how this comparison is done can be found in Figure 3.2.

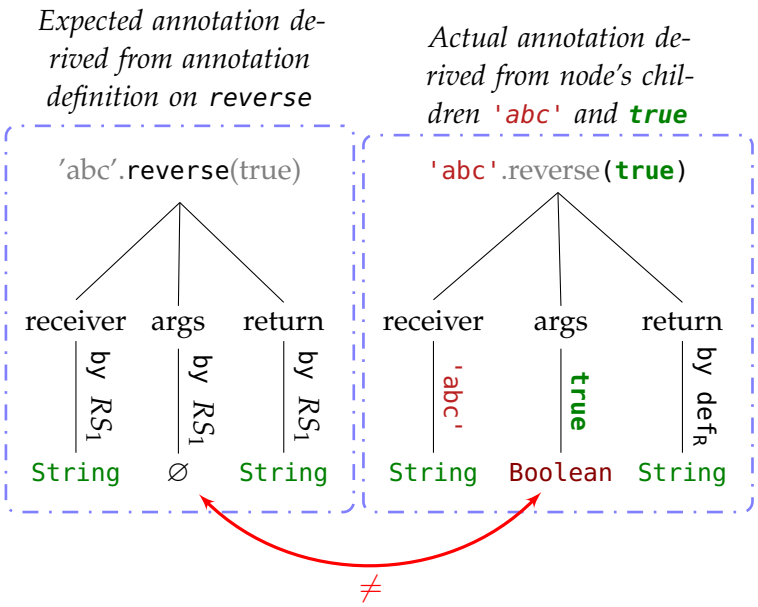


Figure 3.2: Expected versus actual annotations on `'abc'.reverse(true)`
- RS_1: Resolution strategy 1 as described in Section 3.2.2.1
- $\texttt{def}_R$: the actual return type is only for symmetrical reasons defined to be equal to the expected return type.

4 IMPLEMENTATION OF ANALIST

In this chapter it is explained how Analist was actually developed. Section 4.1 starts with the choice of the programming language Analist is built in. Thereafter, when the language is chosen, Section 4.2 explains how every pass as described in Chapter 3 is implemented.

4.1 CHOOSING A PROGRAMMING LANGUAGE

Implementing Analist starts with the choice for a programming language. Two languages are compared to this end. Ruby, which is the language that is analyzed, and OCaml a language that is commonly used for static analyzes.

To make a decision based on hands-on experience, multiple example programs were crafted to perform type checking. The bare minimum was put together to check for both type and argument errors. The examples themselves are simple to quickly build a proof of concept for. Nevertheless, they give a good feeling whether the tested approach is sufficiently and extendable enough.

4.1.1 *Proof of concept*

```
1 100 + 'word' # TypeError: String can't be coerced into
  ↪ Integer
```

Listing 9: Type error example. *A number cannot be concatenated with a string without any proper conversion, thus this line of code results in a type coercing error.*

```
1 def method(arg1)
2   # ...
3 end
4
5 method(arg1, arg2) # ArgumentError: wrong number of
  ↪ arguments (given 2, expected 1)
```

Listing 10: Argument error example. *A method invoked in Ruby has to have as many arguments as its method definition tells, thus the method call above results in a argument error.*

4.1.2 *Requirements*

Beforehand, some requirements were put with regards to developing the proof of concept. For parsing a language, *an existing parser* has to be used. This research does not focus on parsing Ruby and because it is a complex language, it would simply cost too much time to build one. Next, it is *easy for developers to run the type checker*, moreover, it should be *easy for other developers to extend* the type checker.

4.1.3 *Ruby*

Ruby was chosen as the possible language of choice as it is the language we are building a static analyzer for and it has a large development toolset, making development way more convenient. However, it is also known to be a slow performing language in the past, while some say this is no longer significant from Ruby 2.0 and onwards, especially when comparing it to for example Python or Node.js [2][8].

4.1.3.1 *Parsers*

For Ruby, multiple parsers exist. For example, Parser[1], `ruby_parser`[2], Ruby's built-in Ripper[3] and others were developed. We choose Parser as they announce themselves to be the best compatible with the Ruby language, even over several Ruby versions. They claim that a lot of the existing Ruby gems were processable by Parser, which is on its own a good benchmark proving its parsing capacities. Moreover, it is sponsored, yet an open-source project and currently actively maintained. It is tested for a 100% on grammars and it is used in popular gems[4], like a popular community-driven Ruby A plugin that gives feedback about the code that is written, commonly inline and standalone (linter), called Rubocop. Altogether, this makes it a reliable choice at this moment, but also for the future.

4.1.4 *OCaml*

OCaml is a typed and functional language. Both parsing Ruby and type checking are typically tasks that do not require state and can be performed in a functional environment. In contrast to Haskell, it is not completely strictly functional, making it more suitable for real-world applications compared to Haskell. It was chosen as possible language as multiple similar code analysis projects were built using

1 https://github.com/whitequark/parser
2 https://github.com/seattlerb/ruby_parser
3 https://ruby-doc.org/stdlib-2.4.2/libdoc/ripper/rdoc/Ripper.html
4 All gems that depend on Parser can be found in https://rubygems.org/gems/parser/reverse_dependencies

OCaml. For example, DRuby (see Section 2.4.1) and Flow[5], a typed Javascript dialect developed by Facebook, were both developed using OCaml. OCaml is known to be a fast language, as its programs are compiled and contain a very strong sense of type inferencing, resulting in more correct programs by design.

For OCaml no maintained Ruby parser projects exist. Within the `rubytt`[6] project some developers have tried to parse Ruby using OCaml. Yet actually, it lets Ruby do the parsing job (by using the built-in Ripper parser) and then transforms, also in Ruby, the parsed Ruby into JSON. From there on, OCaml reads the JSON objects and turns them into OCaml structures. A large test suite is available to validate correct behavior on many Ruby structures. Using the `rubytt` project as a start, it could be possible to perform a static analysis on the examples above.

4.1.5 *Comparison and evaluation*

After building the proofs of concept for both Ruby and OCaml, we favor Ruby over OCaml. This is mainly due to the fact that it is far better documented, has better development tools and is more commonly used than OCaml is. This makes converting ideas into code a lot more pleasant to do. For example, inspecting what is within a self-constructed object is not supported by OCaml itself [10]. Although some external development packages exist for this purpose, it remains hard to inspect any value. Documentation is according to Kevingray et al. [11] hard to read, on which we agree. This is mainly because (simple) examples for standard libraries are absent, and most of the time only type definitions are available.

In comparison to the Ruby development environment, there is also a lack of version control. For any dependency used in OCaml development, it is currently impossible to lock its version in a reliable way. This was already a problem when trying to load `rubytt` as the latest OCaml version available is already not compatible with all specific packages `rubytt` uses. For Ruby the Bundler[7] gem can be used to lock every gem to a specific version which circumvents dependency problems.

Also, it appeared difficult, even impossible, to encapsulate the Ruby parser for OCaml from `rubytt` as a module, to make it not pollute the global namespace. Also, from a more practical perspective, it would be better for the future development of the static analyzer to write it in a well-known language, close to the language that is analyzed.

Altogether we conclude that the development of a Ruby analyzer is preferably done in Ruby.

---

5 https://flow.org/

6 https://github.com/chenyukang/rubytt, accessed on September 14th, 2017

7 https://bundler.io/, accessed on January 14th, 2018

4.2 IMPLEMENTATION OF ANALIST IN RUBY

In this section the actual implementation of the gem ANALIST, as discussed in Section 3 is explained. The code base for ANALIST is freely available as open-source project[8] licensed with MIT license. The application is tested with RSpec, a test coverage of 92%[9] is reached.

4.2.1 *Code designing for* ANALIST

The code is designed stateless to be as possible, effectively by transforming the AST in each pass. The only exception to this design pattern is the use of the symbol table to keep track of variables, which is by design stateful. The stateless approach comes with multiple advantages: the AST is defined in a recursive way, for which it is very natural to use a stateless approach. Next, it is possible to parallelize execution in the future. By the modular nature of this design, adding actions is quite easy. The latter was already experienced during development since ANALIST started with only simple passes. The number of passes expanded over time.

The tool that ANALIST is modeled after is Rubocop, a linter, that checks for code style and other structures that are considered bad practice and warns about them. It is widely used for guarding code quality within gems and other Ruby projects.

In particular, the parts that handle configuration loading, perform path globbing, and determine the final output format, are heavily inspired by Rubocop's (open) source code.

4.2.2 *Preparation*

This is the implementation of the preparation phase, as described in Section 3.2.1.

4.2.2.1 *Initialization*

During the start of ANALIST's type checking all configuration options are read from `.analist.yml`:

- Exclude files - when given a globbing expression[10], it will ignore all matching files during ANALIST's run. This comes useful when only a few files give false positives, e. g., when they highly interfere with Ruby's standard library. All other Ruby

8 https://github.com/tcoenraad/analist

9 Analyzed using `simplecov` (https://github.com/colszowka/simplecov) on commit `8cfc5bb` of ANALIST

10 https://en.wikipedia.org/wiki/Glob_(programming), accessed on 15th January, 2018

files within the analyzed folder and not excluded by this file are added automatically.

- Define global variable return types - for global variables, it is possible to set the return type of those manually. Ruby is sometimes inimitable, with respect to where variables actually originate from. Defining those variables manually helps ANALIST to resolve their type, instead of giving up.

4.2.2.2 *Exploring*

At first Ruby code is parsed and normalized by putting it into an AST. The `parser` gem is used for mapping Ruby (`*.rb`) source code files into ASTs. An example of an AST can be found in Listing 4.1. Reasons for picking `parser` over its competitors can be found in Section 4.1.3.1.

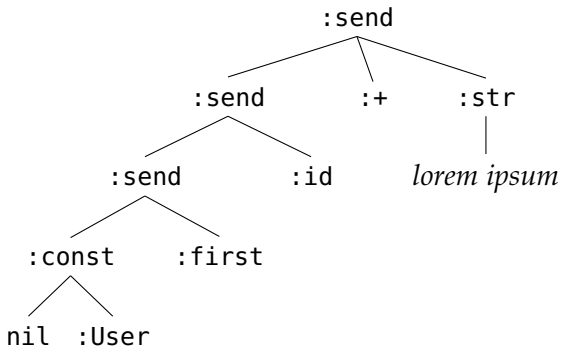


Figure 4.1: AST example for `User.first.id + 'lorem ipsum'`

The parsing by Parser is *flow-sensitive*, meaning that the order of statements is important and (simple) history is taken into account. A statement will not always give the same interpretation. In the examples below, we focus on the (last) `var` statement.

Assign, then refer

```
def method
  var = 35
  var
end
```

When preceded by a local variable assignment, the representing expression looks like `(:lvar, :var)`, being a *lookup reference* to a local variable.

Refer only

```
def method
  var
end
```

When the expression does not contain a preceding local assignment, the representing expression looks like `(:send, nil, :var)`, which represents a method call without arguments.

During this pass, all ASTs are marked with the filename they originate from. This makes it possible to have error messages with a source location. At the same time, it inlines Embedded Ruby (ERB) template files, where applicable. For example, when a Rails controller has an `edit` action, the corresponding `edit.html.erb` template is filtered for Ruby source and added. This ensures that Ruby source code that runs as part of the template rendering process is also type checked within the correct context.

4.2.2.3 *Indexing*

During the *indexing pass* all ASTs from the *parsing pass* are filtered. Any class and method definition found is added to a look-up table. This table consists of class definitions, including superclasses and method definitions, with their namespace and corresponding indexed node. External dependencies like gems are not taken into account in this step. An example of a look-up table is found in Figure 4.3.

```ruby
class User < ActiveRecord::Base
  def full_name
    "#{first_name} #{last_name}"
  end
end
```

Figure 4.2: `user.rb`

```
classes: {
  User: {
    scope: [], super_class: 'ActiveRecord::Base'
  }
},
methods: {
  full_name: {
    User: #<AST node for method full_name's body>
  }
}
```

Figure 4.3: Look-up table (simplified), for `user.rb`

4.2.3 *Database schema*

Independent from all passes, but before annotating in the next pass, the Rails database schema is loaded. This makes it possible to resolve object's fields that come directly from the database. Within Rails, they are never explicitly defined but are just assumed to exist when called. These calls fail when this assumption is not met.

```
CREATE TABLE users (
    id integer NOT NULL,
    email character varying,
    -- (...)
);
```

Listing 11: SQL schema

```
users: {
  id: 'int4'
}, # (...)
```

Listing 12: SQL schema (simplified) in Ruby

Both look-up table and SQL schema are used in next passes when resolving a method call. Mainly due to the existence of the classes table, it is also possible taking **inheritance** into account.

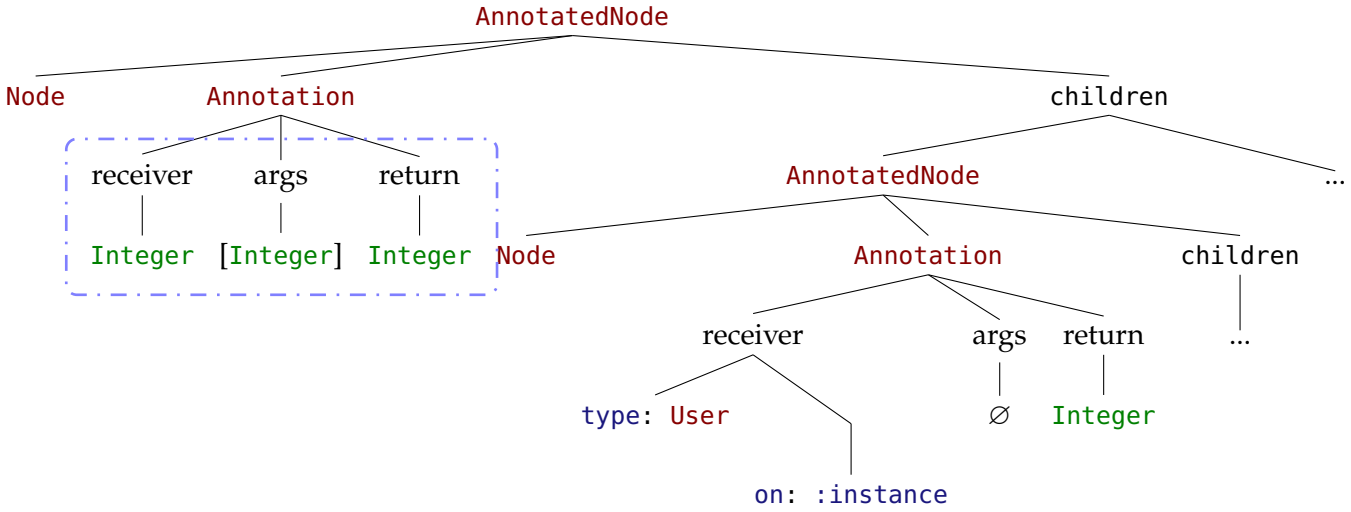
4.2.4 *Annotating*

This is the implementation of the annotating phase, as described in Section 3.2.2.

During the *annotation pass*, all ASTs from the *exploring pass* are recursively transformed into annotated nodes, while preserving the structure. Annotations are formatted as described in Section 3.2.2.3.

All information gathered (e. g., header table, SQL schema) in previous passes is used to give an inferred type as concise as possible. The strategy as discussed in Section 3.2.2.1 is followed to infer a type. When resolving a method during the *Project code* strategy, ANALIST assumes that the last statement of the method is its return type. Making this behave completely in accordance to Ruby's behavior, thus *path-sensitive*, it would take a significant part of the time available. As it is in conventional in Ruby to have only one return type, this was not considered a good investment in time.

Section 4.3 tells how the methods were chosen for which predefined annotations were programmed.



Listing 13: Tree view (simplified) of the annotated node for `User.first.id + "lorem ipsum"`. *On root-level the annotation (see dashed block) of the derived receiver type is **integer**, because the left-hand side `User.first.id` resolves (recursively) into an **integer** return type. Together with the `+` operator, it is (by definition) expected to have an a **integer** as argument type and **integer** as returning type*

4.2.4.1 *Symbol table*

During the annotation pass, local variables and their annotation are stored in the symbol table. Every time a new scope is entered a new, higher, level is used. The level is lowered after leaving a scope. This makes it possible to follow variables through a program in the correct scope. This object is also used to store globally defined variables. Directly after the initialization, those variables are stored at the root level of the symbol table.

4.2.4.2 *Additional checks*

Like mentioned in Section 4.2.1, it is easy to add modules, e. g., to perform additional checks. One check is implemented in Analist, for the Draper[11] gem.

The decorator pattern is a common programming pattern to separate model logic from viewing logic. For example, a user's address (the viewing part) is the composition of street, city and postal code (altogether, the model part). Analist supports this pattern by warning developers for missing decorators[12]. When a method call cannot be resolved by the above-mentioned resolution strategy, it looks up whether the method could be resolved when the receiving object is decorated first. When it is, it is marked as a *decorator hint* and turned into a decoration warning in the *checking pass*.

11 https://github.com/drapergem/draper

12 Analist assumes that a decorator for `Klass` is called `KlassDecorator`, like what is used in Draper

4.2.5 *Checking*

This is the implementation of the checking phase, as described in Section 3.2.3.

After the *annotation pass*, all annotated nodes are validated in the *checking pass*. These nodes are now recursively mapped into expected and actual annotations. The *expected annotation* is the annotation as found in the *annotation pass*, whereas the *actual annotation* is composed of the receiver's annotation, the annotations of the arguments and the node's return value. When annotations are marked *unknown type* or arguments *any type*, respectively the complete annotation or the arguments part are skipped when evaluating the annotation.

Expected and actual annotations are compared afterward. Because all nodes are checked in a recursive manner, any abnormality in any chain can be detected. All deviations are collected.

All collected errors are finally returned to the user in a (human) readable format.

4.3 PRE-DEFINING ANNOTATIONS

To have a good pre-defined annotation table the most used Ruby methods were collected and annotated manually. To know which methods are most used, we determined the most popular Ruby projects as GitHub[13] lists them:

1. **Ruby on Rails**, a web-application framework, see `https://github.com/rails/rails`
2. **Discourse**, open discussion platform, see `https://github.com/discourse/discourse`
3. **GitLab**, an end-to-end software development platform, see `https://github.com/gitlabhq/gitlabhq`
4. **Devise**, a flexible authentication solution for Rails, see `https://github.com/plataformatec/devise`
5. **Diaspora**, a privacy-aware, distributed, open source social network, see `https://github.com/diaspora/diaspora`

Next, we took a copy of their open-source code projects and analyzed it using CodeQuery and Starscope [14] to extract all method calls. The results were put into an SQL database using a shell script (see Listing 14) to perform queries on the gathered data. As it turns out, the most used methods in these projects are test development related

13 `https://github.com/topics/ruby`, accessed on December 4th, 2017

14 `https://github.com/ruben2020/codequery/blob/master/doc/HOWTO-LINUX.md#how-to-use-codequery-with-ruby-go-and-javascript-code`

(see Figure 4.4). As these methods most likely will be run by a continuous integration service, we filter any spec or test folder and focus on the methods left and annotate the methods that are left. Based on the percentages on the top 10 method calls in Figure 4.5, both Ruby and Rails methods need to be annotated to get a coverage of about 75% of all method calls.

The aggregated list was used to perform a manual annotation of the top 100 of all method calls, as far as possible. Ruby-Doc.org is used as a guideline for the annotations. During manually annotating this top 100 it became apparent that the method definitions sometimes lack detail or are even incomplete. E. g., one time it occurred that only within the description exceptional behavior was defined, which was only discovered by running ANALIST on a mid-size Ruby application. This makes it therefore impossible to perform auto annotating, based on the documentation. That should be improved first.

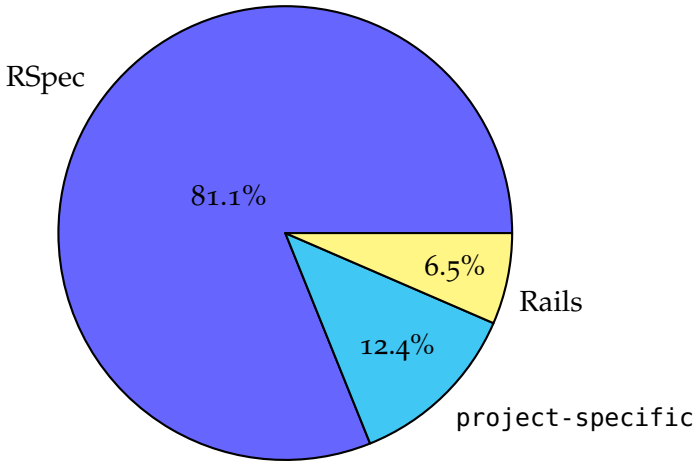


Figure 4.4: Top 10 most used Ruby method calls for the top 5 Ruby on Rails projects, based on Listing A.1. *Note that RSpec is a test framework for Ruby*

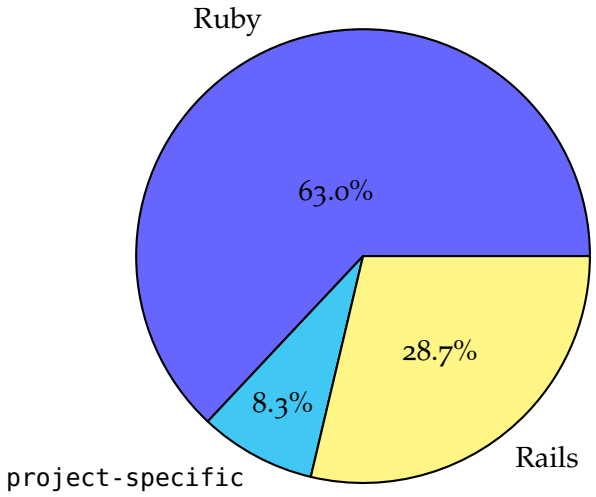


Figure 4.5: Top 10 most used Ruby method calls for the top 5 Ruby on Rails projects, without testing-related methods, based on Listing A.2

4.4 AN ATOM PLUGIN

As one of the subquestions of this research is about offering substantial benefit for developers (see Section 1.6), it became apparent that building an editor plugin could be very useful for ANALIST. One of the easiest editors to do this for is Atom[15], which is an open-source editor with extensibility in mind. Given that Atom is used by about 20% web developers [15] in 2017, a large audience that can be reached at once.

The editor plugin made for ANALIST is modelled as linter. For Atom, already a large linter framework exist, named `linter`[16]. To make a linting plugin for ANALIST for Atom, the existing `ruby -wc`[17] `linter-ruby`[18] was adapted to use ANALIST instead.

4.4.1 *Needed changes*

To achieve Atom support, a few changes had to be made to ANALIST to make it work. E. g., the output format was not machine-readable to convert into an Atom linter compatible format. Also, there was no support yet for streaming files through *standard input*, which is necessary for files that are not saved yet. It is future work to perform autoloading here, to make sure that references can be resolved, in the sense of resolution strategy *Project code* in Section 3.2.2.1.

15 https://atom.io/
16 https://atom.io/packages/linter
17 `ruby -wc` gives errors related to syntax errors, etc.
18 https://atom.io/packages/linter-ruby

4.4.2 *Show case*

Like ANALIST, `linter-analist` is open-source with MIT license. Therefore freely available on `https://github.com/tcoenraad/linter-analist` and published as listed Atom plugin[19]. An example of the linter plugin in action, can be found in Figure 4.6.

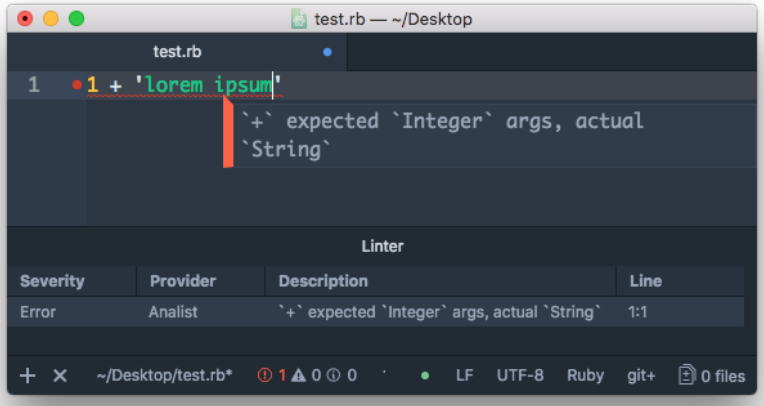


Figure 4.6: `linter-analist` in action within Atom

19 `https://atom.io/packages/linter-analist`

Part III

REVIEWING ANALIST

In this part, we validate Analist to see how it performs in both macro and micro benchmarks. At last, we look back on the research questions as stated in the introduction and give advice on future work.

5 VALIDATION

In this chapter, ANALIST is tested to see how it performs in both a macro and micro benchmark. In the macro benchmark, we focus on performance and compatibility of ANALIST with real-world code, while the micro benchmark gives a good idea how developers experience using ANALIST. After all this, the requirements as set in Chapter 1 are reviewed.

5.1 MACRO BENCHMARK

In this benchmark, the performance and compatibility of ANALIST with big real-world projects is checked. In particular, the number of changes needed for the projects for running ANALIST, and its running time are of interest. Also, a quick look is thrown upon the results to evaluate the quality of the warnings given. Altogether, this answers the main research question:

To what extent is it possible to create ANALIST*, a static type analysis tool for Ruby on type checking that conforms to the requirements as put forward in Section 1.1.3?*

To find big real-world projects, ANALIST is run on a few popular Rails projects, as listed by GitHub[1]. That list is filtered on projects that use Rails as web framework, to exclude projects that solely extend Rails.

1. **Discourse**, open discussion platform, see `https://github.com/discourse/discourse`
2. **GitLab**, an end-to-end software development platform, see `https://github.com/gitlabhq/gitlabhq`
3. **Diaspora**, a privacy-aware, distributed, open source social network, see `https://github.com/diaspora/diaspora`
4. **Spree**, an open source ecommerce solution, see `https://github.com/spree/spree`

Next, the Moneybird main application is considered. At last, ANALIST itself is tested.

1 `https://github.com/topics/rails`, accessed on 1 February 2018

5.1.1 *Needed changes*

A few changes had to be made to make ANALIST compatible with the abovementioned projects.

- *Relaxing parser warnings* to make them not fatal. The gem Parser is unable to handle ambiguous Ruby code[2] and throws an error when it finds such code. ANALIST used to re-raise this error, but for this macro benchmark, it ignores it.
- *Exclude Ruby template files manually* in `.analist.yml`. These template files do not have valid Ruby syntax before being rendered but nevertheless had a Ruby extension. Therefore, ANALIST tried to parse these files erroneous. For this macro benchmark, these files are ignored.
- Some incompatibilities with *file encoding* arose, again without appropriate error handling. For this macro benchmark, these files were just excluded.

All of these problems can easily be fixed for ANALIST. Therefore, these are put into future work.

5.1.2 *Results*

project (`commit`)	Ruby LOC	running time	warnings (unique)
Discourse (`ce507b7957`)	160K	78s	4941 (146)
GitLab (`dfe9d49534`)	367K	130s	4704 (158)
Diaspora (`3121de795`)	40K	13s	666 (42)
Spree (`d7bf8ee47`)	58K	19s	800 (53)
Moneybird (`9e1c701ed`)	168K	65s	1771 (178)
ANALIST (`3150476`)	2K	1s	0 (0)

Table 5.1: Performance of ANALIST v0.1.21.alpha

For all warnings found in Table 5.1, a look at the unique[3] warning reveals that mostly false positives are shown. As the total number of warning is much larger than those unique, fixing these warnings is probably a quick gain. These warnings can vanish by defining more concise annotations within ANALIST. For example, 1.4*K* warnings (collected on all mentioned project above) are related to the `where` method that *accepts* a `Hash` as well, whereas ANALIST *rejects this* at the moment. They would be removed all

2 An ambiguous statement is for example `Object.method *x`, which can be `Object.method(*x)` or `Object.method() * x`

3 Unique warnings have an equal type signature, but possibly different values

Positively, most annotations that give errors now there are (almost) no new features for ANALIST required, which indicates that the *proof of concept* is mostly feature complete.

A second result that stands out, is that the running time is quite linear to Ruby LOCs. It seems that parallelizing ANALIST could improve its performance, especially after the *indexing* pass is over.

5.1.3 *Threats to validity*

- Only a limited amount of projects is reviewed. Although these are popular according to GitHub, we can only assume that this represents a typical Rails project.
- Software is only reviewed after being published online. ANALIST focuses on adding value to the developer's cycle already during writing code, not after publishing it. McConnell [13] confirms that the amount of published errors is smaller than when developed.

  Moreover, Ruby code is typically tested with extensive test suites, resulting in less warnings by design. This makes it hard to compare the results above with earlier research for different programming languages.
- It is very hard to quantify the effectiveness of ANALIST. Although this is tried by tracking running time and counting warnings, it still gives only a feeling of how good the tool performs.

5.2 MICRO BENCHMARK, THE CASE STUDY

This section describes a micro benchmark in which the developer's experience is qualified. Both installation and usage experience are of interest. A small group of developers is asked to install and use ANALIST for about a week. Therefore, this section directly answers the subquestion:

To what extent do developers have benefit from using ANALIST*?*

5.2.1 *Experiment*

During the period of one week, 5 developers at Moneybird tested ANALIST during their day to day work within Atom. All (5) use Atom as their daily editor and mainly develop for Ruby. Almost half (2) of them is back-end developer, the rest (3) develops for both back- and front-end. Although the group is small, we still can get a good feeling how well ANALIST is experienced. To get extra data out of this experiment, all warning messages that ANALIST gets were put into

log files and analyzed afterward. These logs were written file change by default, but 3 participants noted that they run linters only on file save. The participants were also asked to log warnings themselves, to see to what extent they experience the same as what actually is registered.

5.2.2 *Installation*

The developers rate the installation of ANALIST at a scale of 7 rather positive: 5 (1), 6 (3) and 7 (1). They comment that a few manual adjustments were needed to make it work, but in the end, the installation boils down to 1 single command. Some issues appeared regarding earlier installed versions of ANALIST, and last-minute changes to the Atom plugin, but they were resolved within one day.

5.2.3 *Results*

After one week, the participants were asked how they experienced the use of ANALIST. The raw results can be found in Table A.3 and Table A.4.

5.2.3.1 *Caveats*

One developer experienced unpredictable behavior of ANALIST after one day. It looks like an isolated problem due to his working environment, although this was not completely ruled out.

The other (4) participants did not use ANALIST for the complete week, due to several reasons. One of the participants got sick for a week, another started after a few days, after being ill. Others said they were mainly assisting engineer for Moneybird's support desk and did not develop much in a way ANALIST could assist.

5.2.3.2 *Experiences*

The responses after using ANALIST for a week were promising as developers mentioned that the tool has potential and could be useful, but the results *at this moment* were disappointing. Most developers report that *only false positives* were shown. In the end, after the experiment one found a *true positive*. In the future, most expect that ANALIST could help because it validates code already before execution.

Judging by the log files, a picture emerges that is similar to what already was found in the macro benchmark and was reported by the participants of this experiment. During one week 256 warnings were logged, of which 40 were unique. Most warnings shown were false positives and, according to our belief, are simple to fix.

5.2.3.3 *Suggestions*

The developers gave two suggestions. One was to give helpful warning messages with a reference to the Ruby documentation for correct usage. The other suggestion was to give also warnings on `nil` values.

5.2.4 *Threats to validity*

- Only 5 developers used ANALIST, which is an amount that low, that some opinions must be missed.
- Most developers did not use ANALIST for the complete week, but only for a few days.
- Due to the short timeframe, there was no room for improvements to the ANALIST to see how much of difference improvements could have.
- It is hard to tell whether ANALIST should have come into action during the run-time of this experiment or that the errors developers made were of a different kind, e. g., errors in business logic.

5.3 COMPARISON WITH EARLIER RESEARCH

It is hard to compare the validation results for ANALIST with earlier research, due to the fact that the programming language analyzed differ, projects checked were written in a different language and the projects used to compare with were less mature and also significantly smaller in LOC. Some earlier research gives true positives, which is something we did not see for ANALIST in most cases. In that way, those programs performed better, as they *found errors in published projects*.

What should be noted is that within the Ruby world, it is far more common to have an extensive test suite, than in other languages. That makes it less likely that ANALIST will give true positives after a project release is published. Earlier research did not do anything about type checking *while developing*.

With regards to running time, ANALIST is much faster (in LOC/s) than Phantm, Pixy, and WeVerca, according to the results of Hauzar and Kofroň [6] and DRuby, according to the results of Furr et al. [4]. RPython did not run any benchmark that tells how fast it is per LOC and is therefore incomparable on this part.

5.4 REVIEWING REQUIREMENTS

In this section, the requirements for ANALIST are reviewed, one by one. This partially answers the main research question:

To what extent is it possible to create Analist*, a static type analysis tool for Ruby on type checking that conforms to the requirements as put forward in Section 1.1.3?*

- It *must* do a static analysis focusing on type checking for Ruby, in particular on a project that uses Rails as its framework, both for their latest released versions.
  - ✓- with a slight remark. For compatibility reasons with the majority of Ruby applications, Ruby 2.3 syntax is the default (whereas 2.5 is in February 2018 current), but Rails 5.2 is supported out of the box. It is not hard to change the used Ruby syntax version, as Parser supports all versions that still are maintained by the Ruby developers.
- It *must* show only relevant errors, thus only when it is *almost* certain that it is a programmer's mistake and will result in run-time errors.
  - ✗- unfortunately, as was found in both the macro and micro benchmark there are many warnings for false positives. It is expected that getting this right is only a matter of time. Section 5.1.2 confirms this, as about 1.4K warnings can vanish with probably one change.
- It *should* be flow-sensitive, context-sensitive and interprocedural (see Section 2.5.3 for an explanation of these terms). It was found in earlier research that this may improve results significantly.
  - both ✗ and ✓- Analist is both flow-sensitive and supporting interprocedural structures. However, support for context-sensitivity is missing. As this is highly interwoven with path-sensitivity, both should be added to make this requirement pass.
- It *should* take advantage of any supported gems that are available within the project. For instance, Moneybird uses the `mutations` gem, that enforces run-time validation for models and gives an outline for what the data model should look like, including semi-automatic coercion (implicit casting) from one value type to another. The database scheme as defined in source code tells how the object's fields are defined. Rails models tell what relations they have and can be used as an anchor for what kind of object is to be expected when referring to it.
  - ✓- gems for mutations and decorators are supported. The Rails database schema is used. What is not used yet is the defined relations within Rails models.
- It *should* be adapted to work with Rails out-of-the-box, as is common for gems that support Rails. This means that simply

adding the program and running it would be enough to give decent results with sane, yet opinionated default settings.

  - ✓- as part of the design. Also during the micro benchmark, developers confirmed that starting to use ANALIST is quickly done.

- It *should* be possible to use it within an automatic building process.

  - ✓- similar to the previous requirement. This is tested within one project of Moneybird during ANALIST's development. It was experience equal to what the developers experienced during the micro benchmark.

- It *should* be possible to configure what kind of errors and warnings are given, to make the tool as compatible to a developer's style as possible.

  - ✗- it is not possible to configure the exact type of errors a developer gets on a fine-grained scale. Already configurable is the exclusion of complete files. This configuration can simply be extended with options for the specific error type (e. g., type or argument errors) and this can even become more fine-grained by excluding specific methods.

- It *should* be fast enough to be able to run the program on a Ruby file after each save, or a shorter time frame.

  - ✓- the linter for ANALIST clearly shows that the run-time of the checker is fairly acceptable.

- It *could* preferably have a way of saving an initialized run-time state. This is suggested by DRuby and implemented in RPython. This might make it easier to deal with variables that dependent on the run-time environment

  - ✗- this is only partially implemented. It is possible to define manually global variables, but in no sense support for specific instances is available. Also, with the lack of path-sensitive analysis, this feature does not bring much profit at this moment. With renewed insight, we think that this is not a great concern within Rails projects, in which it is common to have differences between environments as small as possible.

6 CONCLUSION

In this chapter the research question and its subquestions are looked back on and answered shortly. Finally, in Section 6.2 future work is presented.

6.1 RESEARCH QUESTION

The main research question that is answered in this research is:

To what extent is it possible to create Analist, *a static type analysis tool for Ruby on type checking that conforms to the requirements as put forward in Section 1.1.3*?

The answered subquestions are:

- To what extent is there benefit from using information that is being exposed by some gems
- To what extent do developers have benefit from using Analist?

After reviewing all requirements in Section 5.4, we conclude that creating a static type for Ruby analysis tool is possible, but with serious remarks. The tool Analist is created and works for every implemented aspect correctly, according to the passing tests, with 92%[1] code coverage[2]. Moreover, the pragmatic approach of type checking seems to give performance benefits (see Section 5.1) and makes Analist much more predictable towards a developer than a type checker that performs automatic type inferring.

All types are now derived from configuration files that are part of gems, being either Rails for the database schema, `mutations` for types for user input or `draper` for decorator (see Section 4.2.4.2 for a comprehensive explanation of decorators). Without support for the gems mentioned above, a type checker could only guess by observation what the type is, which is less reliable and precise.

However, when used in the work field as described in Section 5.2, we get a view that is less positive. Many false positives with only one true positive make that the added value at this moment is too low to make use of Analist on a day-to-day basis. This is the result of the trade-off between a fully automatic typing system which adapts to any project-specific code or having an approach in which the types

1 The 8% missed coverage mostly concerns run-time initialization and some method annotations, but this does not concern the annotation patterns themselves

2 This code coverage was measured using `simplecov`, see https://github.com/colszowka/simplecov

are determined by hand. The former is what is usually done in this research area, but often with giving noisy, dubious feedback. In contrast, ANALIST follows the latter approach, in which the method's behavior is manually stored resulting in clear feedback, but with the chance that wrong annotations are crafted, and project-specific changes are missed.

Altogether, developers see the potential advantages of having a type checker like ANALIST, but they note that *at this moment* the lack of any useful feedback lowers the benefits of using one.

6.2 FUTURE WORK

In the future, there are different directions the development of ANALIST could head to. In order, what we think the most impactful changes to least impactful changes are described.

6.2.1 *Handle Parser exceptions correctly*

As mentioned in Section 5.1.1, many errors that come into play when adding ANALIST to a real-world project were due to missing exception handling. This can easily be added.

6.2.2 *Autoload files in* ANALIST

As mentioned in Section 4.4.1, to have equal results when running ANALIST in a full project and when running it on one single file through the linter plugin, it is needed to perform autoloading. This means that any project code references in that single file are loaded as well.

6.2.3 *Add more pre-defined annotations*

The list of pre-defined annotations is not complete at this moment, only the *top 100* was picked, and annotate to best effort. We think that this list can be extended within the framework as it is already available.

6.2.4 *Improve pre-defined annotations*

The list of pre-defined annotations is not always as concise as possible according to the Ruby docs. Some annotations are *at this moment* not picky on their arguments, or result in an unknown return type. This could be improved.

6.2.5 *Add more Rails and `mutations` support*

At the moment, support for both Rails and `mutations` is mainly limited to basic types. By extending support, also relations and deeper nested structures can be followed and used during annotating.

6.2.6 *Adapt to project environment*

Within Ruby, there is nothing that prevents a developer from redefining a method, extending an existing class, etc. However, ANALIST expects that only default behavior is respected. ANALIST could have a mechanism to extend and alter the functionality of its annotations to adapt its behavior within every project. An important remark is that extending a class is harmless, but redefining a method is considered bad practice.

6.2.7 *Become path-sensitive*

As mentioned in Section 2.1.3, path-sensitivity can improve the results. By better predicting the return type of a method, ANALIST can become smarter. For example, when dealing with code that uses often *early* return statements, predicting which code path is taken, makes the checking thereafter more precise.

6.2.8 *Have fine-grained exclusion*

At this moment only complete Ruby files can be excluded, e. g., when they result into false warnings. It would be better when this could be done on a fine-grained level to let ANALIST analyze and use as much code as possible.

6.2.9 *Deal with business logic errors*

ANALIST at present, will not protect the developer from programming business logic errors. These errors cannot be prevented using any type system. Usually, this is dealt with by having tests, which is very common to the Ruby ecosystem. In other languages, there are sometimes possibilities to make a formal definition that can be validated, e.g. with OpenJML[3] for Java. These definitions could also be derived from the Rails models as they also supply validations[4]. It is expected that this requires a large overhaul in ANALIST's design.

3 http://www.openjml.org/
4 http://guides.rubyonrails.org/active_record_validations.html

Part IV

APPENDIX

APPENDIX

```
#!/bin/bash
find . -iname "*.rb" > ./cscope.files
starscope --verbose -e cscope --exclude '*.js' --exclude
 ↪  '**/test/**/*' --exclude 'test/**/*' --exclude
 ↪  '**/*_test.rb' --exclude '**/spec/**/*' --exclude
 ↪  'spec/**/*' --exclude '**/specs/**/*' --exclude
 ↪  'specs/**/*' --exclude '**/*_spec.rb' --exclude
 ↪  '**/features/**/*' --exclude 'features/**/*' --exclude
 ↪  '**/*.feature'
# to analyze with specs, toggle comment with line above
# starscope -e cscope --exclude '*.js'
ctags --fields=+i -n -R -L ./cscope.files
cqmakedb -s ./myproject.db -c ./cscope.out -t ./tags -p
cqsearch -p 8 -s ./myproject.db -t '*' >
 ↪  ./ruby-method-calls.txt
iconv -f utf-8 -t utf-8 ./ruby-method-calls.txt >
 ↪  ./ruby-method-calls
psql analyze_ruby_method_calls -c "\copy
 ↪  method_calls_no_specs(method_call, source_file,
 ↪  source_statement) FROM './ruby-method-calls' CSV
 ↪  DELIMITER E'\t' QUOTE E'\b';"
psql analyze_ruby_method_calls -c "UPDATE
 ↪  method_calls_no_specs SET project='`basename "$PWD"`'
 ↪  WHERE project IS NULL;"
# to analyze with specs, toggle comment with line above
# psql analyze_ruby_method_calls -c "\copy
 ↪  method_calls_full(method_call, source_file,
 ↪  source_statement) FROM './ruby-method-calls' CSV
 ↪  DELIMITER E'\t' QUOTE E'\b';"
# psql analyze_ruby_method_calls -c "UPDATE
 ↪  method_calls_full SET project='`basename "$PWD"`' WHERE
 ↪  project IS NULL;"
```

Listing 14: Shell script for finding method calls within Rails projects using CodeQuery

method call	count	origin
to	38009	RSpec
expect	36557	RSpec
it	25977	RSpec
let	15786	RSpec
project	14238	project-specific
eq	12687	RSpec
create	12134	Rails
context	11315	RSpec
describe	9513	RSpec
user	8604	project-specific
...	...	...
total	559780	Rails top 5 projects

Table A.1: Top 10 most used Ruby method calls for the top 5 Ruby on Rails projects

method call	count	origin
new	5059	Ruby
params	3747	Rails
include	3357	Ruby
require	2483	Ruby
current_user	2129	project-specific
each	1835	Ruby
present	1796	Rails
id	1786	Rails
private	1728	Ruby
to_s	1623	Ruby
...	...	...
total	169313	Rails top 5 projects

Table A.2: Top 10 most used Ruby method calls for the top 5 Ruby on Rails projects, without testing-related methods

[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
Back-end	analist 0.1.21.alpha	6	I just had to run `bundle update` to downgrade RuboCop (I already needed a higher version of `parser` than for ANALIST)	Yes	6	Clear!	Yes!	Yes!
Back-end	analist 0.1.21.alpha	7	1 command	Yes	6	cmd+, I found a unclear command, otherwise very easy	Yes!	Yes!
Both	analist 0.1.21.alpha	6		Yes	6		Yes!	Yes!
Both	analist 0.1.21.alpha	5	On its own, installing of ANALIST was not difficult, but the environments made it a bit troublesome	Yes	5		Yes!	Yes!
Both	analist 0.1.21.alpha	6	I got `pry` as missing dependency, but that could be due to my own Ruby environment.	Yes	7	I already had an existing version of the linter, but in that time the installation was also easily done.	Yes!	Yes!

Table A.3: Questionnaire on ANALIST's installation (questions and answers are translated)
[1]: Usually I develop for: (back-end, front-end or both), [2]: What outputs do you get if you run 'analist -v'?, [3]: Altogether, how difficult or easy did you find the installation of ANALIST*? (scale 1-7), [4]: Can you explain this? [5]: Do you use Atom als daily editor?, [6]: Altogether, how difficult or easy did you find the installation of the linter for* ANALIST *in Atom? (scale 1-7), [7]: Can you explain this?, [8]: Do you see a warning of* ANALIST*?, [9]: Do you see a warning of the linter?*

[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	
3	Wednesday, I used it partially.	3	Unfortunately, Analist did not find any real errors, but it found a false positive :-(	1, 0, 1	It is nice that Analist works automatically and as a result considers everything. For me, I'm very attentive to linter warnings.		6	If it can search more extensively in files and therefore know more about the types it can enhance the flow during day-to-day programming. You are immediately alerted about possible error, even before automatic tests get to run.	
1	After this an inexplicable error occurred and the extension was disabled	1	Limited results due to the crash				4	I see a future, but I did not have a genuine experience.	
1	As support engineer I did not program much.	1	Only seen 1 false positive	Discovered 1, was a false positive	Saw only 1 false positive	Seen too little	4	Seen too little	
3	Unfortunately, due to illness I missed a week and could not test 'good' with it	3	Probably because of little use also found less 'errors'. And the error were then often false positives	real errors 0, false postive something like 5	integration within Atom is handy.	I like in other linters that you can click an 'error' and then are sent to a website with comprehensive explanation, and good/bad examples.	5	if errors would be discovered during the development process that would save a lot of time during programming.	-
6		4	I still got a lot of false positives.	5, of which 4 false positives.	Immediate feedback in the editor.	Check whether something is `nil`.	6	If the tool would find less unfair errors, than it would be useful in the future to prevent errors during development.	

Table A.4: Questionnaire on experiences with Analist after one week (questions and answers are translated)
[1]: How many days did you use Analist *in the end?, [2]: Can you explain this?, [3]: How helpful has* Analist *been to you? (scale 1-7), [4]: Can you explain this?, [5]: How many errors did* Analist *find to your knowledge? How many were real errors? How many were false positives?, [6]: What feature did you really like?, [7]: What feature would you have like to have, but was now missing?, [8]: Do you think that* Analist *could help you in the future during development?, [9]: Can you explain this?, [10]: Is there something else you would like to tell?*

BIBLIOGRAPHY

[1] Davide Ancona, Massimo Ancona, Antonio Cuni, and Nicholas D. Matsakis. "RPython: A Step Towards Reconciling Dynamically and Statically Typed OO Languages." In: *Proceedings of the 2007 Symposium on Dynamic Languages*. DLS '07. Montreal, Quebec, Canada: ACM, 2007, pp. 53–64. ISBN: 978-1-59593-868-8. DOI: 10.1145/1297081.1297091. URL: http://doi.acm.org/10.1145/1297081.1297091.

[2] Nate Berkopec. *Is Ruby Too Slow For Web-Scale?* Sept. 2017. URL: https://www.speedshop.co/2017/07/11/is-ruby-too-slow-for-web-scale.html.

[3] Cuoq. *What exactly does "context" mean in context-(in)sensitive analysis?* Nov. 2012. URL: https://stackoverflow.com/questions/13397180/what-exactly-does-context-mean-in-context-insensitive-analysis.

[4] Michael Furr, Jong-hoon (David) An, Jeffrey S. Foster, and Michael Hicks. "Static Type Inference for Ruby." In: *Proceedings of the 2009 ACM Symposium on Applied Computing*. SAC '09. Honolulu, Hawaii: ACM, 2009, pp. 1859–1866. ISBN: 978-1-60558-166-8. DOI: 10.1145/1529282.1529700. URL: http://doi.acm.org/10.1145/1529282.1529700.

[5] David Hauzar and Jan Kofron. "Framework for Static Analysis of PHP Applications." In: *29th European Conference on Object-Oriented Programming (ECOOP 2015)*. Ed. by John Tang Boyland. Vol. 37. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2015, pp. 689–711. ISBN: 978-3-939897-86-6. DOI: 10.4230/LIPIcs.ECOOP.2015.689. URL: http://drops.dagstuhl.de/opus/volltexte/2015/5243.

[6] David Hauzar and Jan Kofroň. "WeVerca: Web Applications Verification for PHP." In: *Software Engineering and Formal Methods*. Ed. by Dimitra Giannakopoulou and Gwen Salaün. Cham: Springer International Publishing, 2014, pp. 296–301. ISBN: 978-3-319-10431-7. DOI: 10.1007/978-3-319-10431-7_24.

[7] IEEE. *The Top Programming Languages 2017*. July 2017. URL: http://spectrum.ieee.org/static/interactive-the-top-programming-languages-2017 (visited on 07/19/2017).

[8] Daniel Iwaniuk. *Why do people say that Ruby is slow?* May 2016. URL: https://www.hawatel.com/blog/why-do-people-say-that-ruby-is-slow/.

[9] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. "Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities (Short Paper)." In: *Proceedings of the 2006 IEEE Symposium on Security and Privacy*. SP '06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 258–263. ISBN: 0-7695-2574-1. DOI: 10.1109/SP.2006.29. URL: https://doi.org/10.1109/SP.2006.29.

[10] Rafe Kettler. *Is there a way to print user-defined datatypes in ocaml?* Oct. 2011. URL: https://stackoverflow.com/questions/7518752/is-there-a-way-to-print-user-defined-datatypes-in-ocaml.

[11] Kevingray et al. *Making OCaml Accessible and Learnable for More People*. June 2017. URL: https://discuss.ocaml.org/t/making-ocaml-accessible-and-learnable-for-more-people/381.

[12] Etienne Kneuss, Philippe Suter, and Viktor Kuncak. "Runtime Instrumentation for Precise Flow-Sensitive Type Analysis." In: *Runtime Verification*. Ed. by Howard Barringer, Ylies Falcone, Bernd Finkbeiner, Klaus Havelund, Insup Lee, Gordon Pace, Grigore Roşu, Oleg Sokolsky, and Nikolai Tillmann. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 300–314. ISBN: 978-3-642-16612-9.

[13] Steve McConnell. *Code Complete, Second Edition*. Redmond, WA, USA: Microsoft Press, 2004. ISBN: 9780735619678.

[14] Stack Overflow. *Developer Survey 2017*. 2017. URL: https://insights.stackoverflow.com/survey/2017#technology-languages-over-time (visited on 07/17/2017).

[15] Stack Overflow. *Developer Survey 2017*. 2017. URL: https://insights.stackoverflow.com/survey/2017#technology-most-popular-developer-environments-by-occupation (visited on 07/17/2017).

[16] TIOBE. *Ruby*. Jan. 2018. URL: https://www.tiobe.com/tiobe-index/ruby/.