

Lehrstuhl für Datenbanksysteme
Wilhelm-Schickard-Institut für Informatik
Eberhard Karls Universität Tübingen

Diplomarbeit

Mining for Ruby embedded Queries



Florian Maier
September 23, 2009

Aufgabensteller: Prof. Dr. Torsten Grust
Betreuer: Tom Schreiber, M.Sc.

Declaration

I assure, that I have written this Diplomarbeit on my own and that I only used the sources and tools specified.

Florian Maier,
September 23, 2009

1 Abstract

This thesis describes the analysis and transformation of Ruby programs to make them compilable into a relational algebra, which can be executed on off-the-shelf relational database engines. This thesis is therefore a first step to bring the idea of Database-Supported Program Execution into the world of Ruby: Relational database engines, that directly and seamlessly participate in evaluation of Ruby programs.

Contents

1	Abstract	3
2	Mining for queries	7
2.1	Finding queries on list data and make use of them	7
2.2	Introduction to Ruby	8
2.3	The Ruby AST	9
2.4	The Ripper Library	9
2.5	The working steps	10
2.6	The example	12
3	From Ripper's output to a class model	13
3.1	Symbols of the AST	13
3.2	Creating a class model of the Ripper AST	18
3.3	Transforming the class model back to Ruby code	19
3.4	Implementation: the example	20
4	Normalization	25
4.1	Motivation	25
4.2	Normalizing the blocks	27
4.2.1	The block in Ruby	27
4.2.2	The replacement rule	28
4.2.3	Implementation: the example	31
4.3	Normalizing the methods	33
4.3.1	The methods	33
4.3.2	Adjusting the class tree	43
4.3.3	Implementation: the example	43
5	Handling Ruby's lists relationally	47
5.1	Motivation	47
5.2	Inferring collection lengths	50
5.2.1	Inner and outer length	50
5.2.2	Calculate the lengths	51
5.2.3	Implementation: the example	63
5.3	Evaluating the types of the tree nodes	66
5.3.1	The type system	66
5.3.2	Typecasting the class tree	67
5.3.3	Handling failed type checks	79

5.3.4 Implementation: the example	80
6 Conclusion	87
Bibliography	91
Copyrights	93

2 Mining for queries

This chapter will give an short overview of the work done, on which this thesis bases. At first there will be explained, what mining for queries is all about and what the key idea of this thesis is. Then the programming language Ruby will be introduced and a few of it's features, mainly the Ripper library, shall be explained to get focused on the importance of those features for the purpose of this thesis. By now being introduced into the programming language used for this work, the working goals might be presented and for getting an impression of the entire work the single working steps are shortly explained. A deeper look at this steps will follow in the next chapters and build the main part of this thesis. Finally an example Ruby program will be shown. All the working steps described later, will be applied on that example, so this example will be present throughout the entire thesis.

2.1 Finding queries on list data and make use of them

It is possible to find queries in every programming language as far as that language has programming constructs, supporting operations on objects, that are suitable as containers. The easiest and probably best know of this containers is an array of a certain data type. Every operation on that container might be regarded as a query, because if all the elements of the container are stored in a table of a database, it is possible to do the same operations on the elements by evaluating a query, that works on the database. So methods implemented on container objects are comparable to a similar operation on a table in a database. The difference is, that the operations on a container object and its members are computed by accessing the members on the heap of the program, what takes some time compared to a similar operation done on a table of a database. That's where the key idea of this thesis starts. If container elements and their methods resemble to database operations on a table, it could be much more effective to compute such an operation on a database instead of evaluating it by using the heap, especially if there are lots of members in the container elements, that have to be touched. To be able to do this, the source code of a program has to be analyzed and traversed into a relational algebra. From that algebra form it should be possible to get a database support. That means the database evaluates the queries. The goal of this thesis is to look at the source code of Ruby programs and transform and analyze them in a way, that a transformation into relational algebra is possible. Therefore two main objectives have to be achieved. First the query, that shall be computed by database support has to be found. This can be done by looking at the syntactical structure of a the program. Having finished that, the discovered query has to be analyzed and perhaps to be modified in certain ways. The programming language Ruby provides features, which allow an access to the syntactical

structure of a program, so that the ideas presented above might be implemented. How that works will be the topic of this thesis and the steps, that must be taken to achieve the goals will be presented in short form later in this chapter and detailed in the next chapters. But before doing so and for better understanding a short introduction to the programming language Ruby itself and its needed features for the purpose of this thesis will be given.

2.2 Introduction to Ruby

Ruby is a modern, higher programming language, which was developed by Yukihiro Matsumoto in 1995, meaning it is a very new programming language. Ruby is free software and the source code is available under the General Public License (GPL). So everybody can use Ruby and also modify the source code to adapt it to his requirements.

At first there is one important thing to note: Ruby is completely object oriented, that means, that every value represents an object, even a simple numeric expression, for example an integer value, whereas such expressions are built in or are primitive types in most programming languages. In opposite to languages like C++ or Java, it is not necessary to define a program explicitly in a class, a program can also be built by only defining methods. Because of the fact, that every Ruby expression has a value, it is also possible to use Ruby in functional programming style. So it is despite of the fact, that Ruby is fully object oriented, also possible to use procedural or functional programming in Ruby. In addition Ruby supports meta programming as well. There are two main principles in Ruby. The first one is the principle of least surprise (POLS), meaning that the designers of the Ruby language tried to construct the language the way, that users should not be surprised by the behavior of the language. The origin creator Yukihiro Matsumoto said the following sentence concerning the POLS:

'Everyone has an individual background. Someone may come from Python, someone else may come from Perl, and they may be surprised by different aspects of the language. Then they come up to me and say, 'I was surprised by this feature of the language, so Ruby violates the principle of least surprise.' Wait. Wait. The principle of least surprise is not for you only. The principle of least surprise means principle of least my surprise. And it means the principle of least surprise after you learn Ruby very well. For example, I was a C++ programmer before I started designing Ruby. I programmed in C++ exclusively for two or three years. And after two years of C++ programming, it still surprises me.' [Ven03]

The second principle refers to the fact, that Ruby knows no types. Ruby is a dynamic programming language, based on the idea, that the handling of an object depends not on its class, derivation from a class or formal specification, but on certain features, like methods. In a method definition for example it's assumed, that the types are suitable to the method. This is called 'duck typing', after James Whitcomb Riley's poem 'When I see a bird that walks like a duck and quacks like a duck, I call that bird a duck'. Using duck typing means, that just at runtime an object is checked for the certain features it should implement. This guides to more flexibility but reduces the ability to find errors

at compile time of a program. [FM08] If however a non suitable object is passed, Ruby raises a type error. The programmer can use these exceptions to make the duck typing safer, if he wants to.

Another interesting fact in ruby is, that there is no for-loop. Therefore structures called blocks are used. Blocks are code segments, which are executed after certain rules, for example for all elements of a data structure. Using them in this way, those data structures may be analyzed or altered. We'll see more about the usage of a block in case of a data structure later, when the 'map'-method is introduced and discussed. The latest version of Ruby is the version 1.9.1, which was released in January, the 31st. This version is also the Ruby version used in this thesis.

2.3 The Ruby AST

AST stands for abstract syntax tree. An AST represents a, in most cases, simplified structure of the source code of a programming language. This is done by creating a node for each construct in the source code. These nodes represent the essential structure of the input and are building a tree form in the syntactical order, they are occurring. The creation of an AST is normally done bottom-up. Note, that not every little detail is modeled in the AST as well; unnecessary details of the syntax, like semicolons for example, are omitted. That's why the AST is called abstract. [Jon03] Ruby provides the building of ASTs. This is an important feature, because the programmer is able to look at the syntactical structure of his programs. In Ruby v1.8 it was possible to extract the parse tree of an entire class or specific method. The result is modeled with Ruby's arrays, strings, symbols and integers as s-expression. In the following section there is a short example shown to get an impression, how a Ruby AST looks like. Creating such an AST or lets say more precisely such a parse tree was simply possible by using the ParseTree or RubyParser gem. [sRpb] Unfortunately this feature is no longer usable in Ruby 1.9. But it's still a way available to get an AST of a ruby program. There is a new build in library, which provides similar functionality as the ParseTree gem did. The library is called the Ripper library and will be explained in the next section.

2.4 The Ripper Library

As said before, the Ripper library provides a new possibility to get an AST from a Ruby program by getting access to the ruby parser. There are six methods in the library with different functionality but only one method is of interest for the purposes of this thesis and will be explained in more detail. The method is the 'sexp'-method and has the following syntax:

```
sexp(src, filename = '-',lineno = 1)
```

The method parses 'src' and creates a syntax expression tree. This tree is modeled as deeply nested array, where the members of the different subarrays are ruby symbols,

describing the syntax construct they refer to. In case of a 'leaf-array' (an array, that is no more nested) also information about the symbols value and position is given. To get a short impression, how such an expression tree looks like, I will refer to and show the example in the Ruby documentation for the Ripper library created by rdoc [RDOa] and explain it shortly. (Ruby documentation for every Ruby file and so for every library can be created by the user by using: `rdoc [options] [names...])` [RDOb]

```
require 'ripper'
require 'pp'

pp Ripper.sexp("def m(a) nil end")
#=> [:program,
     [:stmts_add,
      [:stmts_new],
      [:def,
       [:@ident, "m", [1, 4]],
       [:paren, [:params, [[:@ident, "a", [1, 6]]], nil, nil, nil]],
       [:bodystmt,
        [:stmts_add, [:stmts_new], [:var_ref, [:@kw, "nil", [1, 9]]],
        nil,
        nil,
        nil]]]]]
```

Figure 2.1: Example of an AST created by Ripper

The first two lines of output, shown in Figure 2.1, refer to the fact, that two libraries are required (keyword 'require') to use the `sexp`-method. Of course there has to be the Ripper library, because the `sexp`-method is part of the library. In addition the `pp` library is used, which provides a good readable presentation of an object ('pp' stands for 'pretty print'). In this case it is the return value of the `sexp`-call (see 'pp' command in front of `Ripper.sexp(...)`), that is visualized.

The third line executes the `sexp`-method and prints the 'pretty' presentation of the object to the standard output stream. The argument of the `sexp`-method is a short program fragment, noted as String, which would be a runnable Ruby program. Instead of passing a string directly, it is also possible to pass a file to the method. This will be explained later, when it comes to create classes out of the expression tree. The forth to the 14th line represent the output of the 'pp'-call. The output is a visualization in array notation of the syntax expression tree created by the `sexp`-method call. The symbols used to represent the syntax are discussed in the next chapter and will therefore not be explained any further at this point.

2.5 The working steps

Now, that it should be clear, how the Ripper library works and how above all the created output of Ripper, analyzing a Ruby program, looks like, the working goals

can be introduced. As already said, the main goal is to analyze a program in a way, that it can be translated into relational algebra, which will itself be translated into code, executable on a database. Because an analysis of the entire Ruby language would extend the scope of this thesis by far the determination was made, to lay the focus on Ruby constructs, that occur in a program written by Prof. Grust [Gru]. Of course the work done extends the scope of the program, it is possible to say, that this program is the basis of the work. Whenever there is talk of source code or origin code, code fragments or similar code out of that program are meant. To achieve this goal of analyzing that program and its constructs, different working steps have to be done. The first step is the transformation from the AST created by Ripper into a class tree representation. Then that tree has to be analyzed. Possible queries have to be detected and transformed, so that a translation into relational algebra is possible. To make this step easier the tree is normalized. There are two types of the normalization process. The first one normalizes blocks in the form, that only one block variable remains and the second one guaranties, that only the 'map'-function has a block left. That can be ensured by transforming the source code syntactically but not semantically. Another important information for getting database support for some methods is, that the lengths of the container objects, the methods work on, should be known. To get those lengths is another working step. Then the tree and especially the tree nodes have to be provided with some kind of type. The type reflects the encoding of an object, above all of container object, at one node. This is important for the translation. The last step is to check, whether all methods used in the source code get the right type of input to work correct. If that is not the case, the tree and the input types of a node have to be adapted. Finally the new source code of the now modified program can be receipt. This steps are shown in Figure 2.2 and will be explained in more detail in the following chapters. Therefore the example below will traverse all the steps presented above.

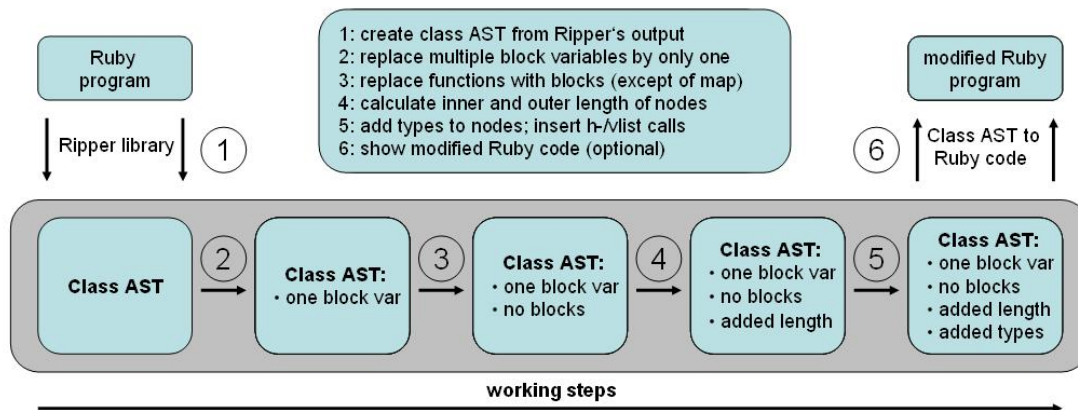


Figure 2.2: The working steps of this thesis

2.6 The example

The example, introduced in this section, is a short Ruby program. This program is used to explain all working steps from the original source code to the analyzed and typed class AST in the end and the resulting modified new source code. It is based on [Gru]. The code of the example is:

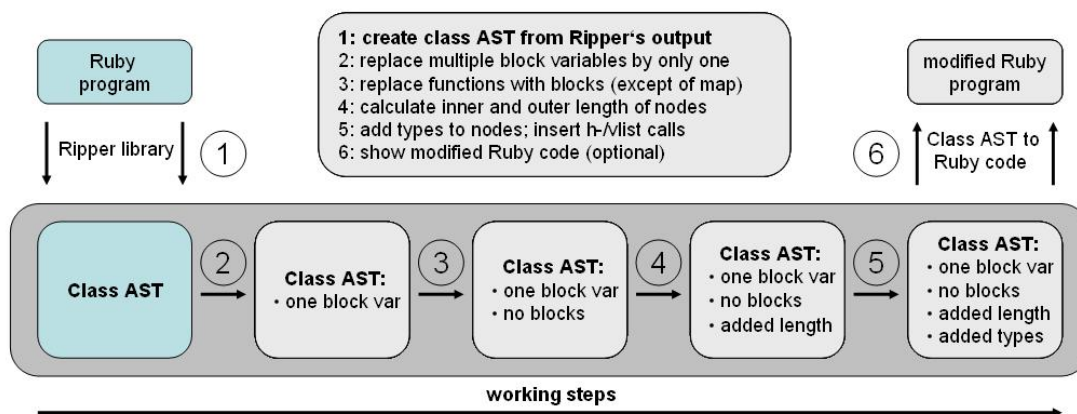
```
1 Lineitems.group_by { |li| li.l_orderkey }.map {  
2   |o, lis| [o, lis.select { |li| li.l_shipmode == "TRUCK" } ] }
```

Listing 2.1: Source code: The example

To understand the working steps, applied on the example in the next chapters, it is important to understand this two lines of code and what actually happens in this lines. At first there is a container called 'Lineitems'. This container stands for a Ruby suitable representation of a table, that has multiple elements and columns. The columns of that table can be accessed by *L_columnname*. In the example two times a special column of 'Lineitems' is addressed. That is the column 'l_orderkey' in the first line and the column 'l_shipmode' in the second line. Now that it is clear what 'Lineitems' represents, the next step is to look at the method called on it. A 'group_by'-call is done. This call groups the elements of 'Lineitems' by their 'orderkey'. The result of the call is taken as input for a 'map'-method call, which creates an array of an array with two elements, the 'orderkey' and the return value of the third method call in the example: the 'select'-call. This call takes every element provided by 'map' and returns an array of those ones, which have the 'shipmode' *TRUCK*.

In short sentences: This program returns an array of all elements of the table like structure 'Lineitems', which have the 'shipmode' *TRUCK*, grouped by their 'orderkey'.

3 From Ripper's output to a class model



This chapter is about the class model, which will be generated from the Ruby AST, produced by the Ripper library to represent the AST. At first, all the symbols, that occur in Ripper's output shall be explained, because they are used further on to represent the created class AST in different parts of the work done. For debugging and controlling purposes, it is also important to have a possibility to get all the way back from the class model to ruby source code, which will be necessary to check, whether the class tree represents indeed the same program than the ripper AST does (see optional working step six). This will be even more important, when transformations in the occurring blocks and methods in the AST will be made. For this chapter the optional sixth working step will be omitted, because the results would be the same as the input ruby code, but it was already a helpful feature for checking the correctness of the produced tree. The last part of this chapter refers to the example and it shall be visualized and explained how the transformation from the AST, produced by Ripper, to the class tree works.

3.1 Symbols of the AST

As visualized by the example in Figure 2.1 the AST created by Ripper is shown as nested array. Each subarray is identified by a symbol, which is stored as the first element of this subarray. The other elements describe the expression illustrated by the symbol closer. Depending on the case of the shown symbol, the description and so the length of the subarray differs. There are lots of different symbols in Ruby to represent all possible syntactical meanings, but for the purpose of this thesis, only some of them are used and

those will be explained in the following section. Unfortunately there is no documentation given by Ruby, which specifies all the possible symbols as a list, neither their meaning. But with a bit of thinking and analyzing the syntax tree, one should be able to get the meaning of the symbols and the following explanation ought to be right.

aref

'`aref`' refers to an array reference, that means an array is accessed at a certain position. So its subarray must have two additional elements apart of its name, one holding the position, on which the array is accessed, and the other containing the reference to the array the access is done.

arg_paren

The '`arg_paren`'-symbol is a very primitive one. It represents the parentheses around an argument of a method. It has only one more element beyond the one holding the name. This element contains an expression, that specifies all the arguments between the parentheses.

args_add_block

This symbol follows in most cases directly after the '`arg_paren`'-symbol. It is holding the elements of the arguments, given to a method. That means, that there are one or more elements in the array from which each one stands for one argument of a method call.

array

'`array`' should be very easy to understand. This symbol illustrates an array. It has always two elements: one for the symbol name '`array`' and one holding a nested array. Every element of this nested array acts as element of the array in the syntax tree.

assign

'`assign`' stands for an assignment, meaning that normally a value is assigned to a variable. So the subarray, containing the information about this node, has except of the first element, which represents as usual the name of the symbol two more elements. The second element holds the information about the variable of the assignment and the third element contains an expression that holds the assigned value.

binary

In this case a binary expression is expressed by the symbol. There are two operands and one operator, all in all implying four elements in this subarray. The element at position two represents the first operand, the element at position four the second one. The operator is located and saved in the third element.

block_var

This symbol expresses the variables of a block and this sub array has only two members, where the second one contains an expression, that names all of the variables used in this block. If it is not already clear, what a block structure means, how it looks like and how it works, it should be noted, that this will be explained in the next chapter, when the 'map'-method is discussed. Maybe this explanation will be clearer after reading that section.

brace_block

As ':block_var' represents the variables of a block, ':brace_block' symbolizes the entire block of a method and has therefore two additional elements in its describing array. The first of these refers to the description of the block variables and the second one to the expression in the block itself.

call

Understanding what a ':call'-symbol in the AST stands for is a little bit difficult. As the name supposes to say, it could be a call to a method, what might be right in some cases, but not always. A ':call' refers to a method call, that has no arguments, for example '*uniq()*' or '*size()*'. Method calls, that have arguments or even a block are illustrated different as ':method_add_arg' or ':method_add_block' but that will be explained later.

command_call

This symbol is similar to the ':call'-symbol. The difference is that a ':command_call'-symbol identifies a call(element two of the describing array) on a value given by another call(element three of the array) and that call has additional arguments(element four of the array).

@const

':@const'-Symbol is nearly self explaining. It describes a constant in Ruby's syntax tree environment. The constant, named by this symbol needs not be a real constant as supposed and could be altered somewhere else in the program, but in the context of the expression tree, the value of the constant remains unchanged. So there is only one more array element, expect the symbol name, and that is the name of the constant, described by this tree node.

dot2

Ranges in a Ruby AST are represented by the 'dot2'-symbols. A range is defined by two values, the start and the end value. A range is noted as follows: (start_value..end_value), meaning all the values between start and end_value (normally incremented by one). The

conclusion is that a 'dot2'-subarray consists of the symbol name and the two values, that define the range.

@ident

'@ident' stands for identifier. It's a very simple symbol, as it only identifies the name of a variable or a variable reference. Therefore it is enough to only consist of two array elements, symbol name and the name of the identified variable.

if

'if' displays a condition. The subarray has three or four elements. Three in case that there is no 'else'-clause. One element for the symbol name, one for the condition and for the expression, executed if the condition part returns a true value. If there is an 'else'-clause, there is one more element. This element contains an array, whose first element is the symbol 'else', which is not explained in this chapter as it is no autonomous symbol but depends on the 'if'-symbol. The second element of the 'else'-array describes the expression in the case the condition turns false.

@int

Integers are visualized by this symbol. Similar to the identifier or the constant, there's one additional array element and that is of course the value of the integer described.

method_add_arg

As mentioned in the subsection, referencing the 'call'-symbol, this symbol expresses the syntax of a method call with one or more arguments. For such a representation, there are two more elements in the subarray. One, that describes the call to a method (by a 'call'-symbol) and one, that describes the arguments of the method.

method_add_block

Nearly identical to the 'method_add_arg'-symbol, concerning the syntactical structure, this symbol refers to a method, that is called with a block. As above there is an additional element, describing the call and one for containing the informations about the block.

params

'params' stands for parameters, more special for the block parameters within a block. These parameters are laid down in an array, which is the second element (and so a subarray) of the array describing this symbol.

paren

Similar to the `:arg_paren`-symbol, this one holds the information about the content, that lays between two parentheses.

program

This is always the first symbol, showed in the expression tree, as it identifies the program, which is analyzed by Ripper. The array representing a `:program`-symbol has normal multiple elements. Each holding the information for one entire expression.

rest_param

`:rest_param` categorizes an array, that contains all the elements of an underlying data structure, that are not directly accessible by a block parameter. More informations about this symbol and a more detailed explanation of the `rest_param`-identifier `*name_of_rest_param` are given in the next chapter, in the section, where blocks are treated.

string_content

The `:string_content`-symbol describes, as the name says, the content of a string literal. There is only one additional parameter, which contains a `@tstring_content`-symbol.

string_literal

A `:string_literal` refers to a string. But the string is not represented directly. The content of this symbol is not the string itself, but a `:string_content`-symbol.

symbol_literal

`:symbol_literal` has nearly the same meaning as the `:string_literal`. But it references to a `:symbol` instead of a `:string_content`.

symbol

`:symbol` shows us a symbol by containing an identifier, which holds the name of the symbol, described by this subarray.

@tstring_content

Another subsection of a `:string_literal`-symbol. But this one finally holds the value of a string. Why there is a three times nesting for illustrating one simple string literal seems to complicated to know and therefore, but mainly due to the lack of documentation, remains unknown.

unary

This symbol is similar to the one visualizing a binary expression, the only difference is of course, that there is no second operator and so one element less in the describing array is enough to save the information of this symbol.

var_field

A `'var_field'` refers to a new variable, which was not already defined. It has one more element, that holds an identifier, to name the new created variable.

var_ref

Last but not least there is the variable reference. This symbol represents an access to a variable, that has been defined earlier in the program text. As the `'var_field'` in most cases, it is holding an identifier for naming the variable, but it could be a constant as well, because the variable was already defined.

3.2 Creating a class model of the Ripper AST

As already mentioned at the beginning of the last section, the AST is shown as nested array. This nested array can be regarded as tree. That means in more detail, for every symbol, that represents some syntactical information of the AST, an array is created. This array represents a node in the tree structure, given by the nested array. In other words: every subarray of the hole tree(the entire nested array) contains the information about the symbol and the node in the tree, which the symbol stands for. This information about the symbol and node is not only the name but much more important the continuing tree structure. So in this case a symbol, referring to its corresponding subarray in the nested array, is the same as a node in the syntax tree and so these terms can be used equally.

It should now be clear, how to create a class tree out of the nested array. The nested array's first element, which was given by the Ripper library ought to be a `'program'`-symbol. To create a class model the nested array has to be traversed and for each subarray, that was found on the way through the nested array, a new node in the class tree has to be created, until all subarrays have been traversed.

How those new created classes look like depends on the kind of symbol, that expresses the node wanted to be created. So the new class refers to the symbol in the tree and of course to the information, contained by the other elements of the subarray of the current symbol. This information is very important for getting a appropriate class tree, because this information shows the features of the node, that should be created. Here features means, how the tree goes on in case of a binary expression or how the name of a constant is, what the name of a method is and so on. That means, that for every different symbol in the nested array, there has to be a corresponding class and the abilities of that class must refer to the informations, given in the subarray of the symbol. To create a sufficient

class to a symbol the meaning and content of that symbol have to be considered, which was done in the last section.

Although a short example should be given how such a tree node class could look like. A binary expression should illustrate this as example. At first, lets look at the description of the ':binary'-symbol in the last section. To get a appropriate class we need two operands, which both could be any expressions, and one binary operator. So the class created should have two children, a left and right one to represent the possible expressions and a structure, that contains the operator. Assumed that our new class object should also have a name, inherits from a super node class and all the mentioned attributes should be accessible, a binary class could look like the one shown in Listing 3.1.

```
1 class Binary < Ast_node
2   @name
3   @left
4   @right
5   @operator
6
7   def initialize(name, left , right , operator)
8     @left = left
9     @right = right
10    @name = name
11    @operator = operator
12  end
13
14  attr_accessor :name, :left , :right , :operator
15 end
```

Listing 3.1: A possible binary class

3.3 Transforming the class model back to Ruby code

When the entire nested array is transformed into a class tree, several changes are applied on this class tree. It is normalized, as the next chapters will explain, that means certain expressions are replaced with other expressions, but the meaning does not change. Normalization goes hand in hand with altering the class tree by certain rules. These rules and the goals of the normalization will be discussed in the next chapter, but there is one problem, created by altering the tree and its structure. It is not quite clear, if the normalization was correct and the tree still has the same information as it had before. This fact should be able to be tested. Also there should be a possibility to visualize the new modified source code, which is created by the transformation of the origin tree. The simplest way to do this is to transform the modified class tree back to Ruby source code. Then it will be easy to check whether the new created source code produces the same results as the original source code without any altering.

The implementation of this step is very similar to the way described above, where an nested array is transformed to a class tree. The difference is, that it is not quite the same way back, because there should be source code created and no nested array. As

said before the AST representation of the program code omits unimportant syntactical details. This must be considered, when writing the class tree back to the source code. Every syntactical detail, that was omitted must be added even when the class representation of a node does own the exact information about these details. To give an example of such omitted details: the commas between multiple arguments of a method are not illustrated in the AST nor in the class tree. That's why they have to be added explicitly to the resulting source code.

The way to do this, is to traverse the class tree top down. For every node source code has to be created. The source code is maintained by a string, which is modified by one node after another, until every single node in the class tree was visited by the program which creates the source code back. An important thing about this is, that the children of the nodes have to be visited in the correct order to get the right source code. That is something, what the programmer has to look at carefully. When we take the example from above, the binary expression, the source code generation would be done like follows. As the node, that should be transformed back to Ruby code, is a binary expression, it is clear, that it has two children, which both represent a new node in the class tree and one operator. At first the left child and the expression, laying under that node, have to be evaluated. When that is done and the program returns from analyzing and writing the code of the left expression, the operator is added to the source code string. Finally the evaluation of the right child and its contained expression is done. After that the next node will be treated.

3.4 Implementation: the example

To visualize the working step explained in this chapter, this step is implemented by using the example described in the first chapter. At first the source code showed in Listing 2.1 is passed to the Ripper library, more specified to the 'sexp()' -method call of that library. That call produces a nested array object with different symbols, describing the syntax of the program. By using the Ruby function 'pp' alias 'pretty print' that object can be displayed in a form its syntax is easy to understand. That output is shown in Listing 3.2. Every subarray, starting with one of the symbols described in section 3.1, that lays on the same imaginary vertical line belongs syntactically to the array, that is lying above and more left than the subarray. So it is easy to see, that every other subarray belongs to the symbol 'program' and that the 'call'-symbol in line three of Listing 3.2 belongs to the 'method_add_block'-symbol in line two, while having three children: the 'method_add_block'-symbol in line four, the '.' in line 17 and the 'ident'-symbol in line 18.

```
1  [: program ,
2    [: method_add_block ,
3      [: call ,
4        [: method_add_block ,
5          [: call ,
6            [: var_ref , [: @const , "Lineitems" , [1 , 0]]] ,
7            : "." ,
```

```

 8     [:@ident, "group-by", [1, 10]],
 9   [:brace_block,
10     [:block_var,
11       [:params, [[:@ident, "li", [1, 20]], nil, nil, nil, nil],
12         nil],
13       [[:call,
14         [:var_ref, [:@ident, "li", [1, 24]],
15           :".",
16           [:@ident, "l_orderkey", [1, 27]]]]]],
17     :".",
18     [:@ident, "map", [1, 39]],
19   [:brace_block,
20     [:block_var,
21       [:params,
22         [[:@ident, "o", [1, 45]], [:@ident, "lis", [1, 47]],
23         nil,
24         nil,
25         nil,
26         nil],
27       nil],
28     [[:array,
29       [[:var_ref, [:@ident, "o", [1, 53]],
30         [:method_add_block,
31           [:call,
32             [:var_ref, [:@ident, "lis", [1, 55]],
33               :".",
34             [:@ident, "select", [1, 59]]],
35           [:brace_block,
36             [:block_var,
37               [:params, [[:@ident, "li", [1, 67]], nil, nil, nil, nil],
38                 nil],
39             [[:binary,
40               [:call,
41                 [:var_ref, [:@ident, "li", [1, 71]],
42                   :".",
43                 [:@ident, "l_shipmode", [1, 74]],
44                 :=,
45                 [:string_literal,
46                   [:string_content,
47                     [:@tstring_content, "TRUCK", [1, 89]]]]]]]]]]]]]]]]]]]]]]]]]]]]]] >

```

Listing 3.2: Output of the `sexp()`-call of the Ripper library

If the entire array is traversed and the corresponding nodes to the subarrays and symbols have been created, as described in section 3.2, then that results in a class tree of nodes. As an output using 'pp' would show this tree in a similar way than Listing 3.2 does, there should be another possibility to present such a tree in a more vivid form. Therefore the program Dot can be used. As that is not the main topic of this thesis, but also a necessary part for the visualization of the class trees a short slide-in at the end of this chapter handles this step.

Slide-in: Creating a graph with Dot

Dot is a free available program from Graphviz, that allows the user to draw graphs easily. For doing so only the nodes must be created and their edges must be defined, how that exactly works, can be found in [GKN06]. Fortunately there is a support library for Ruby, so that no output strings have to be created as in the dot guide described, but the output can be created directly in Ruby. Unfortunately there is only a very poor documentation about this library given but some examples facilitate the usage [sRpa].

But that is not the main focus, the interesting part should be how to create a Dot-graph from our class tree. Similar to creating that tree out of the output from the Ripper library, the class tree has to be traversed node by node. At every node, it has to be checked, if that node has some children and how much children the node has. For every child an edge has to be created, which starts at the parent and ends at that child. So the class tree has to be traversed from the top to the bottom recursively, until all nodes have been traversed. If that is finished, the tree can be printed.

To simplify the tree and to make it smaller, some nodes can be summarized to one graph node, depending on whether their syntactical meaning is. So this is done for example for the 'method_add_block' and the 'call'-nodes. They build together a new graph node called 'Method with block'. In an appropriate manner other nodes are summarized too, but it should not be necessary to mention all these compressions, because their semantical meaning is not affected by the compression.

After describing the creating of the graph out of the class tree shortly in the slide-in above, the result can be seen in Figure 3.4. As said in the slide-in some nodes in the graph are a compression of multiple nodes in the Ripper AST. Method or function calls are displayed by an ellipse and light blue color, because that nodes might be normalized in the later working steps, whereas all other nodes are visualized by a gray rectangle. In any case it is obvious, that the graph visualization is much more vivid, than looking at the array representation created by 'pp' from Ripper's output.

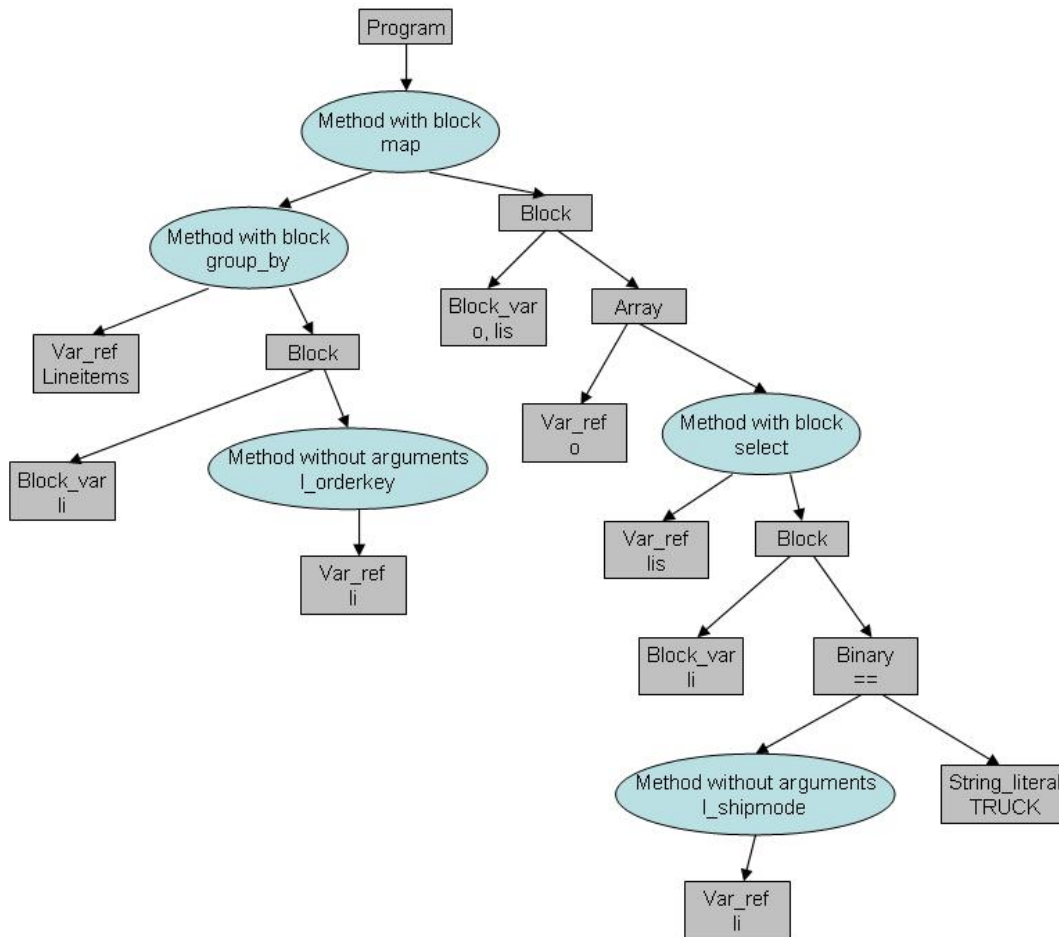
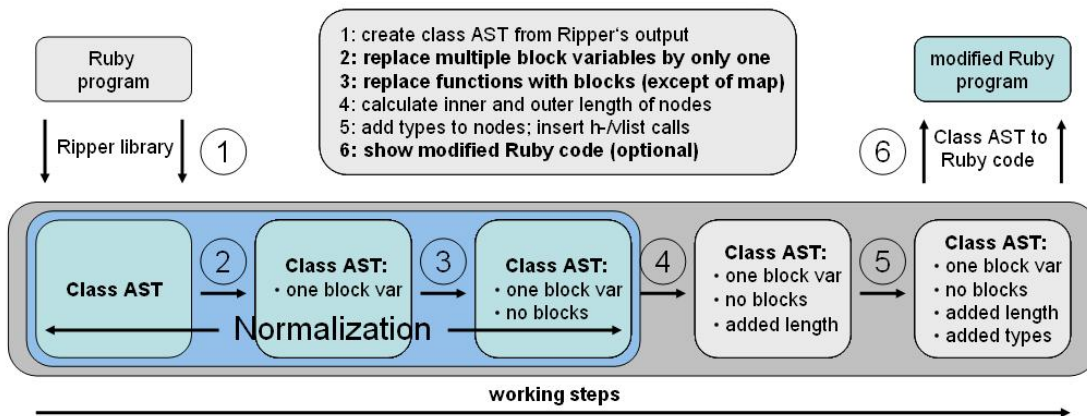


Figure 3.1: Visualization of the class tree

4 Normalization



This chapter refers to the normalization of the class tree. The normalization covers two working steps, as can be seen above: The replacement of the blocks as well as the replacement of the methods. Despite the fact, that both working steps are independent of each other, they are discussed in one chapter, because they concern the same topic. The normalization is just split into two working steps, but both steps refer to the same problem of the complexity of the source code, especially the block constructs and different methods. Why that is a problem, will be motivated in the next section, followed by the explanation of the two working steps.

4.1 Motivation

To understand why normalization is necessary and indispensable to the purpose of this thesis, the goal of the work has to be regarded. The goal is to prepare Ruby programs for their translation into relational algebra. To facilitate the translation process, the source code of the translated program should be as easy as possible. At this point the normalization comes into play. As Ruby provides powerful but also complicated programming constructs, like blocks or complicated methods, a consideration has to be made, how that constructs can be transformed in a easier form, without altering their semantic.

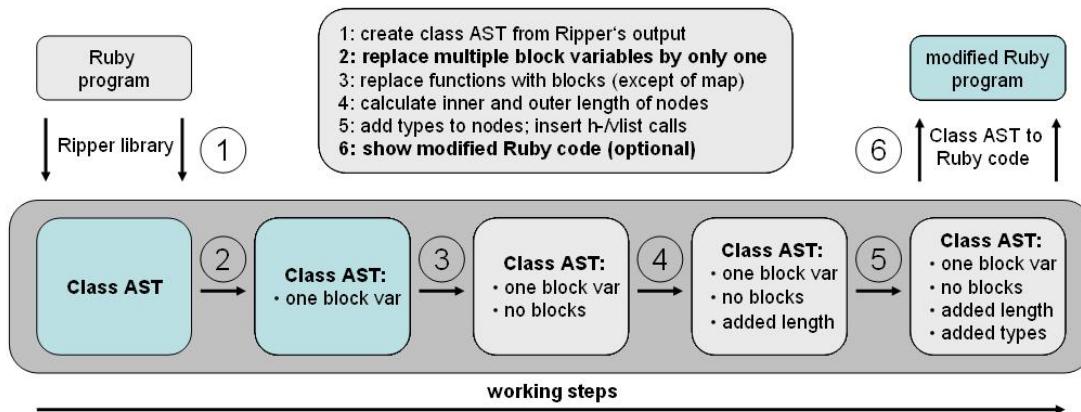
Therefore the first step is to transform the syntax of blocks in a less complicated form. That can be achieved by the guarantee, that a method has only one block variable left, after it was normalized. This makes sense, because a block is a very specific program-

ming construct and so it is difficult to translate a block construct, having multiple block variables, into relational algebra for further database support. As the block describing section 4.2.1 will show, it is possible to use an arbitrary amount of block variables and even an array parameter as block variable, what would make a translation very complicated. In any case, it is much easier to translate a block with only one parameter, than taking care of an arbitrary number of block parameters. After this working step however, every occurring method has only one block variable left, that is used in the block, independent on how complicated the block looked before. An additional advantage of normalizing the blocks is, that the also necessary normalization of the methods, following in section 4.3, is easier to handle, because it can be assumed, that every method has only one block parameter left and the normalization of those methods does not have to look after cases with more block variables.

The second working step of the normalization concerns the methods, provided by Ruby. Very powerful and versatile method are made available by the Ruby libraries. The problem is, that this advantages result in very complicated methods. Most methods are able to iterate over a given collection of objects and to handle every single one of it, by passing it to a block. That however, is a very complex and complicated behavior. Because the process of the translation into relational algebra should be as easy as possible, such complicated methods are undesired for translation. Quite the contrary is the case, methods should have an easy behavior to offer a simple way of translation. So the complicated methods need to be transformed into an basic form, with basic behavior. Therefore the rule is introduced, that the only method, allowed to iterate over a collection of objects, is the 'map'-method. All other method, must use this method for the iteration process. The result is, that the complicated way of iterating over elements of a container object and the simultaneous evaluation of a predicate is separated into two method calls. The 'map'-method call for the iteration and a call to the basic method of the normalized method to evaluate the predicate. That separation of this two steps makes the translation much easier, because only basic methods have to be translated and if an iteration occurs, that iteration is always done by the 'map'-method, resulting in only one translation for an iteration process. Another advantage is, when methods are treated like that and normalized into combinations of 'map'-method calls and calls to basic methods, that only the basic methods have to be translated, what leads to much less work.

To summarize the goals of the normalization: The syntax of the source code, given by a Ruby program, should be as simple as possible to guarantee the easiest translation into relational algebra. That is reached by transforming the class tree and the source code according to the following suggestions. It must be ensured, that after the first normalization process, every method has only one block variable left. After the second step, the only method allowed to have a block, is the 'map'-method, resulting in a separation of iteration over a collection and computing it. Additionally all method should be reduced to some basic methods, with simple behavior. In the next two section an explanation is given, how these goals are achieved and implemented.

4.2 Normalizing the blocks



This section refers, as shown in the figure above, to working step two, how is an independent part of the normalization process. For understanding, how the normalization of a block works, it is important to understand, what a block actually is. That's why an explanation of that, will follow at first, before the focus lies on the normalization itself. The goal of the normalization is discussed there and it will be described, how a block has to be transformed to suffice the normalization goals. Also the replacement rules for that transformations are shown.

4.2.1 The block in Ruby

So called blocks are a Ruby specific feature, which might be unknown to programmers, used to other programming languages. Therefore and for the fact, that blocks are used very often and represent a powerful feature of Ruby, the meaning and syntax of blocks will be explained in the case, they are used in this thesis.

Blocks are actually nothing more than chunks of code, that are enclosed by curly brackets. These chunks can be used in method invocations just as parameters, like it is done below in the 'map'-method or other methods. Concerning these methods, the code in the block determines the value of an element of the return array. The calling data structure determines just the initial value, that is passed to the block and the block does the work. This ability to write any possible code into a block offers many possibilities to use the methods listed in this chapter. For example the 'map'-method is able to compute everything, as the start values of the array are passed and the block can hold arbitrary code. For better understanding the syntax of blocks used by a method, it is shown below.

```
enum.method { |obj| block } → array or enumerator
```

As you can see in the syntax description of the arbitrary method above, a block is build by two parts. The first part is the 'obj'-part. This part refers to the block vari-

ables. Block variables are variables, that are only visible within the block and override the visibility of variables with corresponding names outside of the block. There is the possibility to give more than one block variable to a block, it is even possible to create a parameter, that presents an array of values by using the '*'-operator. To understand this, the second part has to be looked at. What the second part (the block part) does, was discussed above, but the interesting fact is how the input array, the block variables and the block itself work together. For every element in the input data structure (in the case of the code above 'enum'), the following steps are executed. At first the element is referred by the variables. Depending on how much block variables are given and how much elements the passed element (think about a nested array) has, the reference takes place. If there is only one element, the first block variable is associated with that value, any other block variables becomes 'nil'. In case the input data structure is nested, assume an array of a two element data structure, the first two block variables get values. As it is sometimes impossible to know, how the nesting is, Ruby provides a functionality solving that problem, the '*'-operator. This operator creates an array of the remaining nested elements, that have no corresponding block variable. So the block variable marked with the '*' represents actually an array.

When all input values are assigned to their appropriate block variables, those values are passed to the block. They can now be accessed by the block variables, meaning every block variable represents a value of the input data structure and is now able to be computed.

The following lines may give an impression of how a possible definition of multiple block variables may look like. This is important to look at for understanding, how the normalization of a method with a block works and which problems may occur, creating the replacement rules. Note, that such code makes only sense, if the input data structure *enum* is nested. [FM08] [TFH09]

1. *enum.method* $\{|x_0, \dots, x_n| \textit{block}\}$
2. *enum.method* $\{|x_0, \dots, x_n, *y, x_p, \dots, x_{p+q}| \textit{block}\}$
3. *enum.method* $\{|*x, y_0, \dots, y_n| \textit{block}\}$
4. *enum.method* $\{|x_0, \dots, x_n, *y| \textit{block}\}$

4.2.2 The replacement rule

As already said, the normalization goals demand, that every function has only one block variable. That's why all the cases, shown above, have to be normalized into a presentation of the methods, where only one block variable is available. So in every

case, visualized above, the multiple block variables will be replaced by one single block variable. This block variable must have a name, that is never used anyway in the program, because if a replacement would take place and there would be another variable with the same name, only god knows, what would happen to the behavior of the program. The next sections will explain for the different cases, how the replacement is done in the blocks.

In the first case shown, an replacement is a relatively easy task. As it is possible to use the positional access of an array by using `array[position]`, a way to replace two or more block variables by one would work like this. Because there is only one block variable in the normalized case, that variable will refer to the entire array. Instead of using more block variables, it would suffice to replace them with the positional access to the array represented by only one block variable. To illustrate this in the code above: the block variables `'x0, ..., xn'` would be replaced by a new single block variable `'sbv'`. In the block `'x0, ..., xn'` don't occur any more. Only `'sbv'` is used and as the positional access is able to replace `'x0, ..., xn'`, `'x0'` would now be shown as `'sbv[0]'`, `'x1'` would be presented as `'sbv[1]'` and so on. Note once again, that the name of the new block variable must not be used anywhere else in the program, because the replacement would go terribly wrong. It has to be ensured, that the new block variable is only present in the block, that is normalized. In more formally way, that would result in the following replacement rule. To shorten the rule, `'enum'` is represented by `'e'`, `'method'` by `'m'` and `'block'` by `'b'`

$$b \doteq b'$$

$$\frac{}{e.m\{|x_0, \dots, x_n|\} \doteq e.m\{|sbv| b'[\frac{sbv[0]}{x_0}], [\dots], [\frac{sbv[n]}{x_n}]\}}$$

The other three cases are not that easy to normalize. Of course there is a new single block variable (`sbv`) in the definition. The difficulty lies in the problem, that the array, representing the `'*'`-parameter in the block, is able to have an arbitrary size. So there must be some kind of analysis, in which the length of the array is found out. Otherwise no sensible replacement could be made. To do so, it is at first important to know whether the `'*'`-parameter lies at the beginning, in the middle or at the end of the block variable definition and in every of this case the normal parameters without `'*'`, that are listed in the definition have to be counted.

If lying in the front (case three) of the original block variable definition, the `'*'`-parameter represents all items from the beginning of the array, represented by the single block variable (`sbv[0]`), until the position of that array, which is described by the length of that array minus the number of regular block variables (in this case `n`) plus one (array number starts with zero), following the `'*'`-parameter in the original definition (`sbv[sbv.length - (n + 1)]`). According to that range, every block variable in the original definition can be addressed by a positional access on an array (represented by the new block variable `sbv`). The formally replacement rule should make this more clearly. Just as explanation, if any rule takes two lines, the second line shows the exact continuation after the end of the first line.

$$\begin{array}{c}
 b \doteq b' \\
 \hline
 e.m\{| * x, y_0, \dots, y_n |\} \doteq e.m\{| sbv | b' [\frac{[sbv[0], \dots, sbv[sbv.length()-(n+1)]]}{*x}], \\
 \frac{[sbv[sbv.length()-(n+2)]]}{y_0}, [\dots], \frac{[sbv[sbv.length()-1]]}{y_n}] \}
 \end{array}$$

If the '*'-parameter lies at the end (case four), the replacement is the other way round, than described above. The '*'-parameter represents now an array, starting at the position n (array numbering starts at zero) and ends at the position $.length() - 1$ of the array, represented by the new inserted block variable sbv . The replacement rule looks like:

$$\begin{array}{c}
 b \doteq b' \\
 \hline
 e.m\{| x_0, \dots, x_n, *y |\} \doteq e.m\{| sbv | b' [\frac{[sbv[0]]}{x_0}, [\dots], \frac{[sbv[n]]}{x_n}, \\
 \frac{[sbv[n+1], [\dots], sbv[sbv.length()-1]]}{*y}] \}
 \end{array}$$

The hardest case is the case, where the '*' parameter is positioned in the middle. In this case the block variables of the origin definition, that are lying behind the '*'-parameter and the position of the '*'-parameter itself have to be counted. For all variables, lying in front of the position of the '*'-parameter, the simple rule of case one can be used, for all other parameters the rule of case three can be applied. Formally that would look as follows:

$$\begin{array}{c}
 b \doteq b' \\
 \hline
 e.m\{| x_0, \dots, x_n, *y |\} \doteq e.m\{| sbv | b' [\frac{[sbv[0]]}{x_0}, [\dots], \frac{[sbv[n]]}{x_n}, \\
 \frac{[sbv[n+1], [\dots], sbv[sbv.length()-(q+1)]]}{*y}, \frac{[sbv[sbv.length()-q]]}{x_p}, [\dots], \frac{[sbv[sbv.length()-1]]}{x_{p+q}}] \}
 \end{array}$$

Normally every other node, occurring in the class tree, needs to have a replacement rule as well. Otherwise the handling would not be possible by traversing the class tree recursively. But those rules are primitive and don't alter something in the class tree, so they are omitted. To get although an impression of how the rules might look like, the replacement rule of a binary expression, using the '+'-operator, is given as example. It is easy to recognize, that the block replacement has no effects on the binary expression itself. The rules for other node behave similar.

$$\frac{e_1 \doteq e'_1, e_2 \doteq e'_2}{e_1 + e_2 = e'_1 + e'_2}$$

Knowing now, how multiple block variables are replaced in theory, the class AST has to be modified accordingly to these rules. New nodes have to be created and to be inserted. Therefore every node, which contains block variables, has to be found and checked for the case, block variables occur. An analysis of the number of the block variables has to be done and the block variables themselves and all of their occurrences in the block have to be replaced in the way the replacement rules say. Depending on whether type the block variable(s) is, the class tree has to be modified or a new node has to be created and inserted in the class tree.

4.2.3 Implementation: the example

This section shows how multiple block variables are replaced by one single block variable and how the replacement changes the block of the method, on which the replacement is done. In the example shown in Listing 2.1, there is one method, that uses more than one block variable. That method is the 'map'-method and its block variables are 'o' and 'lis'. That block variables must be replaced by one variable, in this case 'x'. This replacement variable must never be used in any other part of the program. That would result in complete chaos, because the program would be altered in a unpredictable way. So it has to be ensured, that a name is used, that does not occur anywhere in the program, not even by the matter of hazard. Normally the new variable would not be as simple as 'x', but in the case of the example 'x' is a possible new block variable. In the block the two old block variables 'o' and 'lis' have also to be replaced by using the new block variable 'x'. As 'o' is the first old block variable and following replacement rule one from above, it is replaced by 'x[0]'. That means the array described by 'x' is accessed and of that array the first position is referenced. Similar the old block variable 'lis' has to be replaced by 'x[1]'. Note, that there is now a difference in the new tree and the old tree. Because the two block variables have been replaced by one, that is used by a positional access on an array, the new tree looks a little bit different. Where the old tree has had only 'variable_ref'-objects for a variable reference, the new tree needs an 'aref'-node, representing a reference to an array. That array is of course again specified by the 'variable_ref'-object from the old tree point to the appropriate variable. Listing 4.1 shows the new source code, which is generated by those replacements and Figure 4.2.3 illustrates the according new class tree. The added/different nodes are emphasized by using bold print.

```

1 Lineitems.group_by { |li| li.l_orderkey }.map {
2   |x| [x[0], x[1]].select { |li| li.l_shipmode == "TRUCK" } ] }

```

Listing 4.1: Source code of the example with replaced blocks

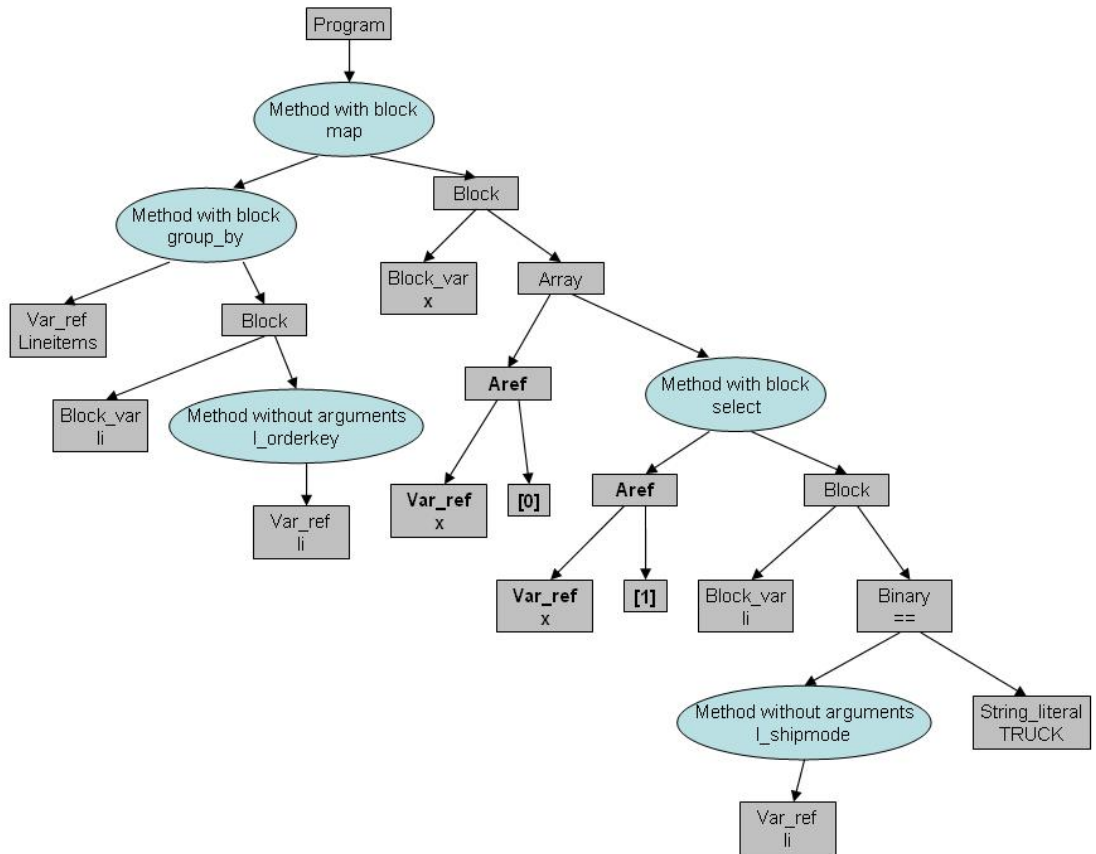
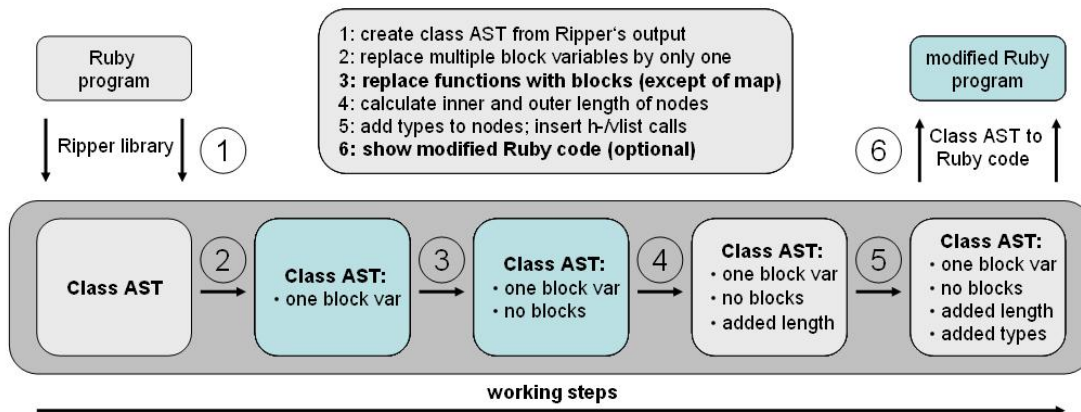


Figure 4.1: Class tree with replaced blocks

4.3 Normalizing the methods



This section is treating working step three, the normalization of the methods. To get an impression, what normalizing methods means in the case of this section and how it is done, there will be a short explanation. Normalization refers to Ruby methods. These methods are implemented in the Ruby class 'Enumerable'. Normalization concerning these methods means an analysis of the method and then a transformation. A few methods shall be taken as basic, implying that these methods don't have to be normalized, in contrary the normalized methods are expressed by using the basic methods.

To realize that, there are two things to do. The first thing is to think about a proper and possible replacement of the normalized methods by an expression, that uses only the basic methods. That means defining a replacement rule.

The second one is to transform the class tree in the way, that, after the altering, it holds the new information, given by the replacement rule. Therefore a new node has to be created in most cases, replacing the entire old method node. If not the whole node has to be replaced, the subexpressions further down in the tree of that node (seen from the root) have to be carefully modified, so that they hold the new information. Because this step may be very complicated, it is essential to test, if the new or altered normalized node contains the right information. That's why the back transformation from the class tree to ruby code, explained in the last chapter is a such indispensable task.

This section will now introduce those methods, which get normalized, including the replacement rules, the 'map'-method as only method used for iteration, as well as it will present the basic methods.

4.3.1 The methods

This section introduces and describes all methods and their functionality, which are supported to a later translation into relational algebra and database code. That includes at first the 'map'-method as the only method, allowed to use a block. Then the basic helper

methods are presented. Those methods are necessary to get a suitable normalization of the last group of methods: the methods, that will be normalized.

The 'map'-method

As all the Ruby methods, that are explained in this chapter, 'map()' is defined in the module 'Enumerable'. The syntax of the 'map'-method is:

$$enum.map \{ |obj| block \} \rightarrow array \text{ or } enumerator$$

The return value of the 'map'-method is a new array. This array holds the results, evaluated by the block. How exactly a block works, will be explained in the next section. Anyway, the block calculates a result for each element in the given structure *enum*. This works like an iterator on the input structure would do. Every element of this structure is passed to the block and evaluated there. If there is no block written, 'map()' will return an Enumerator object.[TFH09] [FM08]

The primitive helper methods

The primitive helper methods are some basic methods, which have a close defined functionality and are directly supported by the relational algebra. Their cause of existence is, that they provide a much more easily normalization. In some cases the normalization would not be possible, if these helper methods did not exist. The database support expressions, normalized with these helper method, may be translated very effectively. Therefore their functionality and syntax will be explained shortly.

DROP(): 'DROP' is a basic method, which allows it to, as the name might suggest, drop elements of a collection. It takes one array, on which it is called, as input and drops the first 'n' elements. So it is suitable to a normalization of methods, which shrink the input array, for example the 'last'-method. Syntactical the methods looks like this:

$$array_with_values.DROP(n) \rightarrow array$$

FILTER(): This method is mainly used, when there is a normalization of a method, that uses some kind of predicate. The predicate is evaluated in some other expression and the results are placed in the second array of this method, then the filtering is done. The syntax is:

$$array_with_values.FILTER(array_with_boolean) \rightarrow array$$

The 'FILTER'-method takes two arrays with the same length as input. The first array is the array on which the method is called. The array contains arbitrary values. The second array, passed as argument to the 'FILTER'-method, is an array containing only boolean values, that means 'true' or 'false'. The return value of the method is an array,

containing all values of the first array with the arbitrary values, which have a matching 'true'-value in the second array at the same position.

FIRST(): 'FIRST' provides a similar behavior than 'DROP' has, the difference is, that it does not drop elements from the end of the input array and return the others, it takes the first 'n' elements of a collection instead and returns them. The 'FIRST'-method is used to normalize the Ruby method 'first' into 'FIRST(1)'. The syntax looks like the one of 'DROP':

array_with_values.FIRST(n) → array

GROUP(): 'GROUP' is a basic method, that helps to, as the name says, distinguish values and group them according to some parameters, given in the second array. The syntax is notated as follows:

array_with_values.GROUP(array_with_group_parameters) → array

The 'GROUP'-method takes also two arrays as input and returns one array as output. The first array simply contains some values, which will be grouped by the method. The second array has the same length as the first one and contains parameters (arbitrary values), that refer to the values in the first array at the same position. All the values of the first array, that have the same parameter in the second array, are put in the same group. These groups are represented by the parameters of the second values, which serve as keys in the return array. Every key has one or more assigned values, so that the return array is an array of hashes, where the parameters of the second input array are the keys and the values of the hashed are the values of the first input array.

SINGLE(): This method takes place, when an element of an array is needed. The given array must have only one element remaining and if that is the case, 'SINGLE' will return that element. This might be achieved by using 'FIRST(1)' too, but there is a slightly different semantic. 'SINGLE()' will return only the element, while 'FIRST(1)' will return an array, containing the first element. That is an important difference, that can't be neglected. As this method takes no argument the syntax is simply:

array_with_one_element.SINGLE() → single array element

SORT(): The 'SORT'-method is a helper method, that supports sorting of values, according to an ordering, which is generated by a predicate. The predicate however is computed somewhere else but in this method. The syntax of this method is:

array_with_values.SORT(array_with_ordering) → array

As the other methods explained above, the SORT-method has two input arrays and returns one array as output. The first array contains the values and is the array, that is being sorted. The second array holds some numbers, representing the evaluation of a predicate for the sorting of the values, given in the first array. As the second array has the same length as the first array, every entry in the second array represents a number and this number stands for a position (of the value in the first array) in the finally sorted array, the values of the first input array are sorted by their corresponding numbers of input array two. The output array returns the sorted version of the first input array.

The normalizable methods

In this section the methods, contained in the module 'Enumerable', shall be explained. [TFH09] [FM08] This methods are the ones, that ought to be normalized to get a easier possibility to translate them into relational algebra for database support. In addition the replacement rules concerning the normalization are illustrated. The AST of the source code has to be modified in a way, so that the new nodes, representing the replacement rules, are accordingly created and added to the syntax tree.

all?(): The 'all?'-method tests for a given collection, that implements the 'Enumerable'-module, if a predicate (computed by evaluating the block) is satisfied for all members of the collection. If that is the case, the method returns true, otherwise it will return false. The predicate deciding this, is given by the block. If however no block is given, an implicit block will be created with the syntax `|obj| obj`, meaning 'all?' will only return a true value, if none of the objects in the collection is 'nil' or 'false'. The exact syntax of 'all?' is:

$$enum.all? \langle \{ |obj| block \} \rangle \rightarrow true \text{ or } false$$

To avoid having a block using 'all?', the method can be normalized using the 'map'-method, because, as the goals of the normalization say, 'map' may have a block. So the replacement of the notation above without a block in the 'all?'-method but in the 'map'-method would be:

$$enum.map \{ |obj| block \}.all?$$

any?(): The difference between the 'all?'-method, described above, and this method is very poor. Both methods have a similar syntax. The syntax of the 'any?'-method differs only in the method name:

$$enum.any? \langle \{ |obj| block \} \rangle \rightarrow true \text{ or } false$$

The semantical difference is that 'any?' tests, if a predicate is satisfied for one member of the collection, it is called on. This is done by computing the block. Again if no block is given, an implicit block is created similar to the 'all?'-method. Depending on how the

predicate is evaluated, the return value is 'true' or 'false'. The replacement rule to avoid a block in the 'any'-method is analogical to the rule for 'all?', shown above:

```
enum.map {|obj| block }.all?
```

collect(): The 'collect'-method is a synonym to the 'map'-method. It provides a similar syntax and semantic and therefore is normalized by the 'map'-method.

```
enum.collect {|obj| block } → array or enumerator
```

For more information, watch the explanation concerning the 'map'-method further above. So the normalization of this method is simply:

```
enum.map {|obj| block }
```

count(): The 'count'-method counts, as the name says, elements of a given collection. It provides three different types. The easiest one, without an argument, counts all elements in the collection and needs not to be normalized. Then there is a type, that takes an object as argument, in this case 'count' will return the number of elements, that match the argument. The third type takes a block and returns the number of elements, for which the block returns a 'true'-value. The syntax of the two different types, having to be normalized, is:

```
enum.count(obj) → int  
enum.count {|obj| block } → int
```

As there are two ways to express a different behavior of the 'count'-method, there must be also two replacement rules. One, that avoids using a block and one, that avoids using an argument. The type, having no argument, has not to be normalized and will be used to express the replacement rules of the two other possibilities to use 'count'. The first rule shown, is the one for the type with argument, the second one refers to the type with a block

```
enum.FILTER(enum.map {|parameter| parameter == obj}).count  
enum.FILTER(enum.map {|obj| block}).count
```

detect(): This method passes each element of the collection, it is called on, to the block and returns the first one, for which the block evaluates to 'true'. If none of the elements satisfies the predicate, specified in the block, 'nil' is returned. If no block is given, calling the method, an enumerator will be returned. The syntax of this method is noted as follows:

enum.detect { |obj| block } → obj or nil or enumerator

To normalize the method, the predicate has to be evaluated and from all results, that fulfill its qualifications, the first one has to be selected, done using the 'first'-method, which will be mentioned later in this chapter. That may be done like this:

enum.FILTER(enum.map { |obj| block }).FIRST(1).SINGLE()

drop(): The 'drop'-method in Ruby has the same syntax and semantic as the basic 'DROP'-method described above, so it will not be explained in more detail. The Ruby syntax is:

enum.drop(n) → array

Due to that the replacement is also very simple. Instead of creating an own new rule using 'map' or other methods, it is simply possible to just use the basic method 'DROP':

enum.DROP(n)

find(): The 'find'-method is another way of writing a 'detect'-method and is just a synonym to this method. Therefore the syntax is exactly the same, except the method name, and is only displayed for the sake of consistency:

enum.find { |obj| block } → obj or nil or enumerator

According to the fact, that 'find' represents a synonym and has also the same syntax, the replacement rule is very simple. Only the method name has to be replaced:

enum.detect { |obj| block }

find_all(): This method has a lot in common with the 'find' or 'detect'-method. Again a collection is searched for elements, that satisfy a predicate represented by the block of the method. The difference is, that not only the first object for which the predicate is 'true', but an array with all the elements, matching the predicate, will be returned. In case of no given block, an enumerator is returned. The syntax of 'find_all' is:

enum.find_all { |obj| block } → array or enumerator

The replacement rule has to take care, that all objects, for which the block is 'true', will be filtered out and returned in an array. That is possible by using the 'FILTER'-method and the 'map'-method in the following way:

enum.FILTER(enum.map { |obj| block })

first(): Similar to the basic 'FIRST'-method this method returns the first 'n' elements of a collection, if an argument is given and the first element, if no argument is specified. The syntax therefore is simple:

```
enum.first → obj or nil
enum.first(n) → array
```

As replacement for the version with an argument, it is obvious, that the the basic 'FIRST'-method should be used. Because there is also a version that does not take an argument, that type has to be normalized the way, that it fits in the scheme with an argument, but provides the semantic of the version without an argument. The difference of this two versions is, that 'first()' returns the first element of a given collection, while 'first(1)' returns an array with the first element of a collection. That is an important difference and so the first rule normalizes the type without argument, by using the second rule and the primitive helper method 'SINGLE', whereas the second rule should be clear. It just using the corresponding basic method:

```
enum.FIRST(1).SINGLE()
enum.FIRST(n)
```

group_by(): 'group_by' is a method, taking the input 'enum' and partitions that collection. A hash will be returned, where the keys are the results of the operation, computed by the block, and the values, belonging to a key, are the items, for which the key is generated. If there is no block specified, an enumerator object is returned. Syntactical the 'group_by'-method looks like:

```
enum.group_by {|item| block } → hash or enumerator
```

To replace this method accordingly to the goals already specified, another basic method is used, the 'GROUP'-method. Combined with 'map', the replacement has the same semantical behavior. The replacement rules is:

```
enum.GROUP(enum.map {|item| block})
```

include?(): This method is used to check, whether an object belongs to a collection or not. So a 'true'-value is returned, if that is the case and a 'false'-value is returned otherwise. The method's syntax is:

```
enum.include?(obj) → true or false
```

To normalize this method, meaning to avoid the argument, a little bit longer and more complicated replacement rule, using three methods has to be applied:

enum.FILTER(enum.map {|parameter| parameter == obj}).any?

last(): 'last' is a very simple method to explain. It has two forms: one without an argument, which will just return the last element of a collection and additionally a form, that takes an argument 'count' (a number) and returns the last 'count' elements of the collection. If the collection is empty and has no elements, the first version returns 'nil' and the second one returns an empty array as result. The syntax of this two forms is:

enum.last → *obj* or *nil*
enum.last(count) → *array*

When 'last' is normalized, there have to be of course two replacement rules, one for each version of the syntactical form. As it is not quite clear, if the normalized version of this method is used on an array or a range, the replacement rules for the second version, illustrated above, differ. There must be an other replacement rule for a range, because a range does not have a 'length'-method. So the first two replacement rules shown, are the ones for 'last' without arguments, which will transform the version without an argument into a version with an argument, by using the third and fourth rule and the primitive helper method 'SINGLE'. That is necessary, because a 'last'-method, called without argument returns an element, while 'last' with argument returns an array, even if the array has only one element. The other two rules refer to the version with an argument, whereas the second rule handles arrays and the third rule normalizes the method, when it is called on a range:

enum.DROP(enum.length - 1).SINGLE()
enum.DROP(enum.end - enum.begin).SINGLE()
enum.DROP(enum.length - count)
enum.DROP((enum.end - enum.begin) - (count - 1))

member?(): 'member?' is a synonym to the 'include?'-method, meaning it has the same semantic. So it will not be explained any further. The syntax is:

enum.member?(obj) → *true* or *false*

As this method is a synonym to another method, that has already a replacement rule and is able to be normalized, the replacement rule for this method replaces simply the name of the method by the name of the method (in this case 'include?'), which has already a normalization rule. So the replacement rule for 'member?' is:

enum.include?(obj)

none?(): The method 'none?' takes every member of the data collection 'enum' and passes it to the block. If the block returns 'false' or 'nil' for all elements, then 'none?'

will return a 'true' value, otherwise 'false' will be returned. In the case, that no block is given an implicit block similar to the 'all?'- and 'any?'-method is generated. The syntax is also similar to that of those methods:

$$enum.none? \langle \{ |obj| block \} \rangle \rightarrow true \text{ or } false$$

As the replacement is done in methods like 'all?' or 'any?', the predicate is evaluated by using the 'map'-method with a block and then calling 'none?' on the result. That leads to the following replacement rule:

$$enum.map \{ |obj| block \}.none?$$

one?(): In this method the block computes also a predicate, like it is in the method explained above. But in this case, it is tested, if exactly the predicate represented by the block matches exactly one time for one arbitrary object of the collection. If that is truly the case, the method will return 'true'. In any other case 'false' will be the return value. When no block is given, an implicit block is created as well as in the section above. Therefore the method syntactically is nearly identical to the 'none?'-method:

$$enum.none? \langle \{ |obj| block \} \rangle \rightarrow true \text{ or } false$$

Because the syntax and the semantic of this method and the method described above differ only very little, the replacement rule is accordingly similar:

$$enum.map \{ |obj| block \}.one?$$

partition(): The return value of 'partition' is an array of two arrays. This is the case, because the method splits the members of the collection, it is called on, in two arrays. The first array contains all elements, for which the block returns 'true' and all other elements (the block evaluates to 'false') are placed in the second array. If no block is written, when the method is called, then an enumerator object is returned. The syntax is:

$$enum.partition \{ |obj| block \} \rightarrow [true_array, false_array] \text{ or } enumerator$$

To replace the block of this method, it is important to look at the semantic of the method. As two arrays are computed and returned as elements of one array, it is possible to evaluate every one of the two array separately. So the first array can be represented by a method, that returns all elements, for which the block turns into 'true' in an array and the second one may be a method, that drops all those values and returns all remaining values in another array. Finally the results of those two methods have to be put into an array. There are two methods, that provide the searched semantic, the 'find.all'- and the 'reject'-method ('reject' will be explained further below). That results in the following replacement rules. The first one is the replacement with the insertion of the two meth-

ods, which both have their own normalization rules. The second one normalizes that representation, according to the replacement rules the 'find_all' and 'replace'-method have, to avoid the blocks the methods are using:

```
[enum.find_all {|obj| block }, enum.reject {|obj| block }]  
[enum.FILTER(enum.map {|obj| block}), enum.FILTER(enum.map {|obj| not (block)})]
```

reject(): This method builds the opposite of the 'find_all'-method. So an array of every element, for which the block is 'false' is returned. In other words to get a better connection to the method's name; every element, for which the block returns a 'true'-value is rejected. The syntax is simple and has just a 'standard' block. If this block is omitted, an enumerator is returned instead of an array:

```
enum.reject {|obj| block } → array or enumerator
```

As already said: if this method has the same predicate as the 'find_all'-method, it returns the opposite of that method. Therefore it is very easy to get a replacement rule for the method, if a rule for 'find_all' is already present. Negating the predicate will suffice to get the correct replacement:

```
enum.FILTER(enum.map {|obj| not(block)})
```

select(): The 'select'-method is another synonym for a method provided by Ruby. In this case the synonym is 'find_all'. So the semantic will not be explained any further and the syntax is, except of the name, exactly the same as the syntax of the 'find_all'-method:

```
enum.select {|obj| block } → array
```

Because this method is just a synonym for 'find_all', the replacement is very simple. The method is replaced by its synonym method:

```
enum.find_all {|obj| block }
```

sort_by(): 'sort_by' sorts the collection, given by *enum*, according to keys, which are generated by evaluating the members by the block. The return values of the block are used to compare the elements for their ordering. The syntax looks like:

```
enum.sort_by {|obj| block } → array
```

To normalize this method a basic method has to be used, the 'SORT'-method. This method does the sorting, while the 'map'-method, shown in the replacement rule, computes the keys, which represent the position of an element by evaluating the block:

```
enum.SORT(enum.map {|obj| block})
```

4.3.2 Adjusting the class tree

Now that all methods, which have to be normalized, are introduced and their replacement rules are explained, it is time to show, how the replacement takes actually place.

At first the class tree has to be traversed node by node and it has to be checked, whether a node represents a method call, that has to be normalized, according to the rules shown above. So when such a method is found in the tree, the normalization process starts. As every method has a different replacement rule, it is necessary to treat every method separately. For every method to be replaced, which was found in the class tree, a new node has to be created, replacing the old node, that was representing the method, having to be transformed to suffice the normalization goals. But a single new node does not provide all the information the entire replacement rules inherits. So that node consists actually of multiple nested class objects and must therefore be created with several children/subnodes ensuring, that the entire new node has a semantically correct representation of the corresponding replacement rule. By doing this another problem occurs. For some methods a predicate (mostly represented by the block) has to be evaluated, that has also to be checked in the new normalized version of the method. Looking at the example in the next section the 'select'-call is replaced by a 'map'-call. But the predicate, that is evaluated, remains the same. The problem is, that this predicate can be of arbitrary length and can hold other methods, that have to be normalized as well. The handling of the normalization has to start therefore in the deepest method call, meaning a recursive analysis and treatment of the class tree has to be done. That's why there must be a mechanism, taking the subpart of the class tree, that represents the predicate and appends it to the new created node. Appending the subtrees in this way satisfies the task of a recursive tree analysis. It must be at the right place of course: exactly at the place of the new created method-node, that holds the predicate of that method. It has to be taken care not to create a class tree with wrong semantic, compared to the original source code. So for every method, that is replaced, an own handling has to be established at its own new node has to be created and inserted at the appropriate position in the tree. But all that is explained more detailed below, when the methods of the example are replaced.

4.3.3 Implementation: the example

Now that all multiple block variables have been replaced appropriately, Listing 4.1 can be used as new source code to this working step: the normalization of the methods. As the normalization goals prescribe, only the 'map'-method is allowed to have a block left. However there are two methods except 'map', that have a block in the new source code. That are the 'group_by'-method in line one and the 'select'-method in line two. To recapture the replacement rules of those two methods shortly, they are once more noted here:

```
'group_by': enum.GROUP(enum.map {|item| block})
```

```
'select': enum.FILTER(enum.map {|obj| block})
```

Both methods need one of the basic helper methods to be normalized and both methods use the 'map'-method to evaluate the predicate of their origin form. Fortunately those two methods are placed in different parts of the tree, separated by a 'map'-method. Therefore they are not nested in each other and so they can be handled independent of each other. Let have a look at the 'group_by'-method:

The replacement start as follows. At first the tree has to be traversed and searched until a 'method_add_block'-object is found, which has the name 'group_by'. Then it is clear, that this node has to be replaced. So a new node has to be created, having the exact same semantic as the replacement rule says. When it is spoken about creating a new node, that means of course new objects have to be created in the class tree, as the a node corresponds to one ore more objects. It is just easier to speak from nodes, because they can be visualized by a tree. Anyway, this new node consists of two methods: the 'GROUP'-method, which represents an 'method_add_arg'-object and the 'map'-method representing an 'method_add_block'-object. For both methods a new class object has to be created. At first the object for the 'map'-method is created, because it has to be included in the 'argument'-child of the object, that is created for 'GROUP'. While creating this objects, it is important, that informations of the origin node, that has to be replaced are saved. This informations are needed when the new object(s) are created. For example the 'GROUP'-node needs to know, on which object it is called (in this case 'Lineitems') and the 'map'-method needs even more information, also the object, it is called on, but also the block operations from the origin node to evaluate the predicate. That building and composing of those subparts to the final new node, that is included in tree, has to be done very carefully to not pass wrong semantical informations to the tree.

At the end of this working step the old origin node referencing an 'method_add_block'-object with name 'group_by' is replaced by the new created node: an 'method_add_arg'-method with the name 'GROUP', that has a 'method_add_block'-method as arguments, which has the name 'map'. For the second method to be normalized the replacement works nearly identically to that of the 'group_by'-method, so it is not explained any deeper. The results of these replacements can be seen in the new source code displayed in Listing 4.2 or in the appropriate class tree visualization, presented in Figure 4.3.3. The new created nodes are shown by bold print. Therefore it is easily to see, where new nodes/class objects must be created and inserted and where subparts of the old class tree simply can be appended.

```
1 Lineitems.GROUP(Lineitems.map { |li| li.l_orderkey }).map {  
2   |x| [x[0], x[1].FILTER(x[1].map { |li| li.l_shipmode == "TRUCK" } )] }
```

Listing 4.2: Source code of the example with replaced blocks and methods

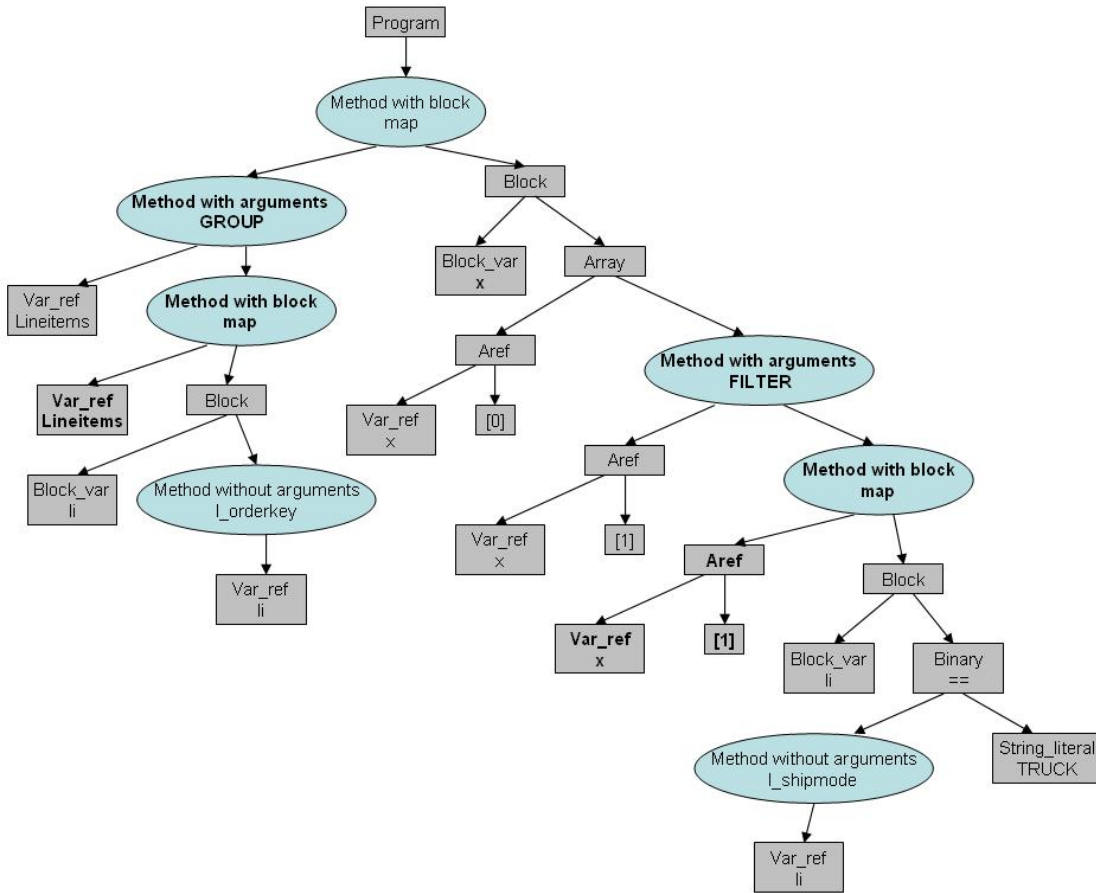
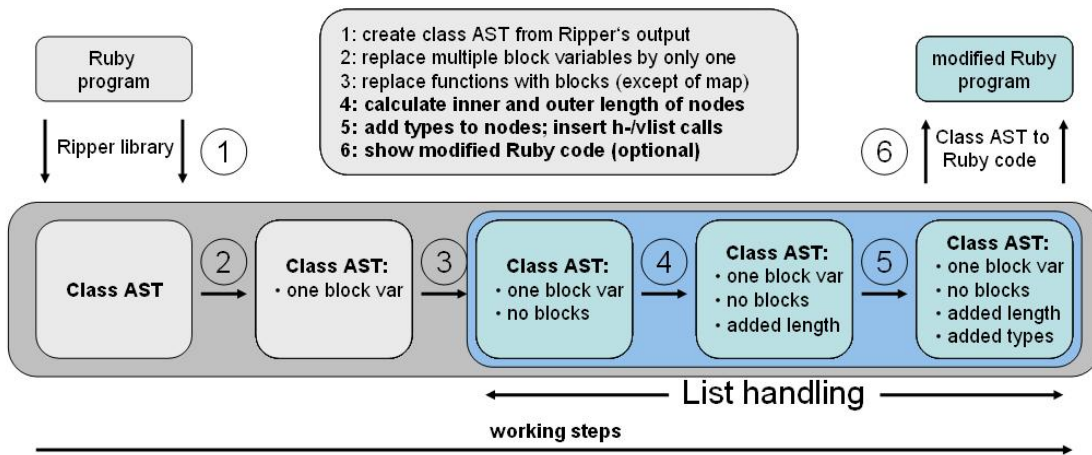


Figure 4.2: Class tree with replaced blocks and methods

5 Handling Ruby's lists relationally



In Ruby lists are represented otherwise, than they are implemented in a database or in relational algebra. So for a translation from Ruby into relational algebra, there has to be a special handling of all those lists to enable a suitable translation. The handling can be parted into two working steps, from which the first one concerns the length evaluation of the lists and the second one refers to the type of the lists. This chapter will now explain, why that two working steps are necessary and afterwards describe the working steps themselves.

5.1 Motivation

To get an impression, why Ruby's lists must be treated separately for being able to translate Ruby source code into relational algebra, the representation of the lists has to be regarded. The key feature is, that Ruby knows no tuples. If a tuple like structure has to be used in Ruby, a list must be taken to simulate the tuple. This fact leads to a problem. It is impossible to distinguish a list, holding tuples, from a list, that actually represents a list of other lists.

```
1 tuples = [ ["A", "Alice"], ["B", "Bob"], ["C", "Chloe"], ["D", "David"] ]
```

Listing 5.1: Nested array with tuples

To visualize that problem Listing 5.1 shows an array, containing four tuples. Every tuple consists of a capital letter and a name, starting with the according letter. In the Ruby source code, this four tuples are not visualized as tuples but as arrays with two elements. So the tuples are simulated by arrays, which have all the same length.

```
1 lists = [ ["Alice", "Al", "Anne"], ["Bob", "Bill"],
2          ["Chloe", "Casper", "Chuck", "Charlie"], ["David"] ]
```

Listing 5.2: Nested array with lists

In Listing 5.2 the other part of the problem is shown. In this case the illustrated nested array represents an array of arrays. The inner arrays are holding a list of names, of which each name starts with the same letter. These lists may have arbitrary lengths, what will result in another problem. To understand this, the goals of translating Ruby into relational algebra have to be looked at once more. In relational algebra and in databases tuples are a possibility to represent data. Normally a list is encoded vertically in relational algebra, but in combination with tuples, a list of tuples can be encoded very easily as vertical list of horizontally encoded tuples. Figure 5.1 shows the data from Listing 5.1 encoded exactly after this principle.

horizontally encoded inner arrays

vertically encoded outer array	A	Alice
	B	Bob
	C	Chloe
	D	David

Figure 5.1: Listing 5.1 encoded horizontally

A translation from Ruby's style of the tuple representation into the relational way is simple, because the tuples have the same length. But what happens to a list of lists? In this case a horizontal encoding is not possible in that easy way, because the lengths of the single lists can be arbitrary and therefore unknown. An encoding in relational algebra is only possible by using a vertical encoding for the outer array and the elements, representing the inner arrays. Figure 5.1 illustrates that encoding.

As the two pictures show, the horizontal encoding is much simpler, than a double vertical encoding. Therefore methods of a database, using the horizontally encoding, work in some cases faster, than a method with the same semantic, which takes vertically encoded data as input. Of course it is desirable to use always the faster method and so

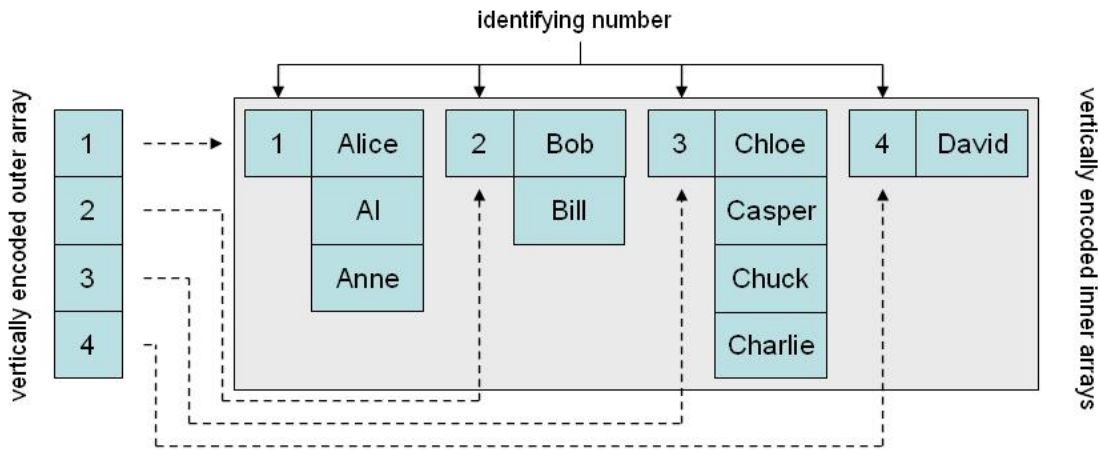
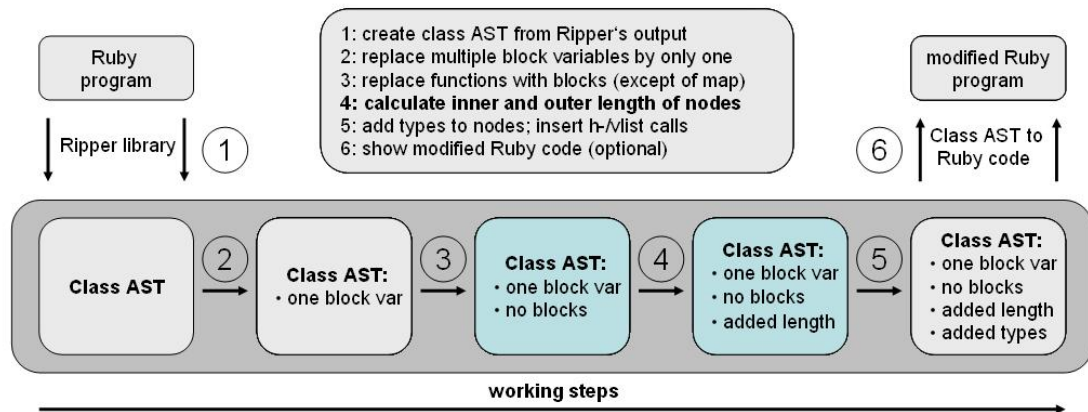


Figure 5.2: Listing 5.2 encoded vertically

the encoding normally should be a horizontal one. In other case however, such a method might only work on a vertically encoded list. That's why there must be a mechanism, that is able to convert the encoding from a horizontal one into a vertical one and of course the other way round too.

As it is impossible to predict in Ruby, if nested lists semantically represent a list of tuples or a list of lists, a directly translation into a horizontal encoding is not possible. To enable this, the data has to be analyzed and the outer and inner length has to be retrieved. Only with that additional information a suitable encoding can be found. If tuples are represented, a horizontal encoding is possible and if not the vertical encoding has to be used. Additionally the encoding could be changed to provide the best performance to a method of a database, depending on whether a horizontally encoded list or a vertically encoded one is needed or, in the case both encodings are possible, which encoding leads to a better performance. That is where the following two working steps come into play. The evaluation of the lengths is needed to change the encoding of nested lists, whereas the types are needed to know, which encoding is underlying. Details of changing the encoding of a list and of evaluating types and lengths shall be spared, as this section should only describe the reason for their implementation. The details will follow in the sections below.

5.2 Inferring collection lengths



In this section a very important step towards a final translation into relational algebra and further on to database support is done. As already mentioned in the motivation of this chapter, it is necessary for translating methods into relational algebra to know the inner and outer length of potential container items, as arrays and lists. How this step of adding the appropriate inner and outer length to a node is realized, is explained in the following sections below.

5.2.1 Inner and outer length

An AST node, that represents an container object, may have two different length types. The outer and the inner length, depending on how the object is defined.

Every object, being a container, has an outer length. The outer length specifies the entire length of that container object, meaning the count of all elements. If for example the outer length of an array is searched, that length can be imagined as the results of the array methods `.size()` or `.length()`.

It is also possible to specify an inner length for container objects. The inner length is the length of the elements, contained in the objects. If once again an array is taken as container object, the inner length of that array refers to the length of the subarrays, that represent the elements of the outer array. So an array 'x', that has elements of a type, like an array 'y', has the inner length `y.size()` or `y.length()`.

The values, that can be assigned to an AST node, may have three different types. At first there is of course the possibility, that both lengths are set to a specific value. But also other types of lengths are possible. If a node represents an object, that has no inner length, that length is set to nil, for the fact, that a length does not exist. On the other hand, there must be a way to express, that a length is actually present (not nil), but it is impossible to predict something about the length itself. Despite of the fact, that actually this length may not have to be truly infinite, it must be set to infinite, as no more closely specifying is possible. If, that is the case, it plays no role, how great the

length is, because for database support the exact lengths have to be known. So it is indifferent, whether the length is set to infinite or an approximate value, because precise values are needed. To simplify the presentation of the tree, in those cases infinite is chosen to represent an unpredictable length.

5.2.2 Calculate the lengths

With the information what the inner and outer length stand for, it is now possible to explain, how that lengths are computed. For that purpose, the class tree has to be traversed and the lengths have to be evaluated for every node. Therefore the tree has to be analyzed recursively bottom up, meaning from the leafs to the root. This is done, because every node, lying further down in the tree structure, influences the length of the nodes further up. The leafs represent the end of a subtree. The leafs are therefore the first nodes, that can be analyzed, because their length is not affected by other nodes below. This kind of nodes is discussed in Subsection 5.2.2. Calculating the lengths of an node, that is no leaf, meaning it lays somewhere in the tree between leafs and root, is more complicated. In that case the inner and outer length of the node depend on the kind of node, that is evaluated. For example a node, representing a method, creates output, depending on the method it is. So the output and the lengths are affected by the kind of method. Similar considerations have to be made for other nodes as well. The length evaluations of those nodes are presented in Subsection 5.2.2. All nodes discussed here refer to symbols, that have been explained in Section 3.1. Only three of them are not handled, because it is not necessary: The length of a 'program'-node is not important for the translation into relational algebra and the 'rest_params'-node and the 'command_call'-node do not occur any more due to normalization. But before the explanation of this working step can be done, a very important thing must be introduced first: the length environment of variables.

The length environment

The length environment is introduced due to the problem, that references may occur in the source code as well as in the appropriate class tree. A reference points only to a variable and gets the information about it this way. In the case of the class tree, that is however not possible. The objects in it have no relations except their children and parent, but definitively no references. The actual problem is now: when a reference to a variable is used, that reference-node needs the length information from the origin variable. As this information is not available, the lengths remain the default values of nil, what will result in wrong lengths for the tree. Therefore a mechanism has to be thought of, to provide the information of a variable to all references on it. The solution is to create a length environment, at which every new introduced variable is registered by its name and its according lengths. To realize that, a data structure has to be maintained, containing all variables and their assigned lengths. Every time a variable is created or overwritten its name and lengths have to be stored in the data structure, because a new length environment is created by the source code. If that is done, it means for every

reference-node, that it has to check the data structure for the variable, being referenced. If the variable is contained, the corresponding lengths for the variable are known and can be passed to the reference-node. When nodes, altering the length environment or needing a environment look up, are explained, this section will be clearer.

The leaf nodes

At first it has to be clear, that an inner or an outer length is not possible or sensible for every node. For every node, not representing an container object, it is difficult to specify a length of this node. As example think about an 'integer'-node. The outer length could be set to one, but what about the inner length? It can not be specified. So for this cases the lengths, that can not be specified, because they don't exist or are senseless, are not calculated and left to nil. Nil means, that there is no inner or outer length due to the art of the node. On the other hand, there can be data structures, that are not familiar, meaning, that there can be no statement about their lengths. For example self defined data structures as the container object 'Lineitems' in the example, that represents something like a table of known length. It is however not possible to know how many items this table has. So the inner length can be declared, but not the outer one. In that case the corresponding lengths are set to infinite, because an estimate has always be set to the top end. Infinite says, that there is actually a length, but it can not be exactly specified for it is unknown. The potential leaf nodes are listed below. For the purpose of a shorter and more vivid tree, they are summarized with other nodes, for example an 'identifier'-node and a 'var_ref'-node are shown in the class tree as one node. Anyway the leaf nodes are evaluated separately and so all nodes, that can be a leaf node are listed. Rules, defining the lengths, are not written down, because they are too easy and obvious, so they don't have to be presented.

1. *Constant*

This node refers to a constant value and has its appropriate lengths, if they are known. If not, the lengths are set to infinite, because a 'constant' could be representing some arbitrary data structure.

2. *Identifier*

An 'identifier' gives only a name to another object, therefore no statement can be made about the lengths and they are set to infinite. They could also be set to nil, but that makes actually no difference, because the node above takes other lengths, depending on what object is named by the identifier and as an identifier could name a container object, so setting the lengths to infinite seems more sensitive.

3. *Integer*

The lengths of the 'integer'-node are both set to nil, because an integer can't represent a container object.

4. *String_content*

The content of a string has nothing to do with a container object in the case, it is

interesting for the purpose of this thesis, so both lengths are set to nil. A string can be looked of course as container of chars, but in this case, that is not necessary.

5. *String_literal*

For representing a node, lying directly above the 'string_content'-node, a 'String_literal' refers to a kind of string. That's why both lengths are set to nil.

6. *T_String_content*

Another way a string is represented in the Ripper's AST. The lengths are accordingly nil.

7. *Symbol*

A symbol also has nothing to do with a container object, so the lengths are nil as well.

8. *Symbol_literal*

This node describes a 'symbol'-node more precisely and has for that reason the lengths nil too.

The embedded nodes

In this subsection nodes, being no leafs, get the analysis, that is necessary to calculate their lengths. As said before, that depends on the kind of node. So every node is listed below with its different features, that lead to the resulting inner and outer length.

Some of those nodes don't actually represent an container object directly and so they take the lengths of the tree nodes below them to not interrupt the recursive length evaluation of the tree. They are called *the simple nodes* for there is no really evaluation of the lengths. If there is a rule, which defines the lengths, it is noted below the explanation of how the lengths are computed. They are handled in 5.2.2.

Other nodes represent container objects, operate on them or are in some other cases more complicated. So the analysis of their lengths is more difficult. They are named *complex nodes*. In deed not every node, explained, is that complex, so the name does not fit perfectly. It shall only show the difference between these nodes and *the simple nodes*. But that will be clearer, when these different types of embedded nodes are discussed and their rules, defining the lengths, are shown below in 5.2.2:

The simple nodes

1. *Args_add_block*

'Args_add_block' describes the arguments of a method. Due to the normalization, it is ensured, that methods have only one argument left, and if, that is the case, the inner and outer length may be set simply to the lengths of that argument.

2. *Arg_paren*

This node is treated simple, because the parentheses, that hold arguments for

example of a method, don't represent an container object. So the lengths are taken from the argument (there can only be one), lying in the parentheses, because the parentheses are not important for evaluating the lengths and the argument in the parentheses is handled elsewhere.

3. *Brace_block*

This node can be ignored, while calculating the lengths, as its only semantic meaning is, that a block is following. That block is parted into the block variable and the content of the block, which are handled directly, when a 'method_add_block'-node is treated. So it is possible to let the lengths remain to the default value nil.

4. *Params*

The 'params'-node has only one parameter left due to normalization and that parameter is handled by the block_var-node described further below. Therefore it's lengths can be left to the default value nil.

5. *Paren*

This node behaves similar to the 'arg_paren'-node, as it represents parentheses. The lengths are therefore taken from the node below in tree.

6. *Unary*

As a unary operation, described by this node, does not alter the lengths of another node, it can simply take the inner and outer length of the node lying below in the class tree.

The complex nodes To understand the replacement rules better, a short explanation about them is given. If an expression is defined like this: $e :: (o, i)$, that means, that the expression e has the outer length o and the inner length i . *inf* represents a infinite length.

1. *Aref*

An array reference accesses elements of an array. As one single element of the array is represented by this node, the outer length of this node must be the length of that element. For that reason the outer length of the 'aref'-node is set to the inner length of the array, the node is referencing, because the inner length of an array is exactly the length of an element of an array. The inner length however can't be specified, for there is no information, how the elements look inside. So the inner length must be set to infinite.

$$\frac{\Gamma \vdash e :: (o, i)}{\Gamma \vdash e[n] :: (i, inf)}$$

2. *Array*

When the node, that is investigated, is an array, then the inner length refers to the number of objects, that are put in it to form one element. By using the array

constructor '[]', that lengths may differ, because the elements need not to have the same type, meaning the inner length must be set to infinite. Only if the lengths of all elements are the same, the inner length can be specified precisely and set to this length, like it is shown in rule two. How many of those elements are actually contained in the array, created by this constructor is known in the class tree, what results in an outer length of the count of the elements.

$$\frac{\Gamma \vdash e_1 :: (o_1, i_1), \dots, \Gamma \vdash e_n :: (o_n, i_n)}{\Gamma \vdash [e_1, \dots, e_n] :: (n, inf)}$$

$$\frac{\Gamma \vdash e_1 :: (o_1, i_1), \dots, e_n :: (o_n, i_n), i_1 = \dots = i_n}{\Gamma \vdash [e_1, \dots, e_n] :: (n, i_1)}$$

3. Assign

An assignment is one of the most difficult nodes to understand. As the name says, a value is assigned to a variable. But also a length is assigned to a variable. Because that variable could represent a leaf, a problem occurs. Normally the lengths of a leaf can be obtained by the node itself, that forms the leaf, like it is done above. But in this case that leaf gets new lengths assigned. So the variable, introduced newly, can't be handled the way a 'normal' leaf is. The 'assign'-node has to pass the lengths, that come from the right hand side to the new variable and override the old or default values. It also has to ensure that other nodes, referencing the new variable, are able to know the new assigned lengths of the variable. That's where the data structure, containing the length environment, comes into play. The new variable and its lengths have to be stored there, so that every reference to the new variable is able to look up the according lengths. To finally come back to the lengths of the 'assign'-node, they are rather simple, because the lengths are exactly the ones, that are provided from the right hand side of the assignment, meaning from the value the is assigned to the variable.

$$\frac{\Gamma \vdash e_1 :: (o_1, i_1), \Gamma + \{v :: (o_1, i_1)\} \vdash e_2 :: (o_2, i_2)}{\Gamma \vdash v = e_1; e_2 :: (o_2, i_2)}$$

4. Binary

In the case of a 'binary'-node the outer length of that node depends on the operator used in the node. The rules listed below refer only to the behavior, when arrays are part of the binary expression. If two atomic values are located on the left and right side of the binary expression, the behavior should be clear. As no container objects are the result, the inner length can be set to nil and the outer length to one, to reach a better readability of the tree. To come back to the case arrays are part of the expression: if the operator is a '+', then it has to be checked, what the outer lengths of the underlying nodes are. If one of them is infinite, the outer length of the 'binary'-node is infinite as well. If the outer length of the left hand or right hand side of the binary expression is nil, then the outer length can set to the outer length of the other side, no matter what that length is. But if both outer

lengths are known values, the resulting outer length is the sum of the outer lengths of the both sides. If a '-' operator is found, an array is returned, that represents the difference of the two arrays, meaning elements that lie in the second array are removed from the first one. For the reason it is impossible to know how many elements are removed, the outer length must be set to infinite. The inner length, concerning both operators, is anyway set to infinite, because it is impossible to make a statement about it, except the inner lengths of both children are the same. When a '*' operator is found and one child is an integer, the array is repeated as often as the value of the integer says. So the count of the elements of the array is multiplied by the integer value, resulting in an also multiplied outer length. The inner length remains the same. If the '==' operator is used for comparing, then the result of the binary expression is a boolean value, meaning, that the outer length can be set to one and the inner one is nil. In every rule both expressions e_1 and e_2 are arrays. Only in rule five, referring to the '*' operator, the second expression n represents an integer. This restrictions are not shown in the rules, but have to be fulfilled to apply the rules.

$$\frac{\Gamma \vdash e_1 :: (o_1, i_1), \Gamma \vdash e_2 :: (o_2, i_2)}{\Gamma \vdash e_1 + e_2 :: (o_1 + o_2, inf)}$$

$$\frac{\Gamma \vdash e_1 :: (o_1, i_1), \Gamma \vdash e_2 :: (o_2, i_2), i_1 = i_2}{\Gamma \vdash e_1 + e_2 :: (o_1 + o_2, i_1)}$$

$$\frac{\Gamma \vdash e_1 :: (o_1, i_1), \Gamma \vdash e_2 :: (o_2, i_2)}{\Gamma \vdash e_1 - e_2 :: (inf, inf)}$$

$$\frac{\Gamma \vdash e_1 :: (o_1, i_1), \Gamma \vdash e_2 :: (o_2, i_2), i_1 = i_2}{\Gamma \vdash e_1 - e_2 :: (inf, i_1)}$$

$$\frac{\Gamma \vdash e_1 :: (o, i)}{\Gamma \vdash e_1 * n :: (o * n, i)}$$

$$\frac{\Gamma \vdash e_1 :: (o_1, i_1), \Gamma \vdash e_2 :: (o_2, i_2)}{\Gamma \vdash e_1 == e_2 :: (1, nil)}$$

5. *Block_var*

The lengths of the 'block_var'-node refer to the lengths, that the method 'map' takes as input (Remember that 'map' is the only method, that has a block left due to normalization). This lengths can only be known, when a 'method_add_block'-node is analyzed. So they are specified by the 'map'-method and directly passed to the 'var_ref'-object, that specifies the block parameter (this is described, when the implementation of the lengths of a 'method_add_block'-node is explained). The 'block_var'-node must simply take the lengths of that var_ref-object, as it has the same lengths and refers to the 'var_ref'-node in the class tree. To guarantee, that

this works the 'block_var'-node is treated as some kind of leaf node, because no further recursive call to the method, that is traversing the class tree and calculating the lengths, must be done. All nodes, lying below the 'block_var'-node are handled separately. It must be ensured, that the 'params'-node gets the same lengths as the 'block_var'-node itself, in particular the lengths, that have been assigned to the 'var_ref'-node by the corresponding 'map'-method call. Also it is important, that the new block variable and its fitting lengths are added to the data structure, introduced in Section 5.2.2 and holding the length environment. This is necessary because a block variable is always created newly and there is no way of adding a length to that node like it is done, when a leaf node gets its lengths. If that is the fact, all references to the new block variable are able to look up its lengths in the data structure. Otherwise those references would not be able to know, what the lengths of the referenced block variable are.

6. Call

As the 'call'-nodes differ in behavior, depending on which method is called, every of that different methods has to be treated separately. Despite the fact, that 'call'-node specify 'method_add_arg'- and 'method_add_block'-nodes, this 'call'-nodes, following the method-nodes are ignored, because the treatment of the method-nodes is done directly, when their according nodes are handled. Simply because a 'call'-node does not provide the entire information, that is necessary to evaluate the lengths of a method-node correctly. The methods, that actually represent calls (which are named 'method without argument' in the visualization of the class tree) and that are used in the context of this thesis, are listed below:

a) *all?*

As the 'all?'-method without arguments returns a boolean value, the outer length is one and the inner length is set to nil, for it does not exist. The outer length could be set to nil too, due to the fact a boolean value is not a list, but it makes the class tree more readable, when it is set to one.

$$\frac{\Gamma \vdash e :: (o, i)}{\Gamma \vdash e.all? :: (1, nil)}$$

b) *any?*

'any?' behaves syntactically equally as the 'all?'-method, so its lengths are accordingly one for the outer length and nil for the inner length.

$$\frac{\Gamma \vdash e :: (o, i)}{\Gamma \vdash e.any? :: (1, nil)}$$

c) *count*

'count', as the name says, returns the count of elements of a container object. So it is easy to provide the correct inner and outer length. The inner length

is set to nil, as 'count' return an integer, representing the number of elements in the container object, and the outer length can be set to one, because the integer returned is consists of one value, it could also be set to nil, because an integer has actually no length, but if it is set to one, it gets more clear, what the node does, when looking at the class tree.

$$\frac{\Gamma \vdash e :: (o, i)}{\Gamma \vdash e.count :: (1, nil)}$$

d) *length*

The 'length'-method behaves quite similar to the 'count'-method and its lengths are similar too. The outer length is one and the inner length is nil.

$$\frac{\Gamma \vdash e :: (o, i)}{\Gamma \vdash e.length :: (1, nil)}$$

e) *none?*

'none?' has the same semantic as 'all?' has, therefore the outer length is one and the inner length is nil.

$$\frac{\Gamma \vdash e :: (o, i)}{\Gamma \vdash e.none? :: (1, nil)}$$

f) *one?*

'one?' has also the same semantic as 'all?', so the outer length is one and the inner length is nil.

$$\frac{\Gamma \vdash e :: (o, i)}{\Gamma \vdash e.one? :: (1, nil)}$$

g) *SINGLE*

As this method returns the only element of a container object, the inner length of that container object represents the outer length of this node, while the inner length of this node can't be specified and must be set to infinite.

$$\frac{\Gamma \vdash e :: (o, i)}{\Gamma \vdash e.SINGLE :: (i, inf)}$$

h) *size*

Being a synonym for *length()*, 'size' has also the same lengths: an outer length of one and an inner length of nil.

$$\frac{\Gamma \vdash e :: (o, i)}{\Gamma \vdash e.size :: (1, nil)}$$

i) *sum*

As many of the other methods in this section, 'sum' returns only one value: the sum of the elements in a container object. Therefore its outer length is accordingly one and its inner length is nil.

$$\frac{\Gamma \vdash e :: (o, i)}{\Gamma \vdash e.sum :: (1, nil)}$$

j) *to_a*

This method converts non array container objects (e.g. hashes) into arrays. Therefore its lengths are simply the lengths of the container object before the conversion, which are provided by the nodes further below in the class tree.

$$\frac{\Gamma \vdash e :: (o, i)}{\Gamma \vdash e.to_a :: (o, i)}$$

k) *uniq*

As 'uniq' returns a new array, in which all duplicate items are removed, it is impossible to say, what outer length that new array has. So the outer length must be set to infinite. The inner length remains the same as the one from the object, that called the method, for 'uniq' does not alter the elements, it only shrinks the length of the input container object.

$$\frac{\Gamma \vdash e :: (o, i)}{\Gamma \vdash e.uniq :: (inf, i)}$$

l) *values*

The 'values'-method takes a hash as input and returns only its values as array. The keys are left away. As there is no possibility to know, how the values look like and particularly how their inner length is, the inner length must be set to infinite. As all values are returned the count of that values corresponds to the count of the key-value-pairs, resulting in an outer length, that is equal to the length of the hash.

$$\frac{\Gamma \vdash e :: (o, i)}{\Gamma \vdash e.values :: (o, inf)}$$

7. Condition

For a 'condition'-node it is easy to set the lengths. Because it is not safe, which part of the condition is actually executed in the program, both lengths have to be set to infinite. If however both parts of the condition provide the same lengths, the lengths are set to the length of one of those parts.

$$\frac{\Gamma \vdash e_2 :: (o_2, i_2), \Gamma \vdash e_3 :: (o_3, i_3)}{\Gamma \vdash ife_1thene_2elsee_3 :: (inf, inf)}$$

$$\frac{\Gamma \vdash e_2 :: (o_2, i_2), \Gamma \vdash e_3 :: (o_3, i_3), o_2 = o_3, i_2 = i_3}{\Gamma \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 :: (o_2, i_2)}$$

$$\frac{\Gamma \vdash e :: (o, i)}{\Gamma \vdash e.\text{values} :: (o, \text{inf})}$$

8. *Dot2*

'Dot2' refers to a range, what results in an inner length, that can't be specified and is set to nil and in an outer length, that has the same value as the range has elements.

$$\frac{x, y \in \mathbb{Z}}{\Gamma \vdash x..y :: (y - x, \text{inf})}$$

9. *Method_add_arg*

All methods, that have arguments, are represented by a 'method_add_arg'-node. Due to the normalization goal, that the only methods, allowed to have still an argument, are the primitive helper methods just that methods can occur as 'method_add_arg'-node. For the reason of being different, returning different values and therefore having different lengths for its results, every method must be treated otherwise. Similar to all methods is, that at first the lengths of the object, calling the method, have to be known, before the resulting lengths can be computed. If the caller of the method is a reference, there has to be a check, if that reference is already bound to a length, by searching the data structure, that holds the information about the variable environment. If the lengths are known, the analysis of the method and the calculation of the lengths can begin. The following methods are used in the scope of this thesis:

a) *DROP()*

'DROP' shrinks a container object as an array by the number 'n' of elements, that is passed as argument. So the length of the resulting array is the origin length of the object the method is called on minus 'n'. The inner length remains the same as the one from the container object, as the element structure is not affected by this method.

$$\frac{\Gamma \vdash e :: (o, i), n \in \mathbb{N}}{\Gamma \vdash e.\text{DROP}(n) :: (o - n, i)}$$

b) *FILTER()*

The 'FILTER'-method filters, like the name says, elements of the container object, it is called on, according to the boolean list, given in the argument. As it is impossible to predict, how many elements remain in the input container object and are not filtered out, the outer length must be set to infinite, as it

can not be specified precisely. The inner length however remains the same as on the input array, because the structure of the elements is not touched.

$$\frac{\Gamma \vdash e_1 :: (o_1, i_1), \Gamma \vdash e_2 :: (o_2, i_2), o_1 = o_2}{\Gamma \vdash e_1.FILTER(e_2) :: (inf, i_1)}$$

c) *FIRST()*

This method returns the first 'n' elements of a container object, therefore the outer length can only be 'n'. The inner length has the same value as the inner length of the container object, because 'FIRST()' does not alter any element, but limits only their count.

$$\frac{\Gamma \vdash e :: (o, i), n \in \mathbb{N}}{\Gamma \vdash e.FIRST(n) :: (n, i)}$$

d) *GROUP()*

'GROUP' realizes a grouping of the elements of the container object, is called, by the list passed as argument. The outer length must be infinite, because a prediction of how many different elements are contained in the argument list is impossible. The inner length by contrast can be specified exactly. It must be two, as an array of key-value-pairs is returned.

$$\frac{\Gamma \vdash e_1 :: (o_1, i_1), \Gamma \vdash e_2 :: (o_2, i_2), o_1 = o_2}{\Gamma \vdash e_1.GROUP(e_2) :: (inf, 2)}$$

e) *SORT()*

Because the 'SORT'-method only changes the order of the elements of a container object, the lengths are simply the lengths of the sorted container element, because there is no change made to the input container object, that affects the lengths of that object.

$$\frac{\Gamma \vdash e_1 :: (o_1, i_1), \Gamma \vdash e_2 :: (o_2, i_2), o_1 = o_2}{\Gamma \vdash e_1.SORT(e_2) :: (o_1, i_1)}$$

10. *Method_add_block*

As a 'method_add_block'-node, due to normalization, is just possible with the 'map'-method, only that method has to be handled. The outer length is simply to calculate. The block is evaluated once for every element of the input container element, so the outer length has to be the same as the outer length of the input container element. The inner length is a little bit more complicated. Every element of the new created array by the 'map'-method has the value, that is computed by its block. Therefore the lengths of the block have to be known first, what is ensured by the recursive kind of traversing the class tree. If that lengths are finally known, the inner length of the 'method_add_block'-node is set to the outer length of the block, because the block provides the elements for the output array

of the 'map'-method. So that output array consists of elements with the outer length of the block, what results in an output array with an inner length, specified by the elements. To guarantee, that the recursive traversing of the class tree is possible, the program part handling the 'map'-method has to pass the lengths of its caller to the block parameter, as it was mentioned above in the explanation, concerning the 'block_var'-node. This is important, because otherwise the subtree holding the block has not the necessary informations about the lengths of its input, it needs to calculate the length of itself correctly. Recapture the fact explained in Section 4.2.1 that a block takes every element of the container object, it belongs to, once as input. So the outer length of the block parameter has to be assign with the inner length of the 'map'-method caller, as that length represents the length of the items. The inner length of the block parameter is not known and must be set to infinite.

$$\frac{\Gamma \vdash e_1 :: (o_1, i_1), \Gamma + \{v :: (i_1, inf)\} \vdash e_2 :: (o_2, i_2)}{\Gamma \vdash e_1.map\{|v| e_2\} :: (o_1, i_2)}$$

11. *Var_field*

A 'var_field'-node represents always a new introduced variable to the variable environment, that is already known. For that reason a value has to be assigned to this new variable, before it can be used. There the 'assign'-node comes into play. An assignment is always positioned higher in the class tree than a 'var_field'. Despite the fact a 'var_field'-node represents a leaf, it can not be treated like the ones described above. As a 'var_field' introduces a new variable, this new variable must get its length somewhere from above in the tree. In this case from the 'assign'-node. That node takes fully care of adding the new variable and its lengths to the known variable environment and also adds the appropriate lengths to it. Viewed in this light, a 'var_field'-node could also be a simple note, because its lengths are implemented elsewhere, but for the reason the new variable occurs in this node and so the necessity of adding it to the length environment, it is put in this section. Only the handling of that problem is translocated to the 'assign'-node.

12. *Var_ref*

Because a 'var_ref' references a variable, that might be added to the length environment (for example a new variable introduced by a 'var_field') the first thing to do, when a variable reference occurs in the class tree, is to look up in the data structure, which holds the length environment, if the referenced variable is contained. If that is the case the outer and inner length are set to the according values, stored in the data structure. If however the variable referenced is not found in the data structure, a leaf-node must be lying below the 'var_ref'-node, that gets its lengths the way, described further in this chapter. If actually the case occurs, that the variable is not found in the data structure nor the lengths can be evaluated by using leaf handling, an unknown variable is referenced, meaning the source code is faulty. That can be possible, if the source code given to the Ripper library was

not yet checked, because Ripper simply analyzes the syntax not the semantic of a program, hence the fault could not be noticed.

$$\overline{\{\dots, v :: (o, i), \dots\}} \vdash v :: (o, i)$$

5.2.3 Implementation: the example

In this section the computing of the lengths is explained, using the example. Every node of the class tree, visualized in Figure 5.2.3, shows now its corresponding lengths. For a short explanation of the presentation: 'out:' refers to the outer length, 'inn:' to the inner length and 'inf' stands for infinite. Now the steps, that are taken to implement all lengths to all nodes, are explained like they occur, when the analysis is done.

When looking at the tree, the actually implementation starts at the 'GROUP'-node, because there the first leaf is found. As the tree lengths are calculated bottom-up, the implementation must always begin on a leaf. If the node has no leaf as child, the traversing is pushed further down in the tree, by calling the length-handling-method recursively. This leaf however occurs as object, the 'GROUP'-method is called on. So the leaf is handled and for the reason, that 'Lineitems' refers to a table-like structure with 16 columns, the inner length of that node must be 16, while the outer length can't be specified and is set to infinite. (In Ruby implementation every element of 'Lineitems' represents an array with 16 elements). This information has to be stored in the data structure, that contains the environment informations. As the 'GROUP'-node has a child, that is no leaf and has no length information yet, because it has not been handled by now and the the 'GROUP'-node needs that information, the tree is stepped down, by another recursive call of the length-handling-method. Another method-node is found, a 'map'-node. That node is called by the same reference to 'Lineitems' as the 'GROUP'-method was. The lengths can be looked up in the data structure, holding the variable informations, so the 'var_ref'-node gets the same lengths. Then the tree is traveled down, because the 'map'-node has a child, that was not handled yet and is no leaf. That child represents a block. As said, when the handling of the block variable was described, the outer length of the 'block_var'-node refers to the inner length of the container object the block (by the method) is executed on. That results in an outer length of 16 and a unknown inner length, that can't be specified. Before going on, the block variable has to be added to the length environment, to ensure its lengths can be obtained, when it is referenced.

The next step is to evaluate the block itself. It contains a call to the method 'l_orderkey' on the block variable 'li'. The leaf node, represented by the 'var_ref'-node, has no information about the lengths of 'li', so it has to look it up in the data structure, holding the length environment. There the information about the lengths is saved and the 'var_ref'-node gets them. The method ('l_orderkey') above that leaf, refers to one element of the container element referenced by 'li'. It takes that one and returns it to the block above. The length is therefore one for the outer length and nil for the inner length. Nil for the inner length, because of the method itself: it returns only one value

(the 'orderkey') and therefore it is safe, that there is no inner length. Now having found out, what the block returns, the lengths of the 'map'-method above can be finished. Because it is impossible to know how much elements are contained in the container object 'Lineitems', the resulting outer length for the 'map'-node must be infinite, the inner length however is represented by the outer length of the block, that is one. Another step further up this information can be used by the 'GROUP'-method call. For not exactly knowing how many 'orderkeys' there are, (represented by the infinite outer length of the 'map'-node) the outer length of the 'GROUP'-node has to be infinite as well. The inner node in contrast can be specified precisely. It must be two, because 'GROUP' always produces key-value-pair and those have the length two.

Now that the lengths of the container object are evaluated, that the 'map'-method, closest to the root, has and is called on, the lengths of the second part of the tree can be handled. At first the block variable 'x' has to be treated. That is done like it was in the left hand side of the tree, so the block variable has the outer length two, the inner length infinite and is also registered in the length environment. Then the content of the block of the 'map'-method is analyzed. There an 'array'-node is found, that consists of two elements. Accordingly the outer length must be two. The inner length refers to the lengths, provided by the elements. Normally that lengths would not be known already and the inner length would be handled, when they are known, but they will be infinite and so the inner length of the array is infinite too. Further down in the tree, the first element of the 'array'-node is an array reference on the block variable 'x' at the position zero. To get the lengths, the node below 'aref' must be evaluated first. It is a 'var_ref'-node, referring to the block variable 'x'. How that is done, was explained above, when the left side of the tree was traversed and analyzed. When an array reference occurs, it is not possible to say something about its inner length, but the outer length can be specified. It refers to the inner length of the container object, it is called on. As the array reference returns the first element of the array it references and the inner length of that array is infinite, both lengths of the array reference itself have to be infinite.

The next step is the handling of the second element of the 'array'-node. That node is a call to the method 'FILTER'. Because that node operates on an array reference, that has both lengths set to infinite, the lengths of the 'FILTER'-method must be infinite as well. The evaluation of the 'aref'-node is done similar, like it is described above and it shall not be explained twice. The argument of the 'FILTER'-node is another 'map'-method call. That method is called on an array reference with the lengths infinite, therefore its outer length must also be infinite. The inner length of the 'map'-method however is set by the block and that block has the block variable 'li', that has also both lengths set to infinite, as it operates on the same array reference the 'map'-method does. The content of the block, that actually influences the inner length of the 'map'-method call, consists of a binary expression. As that expression uses a comparison operator, it is already predictable, that the outer length of the 'binary'-node must be one and the inner length must be nil, because a boolean value is returned. Because the block returns that lengths, the inner length of the 'map'-method can be finally set to one. To complete the analysis of the tree, three more nodes have to be visited. The left side of the binary expression is a method call, that returns the 'shipmode' of a container object. That's why the outer

length is set to one and the inner length is set to nil. That method behaves similar to the 'l_orderkey'-method, described more detailed above. The 'var_ref'-node, lying below the 'l_shipmode'-method, is handled as every variable reference was so far, resulting in setting both lengths to infinite. The last node represents a string literal and can be treated as 'normal' leaf. As a string literal has no lengths, important for the purpose of this thesis, both length are set to nil.

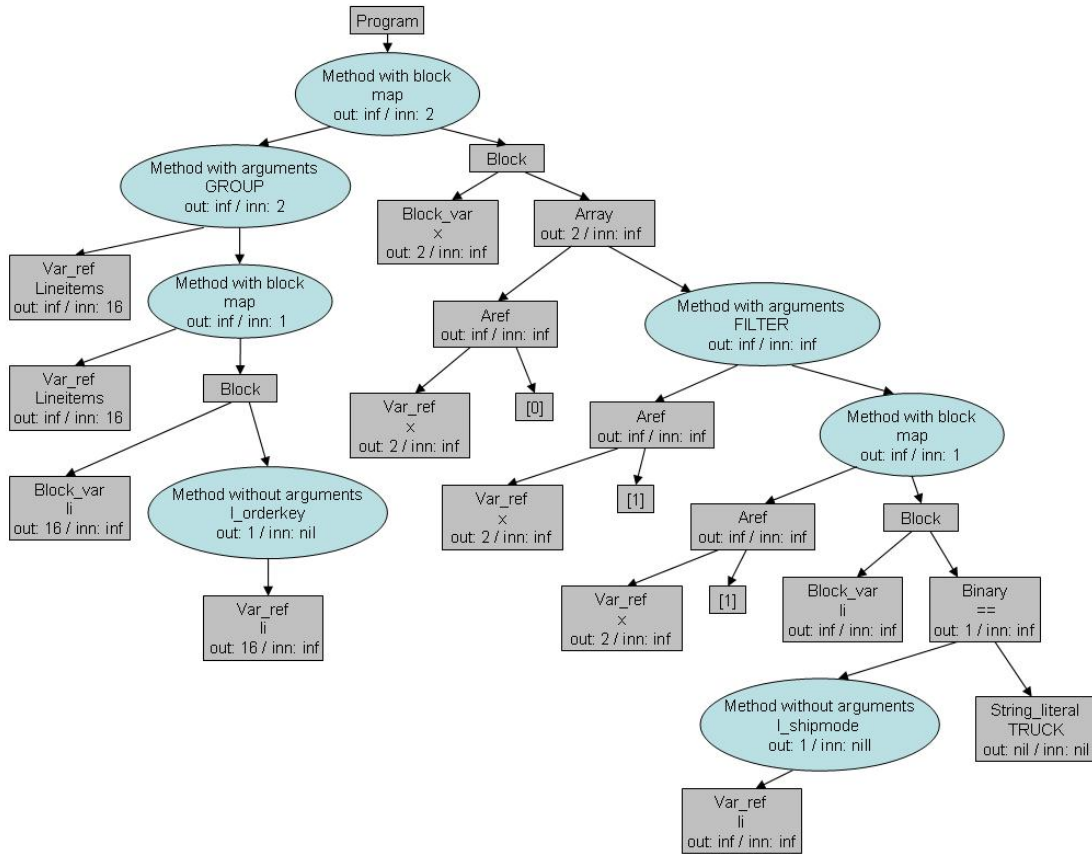
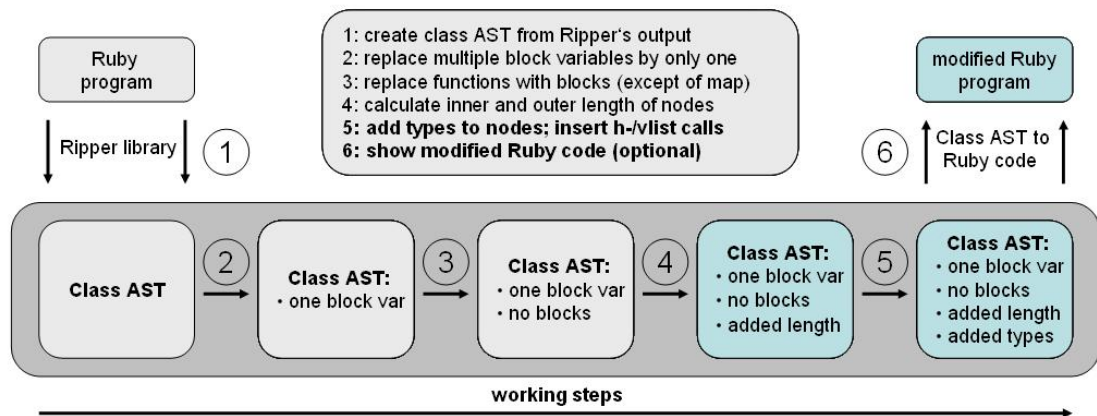


Figure 5.3: Class tree with evaluated lengths

5.3 Evaluating the types of the tree nodes



As explained in Section 5.1, a translation into relational algebra, that results in a fast and efficient database code, some methods have to be called on a certain encoding of a container object, otherwise the execution of the code would lead to a poor performance. To guarantee, that the underlying encoding matches the wanted encoding, types have to be added to the nodes and on some cases the encoding of the underlying lists has to be changed. How this is done, will be the content of this section.

5.3.1 The type system

Based on the motivation for this chapter, the types, the nodes are able to have, can be divided in three parts. There are the simple basic types like integers or strings and so on (atomic type), container objects, that are encoded vertically (vlist type) and container objects, that are encoded horizontally (hlist type).

The basic types are of the type 'atomic'. That means, they are representing no container object in the form of an array or a list. It would suffice to just give the type 'atomic' to all those nodes, which refer to a basic type to ensure the translation into relational algebra is possible and does not go wrong, but in order to get a better visualization, the basic types are referenced by '*Atomic(name_of_the_basic_type)*'. For example '*Atomic(Integer)*' would be the type of an integer-node.

A typically vertically encoded tree object would be an array, because if taking a table like and so database like representation of that array, it would result in a table with one column, that holds all the elements of the array. As this column extends itself vertically, when more objects are added, the array is encoded vertically. A vertically encoding makes only sense, when a container object is regarded. The representation of the type of that object would be '*Vlist(other_types)*' standing for '*vertical list*'. The 'other_types' might be any arbitrary number of other possible types, another 'Vlist', a 'Hlist' or an 'Atomic' type.

The last possible type is represented by a *'Hlist(other_type)'*, which stands for *'horizontal list'*. Similar as a *'Vlist'* it can hold any number of any acceptable other types. A *'Hlist'* refers also to a container object and makes otherwise no sense. It can be looked at as some kind of tuple. Unfortunately Ruby provides no possibility of representing tuples. From Ruby's point of view, that is not necessary, because of the dynamic type system. For the purpose of this thesis however a tuple structure must be present, because in the relational algebra and also in database code tuples may occur. So a *'Hlist'* simulates that kind of tuple structure.

5.3.2 Typecasting the class tree

Because the types and the cases, when they are used, have been introduced, the class tree can be traversed and the nodes can get their types. Similar as in the case the lengths were added, there has to be a method, that traverses the class tree recursively. The typecasting must also be done bottom up, meaning it starts at the leafs. The principle is similar to the one, used in the last chapter for getting the lengths: to climb as deeply as possible down the tree, analyze the leafs of the subtree and afterwards climb up again and handle every node who's children have been treated and have got their type already. To provide such a functionality a type-handling method has to be implemented, that calls itself recursively until a leaf is reached, evaluates the type of that leaf and then returns to the node lying above.

The type environment

Similar as it is explained in Section 5.2.2 for the lengths of the nodes, a type environment has to be created to ensure reference-nodes get the necessary informations, they need about the type of the referenced variable. Therefore another data structure has to be maintained, holding the type information in the form of the variable's name and their according type. The reason for creating such a type environment and the handling of it don't differ from the art explained in Section 5.2.2. So this shall not be explained once more.

The leaf nodes

For some of the leafs nodes, it is possible to predict the type easily according to the description listed below. Most of the types of those leaf nodes are obviously clear, but they shall anyway be listed for completion:

1. *Constant*

A *'constant'* refers to some kind of data structure, that has been defined earlier and must therefore be known by the programmer. So it must be possible to pass its appropriate type to the *'constant'*-node. That type however can be arbitrary, depending on how the data structure, represented by the *'constant'*, was defined. Very important is, that the *'constant'* and its according type have to be registered in the variable environment, that specifies the variable names and their according

types. So an entry, consisting of the name of the constant and its type, has to be added to the data structure, holding the type environment.

2. *Identifier*

Because an 'identifier' just names other objects, it gets no type. This kind of node is handled, when the node directly above the leaf is treated. The 'identifier'-node is accessed for the name of the node above and then that node can get its type.

3. *Integer*

The type of the 'integer'-node is obvious. It must be an 'atomic' type, as an 'integer' is self a build in type. So the type is simply and easily 'Atomic(Integer)'.

4. *String_content*

As easy as the type of an 'integer'-object, the type of this node must be 'Atomic(String)'.

5. *String_literal*

This node refers also to a string, therefore its type has also to be 'Atomic(String)'.

6. *T_String_content*

Yet another form a string is represented in the Ripper's AST. The type is accordingly 'Atomic(String)'.

7. *Symbol*

A 'symbol'-node is not evaluated in the tree. It gets no type, because the 'symbol_literal'-node, that lays always above the 'symbol'-node handles the type of that leaf. As the 'symbol_literal'-node represents in this case no leaf and is more complicated to explain too, it is, in opposite to the last chapter, handled in the next section.

For some leafs however, it is not so easy to get the type. That are leaf nodes, that are just a reference to a variable. It can not be known, what type the variable has. For this reason every new introduced variable has to be saved in a data structure, that holds the type environment. An entry consists of the name of the variable and its according type. When a reference occurs, it a look up in this data structure has to be made, what the type of the variable is, it is referencing, and use that type for itself. If the case occurs, that the referenced variable is not present in the data structure and its type can't be set with help from the list above, then the program source code must be wrong, because a variable is referenced, that never occurred before and therefore has not be added to the type environment data structure. This will be explained more precisely, when such variables, referencing nodes, are discussed in Section 5.3.2.

The embedded nodes

Evaluating the types of the embedded nodes is much more complicated, than getting the ones of the leafs, because depending on which nodes lay below the analyzed and what their type is, the type of the actual node is determined by that. There also must be

guaranteed, that in some nodes a type check of its children is done, because if that type check fails, the source code might be faulty. The type check has to ensure, that the input types from the children of a node, fit the restrictions the node has. What happens, when a type check goes wrong and what can be done to still get a correct source code, that can be translated into relational algebra, will be explained further below in Section 5.3.3. At first however the handling of the different nodes is explained. It is mentioned, when a type check is needed and what the type restrictions of the different nodes are. As it was the case, when the lengths were calculated in the last chapter, the types of every node, that might occur, will also be treated separately. Note that the 'rest_params'- and the 'command_call'-node don't occur any more and need no treatment. How the treatment of the other nodes is done however, will be explained and is listed below. The notation of the types in the typing rules is as follows. 'H()' represents a horizontally encoded list, 'V()' a vertically encoded list and 'A()' an atomic type. 't' represents an arbitrary type, that can be one of the three possible types 'H()', 'V()' or 'A()'. If for some reasons a typing rule is trivial, it is not shown.

1. *Aref*

As an 'aref'-node represents a reference to a variable, although the reference concerns an array, there must be a check in the type environment, if that variable is already bound to a type. If that is not the case, something must be wrong with the source code, as a variable is referenced, that is not defined before and an error message will be created. If however the variable referenced, is found and its type can be retrieved, the type of the variable has to be looked at carefully. Also the position of the array reference, given by the 'aref'-node has to be known, because the array reference has the same type the array has at that position. So the type of the array reference-node is the type of the element at the position, the array is accessed.

$$\frac{\Gamma \vdash e :: H(t_1, \dots, t_n)}{\Gamma \vdash e[n] :: t_n}$$

$$\frac{\Gamma \vdash e :: V(t_1, \dots, t_n)}{\Gamma \vdash e[n] :: t_n}$$

2. *Arg_parens*

Normally the parentheses, that enclose an argument, ought to have no type, but in the tree structure and due to normalization, ensuring only one argument to a method is possible, the type of the argument, lying in the parentheses, can be assigned to the 'arg_parens'-node. So this node needs not to be ignored, like the other occurring parentheses-node is, as will be explained later.

$$\frac{\Gamma \vdash e :: t}{\Gamma \vdash (e) :: t}$$

3. *Args_add_block*

To assign a fitting type to the class tree node, that is specifying the arguments of a method, is as easy as it was, when the 'arg_parens'-node above was handled. Again only one argument is possible to a method, and therefore the argument block, just can be set to the type of that single argument. As the typing rule is trivial, it is omitted.

4. *Array*

Because an array is encoded as vertical list, the type of an array must be a vertically encoded list too. The type of that list is determined by the elements the array has. Every element may have a different type, so the vertically encoded list, that represents the array, holds the information of the types, the different elements have. This gets clearer, if a just a simple array is regarded, who's elements have different types. The array would represent the vertically encoded list, while the different types of the elements represent the content of that list.

$$\frac{\Gamma \vdash e_1 :: t_1, \dots, \Gamma \vdash e_n :: t_n}{\Gamma \vdash [e_1, \dots, e_n] :: V(t_1, \dots, t_n)}$$

5. *Assign*

An assignment must, like it is done in the last chapter, create a new entry in the type environment data structure. The difference is, that now the type instead of the length is saved. The reason remains the same. A variable, that is not known already gets a new type. That type is assigned by the 'assign'-node. To ensure every reference to that new variable can get the necessary information about the type of the variable, it must be saved in the environment data structure for the types. The type of the assignment is however very simple to implement. It is the type, that comes from the right hand side of the assignment. The 'assign'-node takes one additional task too, it sets the type of the new created variable according to the type of itself. That would a 'var_field'-node and its type must not be handled, this is done by the 'assign'-node.

$$\frac{\Gamma \vdash e_1 :: t_1, \Gamma + \{v :: t_1\} \vdash e_2 :: t_2}{\Gamma \vdash v = e_1; e_2 :: t_2}$$

6. *Binary*

Before the type of a binary expression can be evaluated, it needs to be a type check done. Both children of the binary expression must have the same type, because otherwise the two children cannot be computed by using the binary operator of the node. If it is ensured, that both types of the children are the same, there are two possibilities for the type of the 'binary'-node. The first one is the type 'Atomic(Boolean)'. That type occurs, when the binary operator is used to compare the left and the right side of the expression. In this case a boolean value is returned. In any other case, the type of the node is the same type, than the one of the left child, because the result of the binary expression has the same type as the operands

and as both operands have the same type, it suffices to just set the type of the node to the type of the left child. Note, that only the cases of a binary expression, shown in the rules, where a container object is involved. This is done for the reason, that the typing rules of a binary expression using two atomic types, are obvious.

$$\frac{\otimes \in \{+, -, *\}, \Gamma \vdash e_1 :: t_1, \Gamma \vdash e_2 :: t_2, t_1 = t_2}{\Gamma \vdash e_1 \otimes e_2 :: t_1}$$

$$\frac{\Gamma \vdash e_1 :: t_1, \Gamma \vdash e_2 :: t_2, t_1 = t_2}{\Gamma \vdash e_1 == e_2 :: A(\text{bool})}$$

7. *Block_var*

This node represents the block variable of a method's block. Because there is only one method left, that may have a block, the 'map'-method, it is possible to handle the type of this node directly. When a 'method_add_block'-node occurs, it is ensured, that the type of the 'block_var'-node is set by the handling the 'map'-node and therefore no handling of the 'block_var'-node has to be done and its type can remain default at first, because it is set later.

8. *Brace_block*

A 'brace_block'-node represents the structure of the block of a method. Similar as it is done with the 'block_var'-node, the type remains unset, that means left to the default value. The setting of the type is taken over by the part of the implementation, referring to the 'method_add_block'-node.

9. *Call*

Because of the fact, that a 'call'-node represents all method calls, having no arguments or blocks, there must be a distinction concerning this node. Every method has other types, other restrictions and must be handled differently. Of course a call can refer to self written methods as well, but in that case the programmer needs to add a handler for those methods by himself. The methods described below, are the established ones, that occur relatively often:

a) *all?*

The 'all?'-method takes a vertically encoded list as input, meaning it is called on this list. That restriction must be fulfilled. If the type check is passed, the method returns a boolean value, what results in a node type of 'Atomic(Boolean)'.

$$\frac{\Gamma \vdash e :: V(t)}{\Gamma \vdash e.all? :: A(\text{bool})}$$

b) *any?*

As the 'any?'-method has the nearly the same syntax as the 'all?'-method,

its type check for a vertically encoded list as input and its resulting type of 'Atomic(Boolean)' are the same.

$$\frac{\Gamma \vdash e :: V(t)}{\Gamma \vdash e.any? :: A(bool)}$$

c) *count*

This method takes a list of elements and returns the count of the elements. In doing so, it plays no role whether that list is encoded horizontally or vertically. The input restriction just needs a list. The return value of this node is an integer, which results of course in the type 'Atomic(Integer)'.

$$\frac{\Gamma \vdash e :: V(t)}{\Gamma \vdash e.count :: A(int)}$$

$$\frac{\Gamma \vdash e :: H(t)}{\Gamma \vdash e.count :: A(int)}$$

d) *length*

Providing a similar syntax and semantic as the 'count'-method, the type restrictions to the input are the same too. A horizontally or a vertically encoded list can be the input. The actual type of this 'call'-node is 'Atomic(Integer)'.

$$\frac{\Gamma \vdash e :: V(t)}{\Gamma \vdash e.length :: A(int)}$$

$$\frac{\Gamma \vdash e :: H(t)}{\Gamma \vdash e.length :: A(int)}$$

e) *none?*

With a quite similar syntax and functionality as the 'all?'- or the 'any?'-method, the type check needs a vertically encoded list as input and the resulting type of the node is 'Atomic(boolean)'.

$$\frac{\Gamma \vdash e :: V(t)}{\Gamma \vdash e.none? :: A(bool)}$$

f) *one?*

For this method the same input restriction and resulting type are present, as for the three similar methods 'all?', 'any?' and 'none?'.

$$\frac{\Gamma \vdash e :: V(t)}{\Gamma \vdash e.one? :: A(bool)}$$

- g) *SINGLE* Because this method returns the only element of a container object, the type must be set to the type of the element. That can be done, if a vertically or horizontally encoded list of elements is underlying (in fact there is only one element, but that does not affect the encoding). In any case the type is evaluated accordingly to the following rules.

$$\frac{\Gamma \vdash e :: V(t)}{\Gamma \vdash e.SINGLE :: t}$$

$$\frac{\Gamma \vdash e :: H(t)}{\Gamma \vdash e.SINGLE :: t}$$

- h) *size*

As the 'size'-method is just a synonym to the 'length'-method, it's input restriction and type are identical to the ones of the 'length'-method and the typing rules are just shown for completeness.

$$\frac{\Gamma \vdash e :: V(t)}{\Gamma \vdash e.size :: A(int)}$$

$$\frac{\Gamma \vdash e :: H(t)}{\Gamma \vdash e.size :: A(int)}$$

- i) *sum*

The input to the 'sum'-method, requires a vertically encoded list. The resulting type of the node must be an atomic type, that has the same subtype, than the elements of the input list have.

$$\frac{\Gamma \vdash e :: V(t_1, \dots, t_n), t_1 = \dots = t_n}{\Gamma \vdash e.sum :: A(t_1)}$$

- j) *to_a*

'to_a' must be called on a hash list and returns an array, containing the same information. As the hash and the resulting array both need to provide the same information, their type must be identical too. Therefore the type, that forms the input restriction represents also the resulting type of the node. Because an hash is illustrated as a vertically encoded list of a horizontally encoded list, containing itself two arbitrary types, exactly that composed type is the needed type to fulfill the input restriction and the type of the node as well.

$$\frac{\Gamma \vdash e :: V(H(t_1, t_2))}{\Gamma \vdash e.to_a :: V(H(t_1, t_2))}$$

k) *uniq*

Due to the fact, that this method removes duplicate elements of a given input list, the type of the node remains the same type as the one of the input list. The type check can only be passed, when 'uniq' was called on a vertically encoded list.

$$\frac{\Gamma \vdash e :: V(t)}{\Gamma \vdash e.\text{uniq} :: V(t)}$$

l) *values*

A call to the 'values'-method makes only sense and is allowed, when the underlying container object, the method is called on, is a hash. As mentioned in the description of the 'to_a'-method a hash is represented by a vertically encoded list of a horizontally encoded list, that has two arbitrary types as members. This type therefore forms the input restriction. The return value of this method is an array holding all the values of the hash, not the keys. Because the values are specified by the second type of the horizontally encoded list, the type for this node must be a vertically encoded list of the values' type.

$$\frac{\Gamma \vdash e :: V(H(t_1, t_2))}{\Gamma \vdash e.\text{values} :: V(t_2)}$$

10. *Condition*

'Condition' is another node, that needs a type check before its type is evaluated. It is determined, that if there is a 'else'-clause, the 'then'- and the 'else'-clause must have got the same type, as it is not clear, which clause is actually performed. In case the type check is passed, the type of the 'condition'-node is set to the type of the 'then'-clause, because it is indifferent to use the type of the 'else'-clause, because the types must be the same. If only a 'then'-clause is present, then the type is set to 'unknown', because there can't be a prediction, whether the 'then'-clause is executed or not.

$$\frac{\frac{\Gamma \vdash e_2 :: t_2, \Gamma \vdash e_3 :: t_3, t_2 = t_3}{\Gamma \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 :: t_2}}{\Gamma \vdash e_2 :: t_2}}{\Gamma \vdash \text{if } e_1 \text{ then } e_2 :: \text{unknown}}$$

11. *Dot2*

Ranges are represented by the 'dot2'-node and a range represents a vertically encoded list of numbers. That numbers are in most cases integers, but might be of other types to. Anyway the type of a 'dot2'-node is a 'Vlist' of the atomic type, the numbers have.

$$\frac{\Gamma \vdash x :: t_1, \Gamma \vdash y :: t_2, t_1 = t_2, x, y \in \mathbb{Z}}{\Gamma \vdash x..y :: V(t_1)}$$

12. *Method_add_block*

This node refers always to the 'map'-method, another method is not possible any more, due to normalization. This node does not only set its own type, but takes care of all nodes, that belong to the 'map'-method, meaning every node, that represents some information about the block. But before the type of the node can be evaluated, a type check has to be done, because the 'map'-method needs a container object as input, that is represented as vertically encoded list. If 'map' is called on a data structure, encoded as a horizontal list, the failed type check has to be handled according Section 5.3.3. If it is called on an atomic value, the program must be wrong and an error message is created. Before the node actually gets its type, the block variable needs a type first. That type is already set, when a 'map'-method is treated, because only this node has the necessary information to do that. The type of the block variable is influenced by the type of the container object, the method is called on. The type must be the interior type of the 'Vlist' data structure, 'map' is called on. For example, if 'map' is called on a container object having the type 'Vlist(my_type)', then the type of the block variable must be set to 'my_type'. Assigning the correct type to the block variable is not sufficient, the new typed block variable must be registered at, that means saved in, the data structure, that holds the information about the type environment. Only when that step is done, the block can be evaluated the right way. So finally, if the type check is alright, the block variable is added to the type environment data structure and the type of the block is known, the 'method_add_block'-node can get its type. The type is an array of the type, returned by the block. Because an array is encoded as vertical list, the type is therefore a 'Vlist' of the block type.

$$\frac{\Gamma \vdash e_1 :: V(t_1), \Gamma + \{v :: t_1\} \vdash e_2 :: t_2}{\Gamma \vdash e_1.map\{v | e_2\} :: V(t_2)}$$

13. *Method_add_arg*

Only the primitive helper methods are allowed to have an argument left, at this state of the work. That is one of the normalization goals. Therefore only these methods have to be treated, when a 'method_add_arg'-node is found in the class tree by this working step. As every method provides a different functionality, they all have to be handled differently, because they have other input type requirements and produce a different output type. Despite the different input requirements, all methods have in common, that there must be a type check of the input, that has to be passed, to ensure the method can be translated correctly into relational algebra. The following explanations refer to the single methods:

a) *DROP*

'DROP' is a special case of a method, that takes an argument. Similar as the 'FIRST'-method, but different than the other three methods described in this section, it has two possible input restrictions, from which only one has to be satisfied. 'DROP' can take a vertically encoded list, it is called on, as input. This list can be a horizontally encoded list as well. If in both cases an

atomic value is passed as argument, that has the type 'Atomic(Integer)', the input restrictions are met, indifferent of passing a vertically or a horizontally encoded list as input. The difference is only, that in the case a vertically encoded list is passed, the type of the method evaluates to the same type, meaning a vertically encoded list as well. In the case a horizontally encoded list is underlying, the type of the 'DROP'-method will be a horizontally encoded list. That gets more clear, when the behavior and the functionality of this method are looked at. Just some elements are dropped, but the encoding of the container object, that hold these elements, remains the same.

$$\frac{\Gamma \vdash e_1 :: V(t_1), \Gamma \vdash e_2 = A(int)}{\Gamma \vdash e_1.DROP(e_2) :: V(t_1)}$$

$$\frac{\Gamma \vdash e_1 :: H(t_1), \Gamma \vdash e_2 = A(int)}{\Gamma \vdash e_1.DROP(e_2) :: H(t_1)}$$

b) *FILTER*

This method needs two vertically encoded lists as input. The first list refers to the container object, the method is called on and the second list is the container object, specified as argument. The second list must be of the type 'boolean' to provide a correct behavior of the method. If both input requirements are satisfied, the output type of this method is easy to specify: As 'FILTER' just shrinks the origin container object, by removing elements, that are filtered out, the type must remain the same as the one of the container object, the method is called on: a vertically encoded list.

$$\frac{\Gamma \vdash e_1 :: V(t_1), \Gamma \vdash e_2 = V(bool)}{\Gamma \vdash e_1.FILTER(e_2) :: V(t_1)}$$

c) *FIRST*

As this method does not differ greatly from the 'DROP'-method, what concerns the input restrictions and the resulting output type, it will be handled very shortly. Similar to 'DROP' it is possible to call this method on a vertically as well as a horizontally encoded list. The argument must be of the type 'Atomic(Integer)'. The resulting type is depending on the container object, the method is executed, a vertically or a horizontally encoded list.

$$\frac{\Gamma \vdash e_1 :: V(t_1), \Gamma \vdash e_2 = A(int)}{\Gamma \vdash e_1.FIRST(e_2) :: V(t_1)}$$

$$\frac{\Gamma \vdash e_1 :: H(t_1), \Gamma \vdash e_2 = A(int)}{\Gamma \vdash e_1.FIRST(e_2) :: H(t_1)}$$

d) *GROUP*

Like the 'FILTER'-method, the 'GROUP'-method needs two vertically encoded lists as input too: One list it is called on and one list as argument. If that type check is passed, the resulting type of the method can be implemented. The type is a vertical list of a horizontal list, meaning the type of the vertical list is another list, that is horizontally encoded. The content of the horizontally encoded list consists of two types. As a horizontal list represents some kind of tuple, that is possible. The first type is the type of the vertical list, that is passed as argument and the second type is the same as the type the 'GROUP'-method is called on: a vertical list of an arbitrary type (that arbitrary type has of course to be the same type as the one of the vertically encoded input list).

$$\frac{\Gamma \vdash e_1 :: V(t_1), \Gamma \vdash e_2 = V(t_2)}{\Gamma \vdash e_1.GROUP(e_2) :: V(H(t_2, V(t_1)))}$$

 e) *SORT*

'SORT' is a simple method for implementing a type. The type check needs, like the type check of the 'GROUP'- or the 'FILTER'-method, two vertically encoded lists as input. If the condition is fulfilled, that results in an output type of the method, that is similar to the one of the container object, the method is called on. This is obvious, because only the order of the elements of the container object is altered, but the type of that object is never affected and therefore remains the same.

$$\frac{\Gamma \vdash e_1 :: V(t_1), \Gamma \vdash e_2 = V(t_2)}{\Gamma \vdash e_1.SORT(e_2) :: V(t_1)}$$

 14. *Params*

Similar as the 'block_var'-node, this node refers to the block variable. For this reason its type is the same as the type the 'block_var'-node has. How that type is evaluated was described above, when the handling of the 'method_add_block'-node and so the 'map'-method was explained. The type of this node though remains default, until it is set by the part of the program treating the 'map'-method but is handled there like the following simple typing rule says.

$$\frac{\Gamma \vdash v :: t}{\Gamma \vdash |v| :: t}$$

 15. *Paren*

Parentheses don't have a type, so this node is not handled, when a 'paren'-node is discovered in the tree. Its type is not set and remains to the default value of nil. The fact, that no type is assigned, gets even more clear, when more than one object is lying in the parentheses. It is simply not possible to specify a type. However it must be ensured, that nodes lying directly above the 'paren'-node in the class

tree, don't refer to this node, but evaluate their types by using the objects placed in the parentheses.

16. *Program*

The type of the 'program'-node is determined to be the type of the last subtree, that is starting from this node. So the handling of the 'program'-node waits until the entire tree has its types and then itself is set to the type of the last subtree.

$$\frac{\Gamma \vdash e_1 :: t_1, \dots, \Gamma \vdash e_n = t_n}{\Gamma \vdash e_1; \dots; e_n :: t_n}$$

17. *Symbol_literal*

To explain why the 'symbol_literal'-node gets different types, depending on its name and not just an atomic type, is a little bit complicated. It refers to the fact, that a method can be passed as argument to a method, by specifying that argument method with a symbol literal construct. To take this fact into account, the type of the 'symbol_literal'-node is set to the type the method has, that is referenced by the symbol literal. That methods, pointed to by symbol literals, are normally written by the programmer and so their types are known and can be assigned to the 'symbol_literal'-node. This must of course be done by the programmer, that has written those methods. If for some reason a symbol literal is used otherwise and the name of it matches none of the method, addressable by the symbol literal construct, the type is set to 'unknown'.

18. *Unary*

To set the type of a 'unary'-node is very easy. It is simply the type of the child of that node. For the reason, that the type is not affected by the node itself, it must be the type of the node lying below.

$$\frac{\Gamma \vdash e :: t,}{\Gamma \vdash \otimes e :: t}$$

19. *Var_field*

Because a 'var_field'-node occurs only below an assignment, the handling of this node is completely done, when the assignment is treated. The type of this node is set and the new variable, created is also registered at the type environment and saved to the appropriate data structure.

20. *Var_ref*

A variable reference normally refers a variable, that must already be known to the variable environment. So in the data structure, that holds all introduced variable, there must be a lookup for the variable, referenced in the actual case. If that variable is found there, the variable reference takes the same type as the variable, found in the data structure, has got. If however the variable is not found, an error message is printed and the program is stopped, because there is a reference to a variable, that has not been declared. The only exception to this behavior is, when

a 'Constant'-node is referenced. As the 'Constant'-node stands for some self (by the programmer) created data objects, which types are known, it must be ensured, that if a 'constant'-node is referenced, the 'constant' and its type are registered to the type environment first, before the searching for the referenced variable starts.

$$\overline{\{\dots, v :: t, \dots\}} \vdash v :: t$$

5.3.3 Handling failed type checks

As described in the last section, there are a lot of type requirements, that have to be fulfilled to ensure a correct translation into relational algebra. But what happens, if for some reason such a type check goes wrong. Then the entire process of preparing the class tree to translation would be in vain. There should be a possibility to adapt the tree in a form, it was adapted and transformed all the way to the present point of work. That adaptation should ensure, that the type restrictions are met, by altering the class tree. But what altering would make sense? To solve this problem, the first attention has to be paid to the possible errors, occurring at the type check. The first error could be, that a list of elements, indifferent of being encoded vertically or horizontally, is required, but an atomic value is received. In this case something with the syntax of the underlying program has to be wrong and the program would not run faultlessly in Ruby. Anyway there would be no possibility to avoid the type checks error, as it is impossible to convert an atomic value into a list. If however the second potentially error occurs, something can be done to avoid this error. The error occurs for example, when a vertically encoded list is required as input, but a horizontally encoded list is passed. It also could be the other way round. The point is, that the requirement for a list encoded in a certain form is not met, but yet a list is passed as input. The solution to the error, occurring due to the type requirements, is to convert the encoding of the given list in a way, that it satisfies the type check. This is possible, because a list is underlying and just the form of the encoding is wrong. To change the encoding of a list, two functions have to be implemented, one that takes a vertically encoded list as input and returns a horizontally encoded one (*hlist()*), and the other changing a horizontal encoding into a vertical one (*vlist()*). It has to be guaranteed, that both functions just alter the form of the encoding. In no case a list with a different semantically meaning must be created. If both functions work properly the next problem can be faced. The source code of the analyzed program has to be changed. The function, needed of the two possible functions, has to be inserted at the position, where the type error has occurred. Placed there, the function ensures, that the encoding of the underlying list is altered the way, it meets the input restrictions of the node above. As the source code of the analyzed program is represented by the class, the class tree has to be modified. A new node has to be included, that stands for the '*hlist*'- or '*vlist*'-function call, which has the list that produced the type error as input. The result of that new node is the list with changed encoding and that is precisely what is needed at that position in the class tree. It is however not quite that easy as it sounds to create the new node, because what results in one new node in the visualization of the

tree (following in Section 5.3.4 are actually more nested nodes and it has to be taken care, that every of that nodes is placed exactly and gets its correct type and lengths. Anyway by implementing the two functions and inserting the appropriate nodes to the class tree, a guarantee can be made, that a translation into relational algebra fails not just to the case of an inappropriately encoded list.

5.3.4 Implementation: the example

The implementation of the working step, explained in this section, is parted in two sections. The first one refers to the implementation, where everything is done like it is described above, that means all restrictions are used in the way, presented in the last sections. The second section however introduces a new input restriction, that is normally not necessary, but used in this case to show, how the converting of lists and the insertion of the nodes, representing that conversion, is done.

The normal way

As said, this part of the implementation using the example follows exactly the way the descriptions and rules explained above recommend and is shown in Figure 5.3.4. Now it will be explained, how the typecasting of the tree works by traversing the entire tree and determining the types of the single nodes. The type analysis and implementation is implemented strictly recursively, that means the tree is handled bottom up. In the explanation in this section, sometimes that behavior is ignored to provide a better understanding. Therefore sometimes nodes are finished, meaning they got their type predicted, despite the fact, that other nodes, lying below, have not been treated. Anyway the handling starts bottom up, because the first nodes, where something can be said about their types, are always the leaves.

The leaf, that is met first is the 'var_ref'-node on the left side of the tree, that specifies the input list, the 'GROUP'-method is called on. This variable reference refers to the container construct 'Lineitems', which is defined by the programmer of the origin source code and therefore known to have the type $V(H(A(\text{Lineitems})))$. Just to recapture the visualization of the types: a 'V()' represents a 'Vlist', that means a vertically encoded list, a 'H()' stands for a 'Hlist' and so for a horizontally encoded list, while a 'A()' represents an atomic data type. The reason to shorten the type names is simply to make the tree more vivid, otherwise it would still be greater, than is is right now. Also due to this problem of the tree size, the type 'A(Lineitems)' occurs. As the lengths say, the inner length is normally 16, meaning the horizontally encoded list has 16 elements, but to mention and write down all of them in the class tree representation would cost by far too much space. So it is assumed, that the inner types of the table like data structure 'Lineitems' are known. Important for understanding the example is just, that there are two columns, one that can be referenced by the method 'l_shipmode' of the type string and the other of the type integer and accessible by the method 'l_orderkey'. Anyway, to come back to the first leaf node, its type can be set and also be added to the type environment.

The 'GROUP'-method node above knows so far, that the second element of the vertically encoded list it will return, has the same type as the just visited 'var_ref'-node. Because the second type still misses, the tree has to be climbed down to resolve that type. There a 'method_add_block'-node is found. This node has to ensure, that all other nodes, that have something to do with the block get their types set, because they are not able to do that by themselves. But before the block is treated, the type of the input list of the 'map'-method must be resolved. As that represents the same reference to 'Lineitems' as described earlier, the type is clear and is not explained again. Knowing that type, the type of block variable can now be set. The type must be 'H(A(Lineitems))', because every element of the vertically encoded list the 'map'-method is called on, is passed one time to the block, referenced by the block variable. All those elements have the type 'H(A(Lineitems))' and so that is also the type of the block variable. If the type of the block variable is set, it and its type must be added to the type environment. The reason should be clear, but can be obtained below once more. To finally get the type of the 'map'-method itself, the type of the block has to be maintained. To get this type, the next node, which represents a call to the 'L_orderkey'-method has to be evaluated. That method is called on the variable reference to the block variable 'li'. To obtain the type of that reference, a look up in the type environment has to be made. If the block variable had not been added to the type environment, the variable reference would have had no chance to make any statement about its type. That's why adding new introduced variables to the type environment is so important. Now that the type is known, the 'L_orderkey'-method node can be evaluated itself. Because the method is a self defined one, the behavior and the resulting type can be predicted. The method returns the 'orderkey' value of every tuple passed to it. Therefore the resulting type is 'A(int)', standing for 'Atomic(Integer)'.

With this information, the type of the 'map'-node, lying above finally can be set. The return value of the 'map'-method is always an array of the single elements evaluated by the block. As the block has the type 'A(int)', that resulting array must be a vertically encoded list of that type, 'V(A(Int))'. For the reason no more nodes under the 'map'-method must be treated, the handling of the nodes returns back to the 'GROUP'-node. Because it is known, by which input the grouping has to be done, the type of the 'GROUP'-method can be evaluated, according the rules to $V(H(A(Int), V(H(A(Lineitems))))))$. To understand that complex type structure, the following explanation is given. That type represents a list (vertically encoded), that holds tuples (horizontally encoded). That tuples consist of two elements. The first element is an integer representing the 'orderkey' and the second one illustrates a table like structure, that contains all elements of the 'Lineitems' data structure, that have the same 'orderkey' than the first element of the tuples has. In short words, the elements of 'Lineitems' are grouped by their 'orderkey'.

For the nodes on the left side of the class tree have now their types, the analysis goes on with the 'map'-node in the center of the tree. This node has the input type given by the 'GROUP'-method node, that was discussed at last. Depending on that type, the 'map'-node sets the type of the block variable to 'H(A(Int), V(H(A(Lineitems))))' and registers it at the type environment. This is not explained more detailed, because the handling of a 'map'-node has already been done once and the handling conforms always to the same

sents the type of the block of the 'map'-method above. Accordingly the type of the 'method_add_block'-node is a vertically encoded list of the block's type. This type is 'V(V(A(Int),V(H(A(Lineitems))))))'. Finally the 'program'-node can get its type as well, that is the type of the last statement in the program, meaning the type of the first node of the last tree branch, which is in this case, the central 'map'-node. So the return type of the program would be 'V(V(A(Int),V(H(A(Lineitems))))))'.

To finish the example, a few nodes have still to be handled. The remaining nodes, lying below the 'FILTER'-node, need to be treated as well. Normally, that would be done, before the nodes above are evaluated, but as it did not influence them in this case, they are treated afterwards. The rest of the class tree starts with another 'map'-method node. That node takes the type provided by an array reference node as input, that is 'V(H(A(Lineitems)))'. Therefore the block variable 'li' gets the type 'H(A(Lineitems))'. The actual block is presented by a binary expression, that compares its two children. So its type must be an boolean value, which is, as can be seen, in fact the case. The children of the 'binary'-node are a 'string_literal', that has of course the type string and a method 'l_shipmode', that similar to the method 'l_orderkey', described earlier, returns a single column of the tuple, represented by the variable reference to the block variable. That type is also a string. To complete the tree, the 'map'-method gets the type 'V(A(bool))', composed of the type of the binary expression and the fact, that a 'map'-method returns an array.

Added failed type check

When considering the analysis and setting of the types, described in the last section, one important thing was missing. Normally some nodes need a type check, as has been mentioned further above. But nowhere a type check was discussed in the example. The reason is, that all type checks were passed without any problems. Recapture the explanation of the class tree and its types, lying below the 'FILTER'-node, there easily can be seen, what is meant with the type checks. A 'FILTER'-method needs a vertically encoded list of boolean values to work properly, which is in deed provided. The children of binary expression further below in the tree must have the same type, when the operator compares them, that is the fact too. This way all necessary type checks evaluated to true and therefore no conversion of a list's encoding has to be done.

To show how that would work, if there was the need to do so, a new type restriction is implemented. This restriction normally is wrong and not needed, so note this is only done for presentation purposes of converting the encoding of a list and inserting new nodes. As the tree shown in Figure 5.3.4 only differs in the new nodes, inserted to handle the failed type check, just that step is discussed. The other steps of computing the types of the tree, were discussed in the last section.

The additional type restriction should affect the positional access to an array. That access should only possible, when a vertically encoded list is underlying. If however a horizontally encoded list is found, the type check fails. In the example all three array references access a horizontally encoded list. So all type checks fail and normally the program would be aborted and a error message would be printed. Due to the fact, that

it is possible to change the encoding of list, this does not need to happen. Everywhere, where a type check fails, that list must be treated by the 'vlist'-function. By inserting a node in the class tree (actually that are multiple nested nodes, but they are summarized to one node), that problem can be solved. It must be taken care, that the inserted node does no alter the semantic of the class tree, except of the added function call to 'vlist'. The inserted 'vlist'-nodes are highlighted in class tree visualization by the red color. If in front of the 'aref'-nodes, the 'vlist'-call is inserted, that node takes the horizontally encoded list as input and returns a vertically encoded one, with exactly the same semantics. As nothing has changed except of the encoding of the list, the program is able to run on. Because there is now a vertically encoded list accessed by the 'aref'-nodes, the type check restrictions are fulfilled and the analysis is able to continue without faults. Of course inserting the nodes into the class tree goes hand in hand, with altering the source code another time. The 'vlist'- or 'hlist'-call, whatever call is needed, must be visualized in the source code as well. The resulting code and final source code of the example is shown in Listing 5.3.

```
1 Lineitems.GROUP(Lineitems.map { |li| li.l_orderkey }).map { |_fm_rv_nr_1 |  
2   [ vlist(_fm_rv_nr_1)[0], vlist(_fm_rv_nr_1)[1].FILTER  
3   ( vlist(_fm_rv_nr_1)[1].map { |li| li.l_shipmode == "TRUCK" })] }
```

Listing 5.3: Source code of the example with replaced blocks, functions and inserted 'vlist'-calls

6 Conclusion

As this thesis has shown, it is possible to transform a Ruby program in a way, that it is ready for the translation into relational algebra, what finally leads to a database support of certain Ruby methods. Therefore certain goals had to be achieved. At first the source code of the program had to be analyzed syntactically. That was possible by using a build in library of the programming language Ruby, the Ripper library. Because this library provided a method, that was able to transform Ruby source code into a form, holding the syntactically information of the source code by using the source language itself, the entire work was able to be implemented in the Ruby programming language. The form of the output of this method, was an nested array, which described the syntax of the source code with the help of some Ruby programming constructs, like symbols. That representation of the syntactically structure of a program is called abstract syntax tree (AST). To work on this AST, it was necessary to put it in a form, that is easy to handle. A class system was installed, where every symbol contained in the nested array, had its appropriate class. This class needed to have attributes, reflecting the syntactical information of the symbol. The nested array was traversed and out of this representation a class tree was created, where every node of the class tree corresponded to one symbol of the origin representation.

When the class tree was finished, its nodes were analyzed for certain aspects. To facilitate the translation into relational algebra, the source code, that has to be translated, needed to have a preferably easy form. To enable that, two goals were formulated. If some methods were using blocks, the block should only have one block variable. Another goal was, that the source code should have no more methods with blocks or arguments, except the 'map'-method and some primitive helper methods. The 'map'-method was allowed to still have a block, while the primitive helper methods could have an argument. These goals were realized by normalizing the representation of the source code in the class tree. Nodes were changed, removed or created newly and inserted, after certain rules, until the class tree had a form, that satisfied the normalization goals.

The new form of the source code made a translation into relational algebra easier. But just modifying the source code was not enough to enable a translation. The inner and outer length, meaning the count of elements, of all nodes, standing for container objects, had to be known. Therefore the tree had to be traversed once more. By doing this recursively bottom up, the lengths were passed up through the tree towards the root, so that every node could get the information about the lengths of its children, that were needed to compute the length of the node itself.

Similar as it was the fact with the lengths, a type had to be added to every node. It was not so important to know the exact types of the nodes, it sufficed to just know how the nodes, representing lists were encoded. Three encodings were possible. A vertical

encoding standing for a normal list, a horizontal encoding, illustrating some kind of tuples and an atomic encoding, referring to a basic type, representing no list. With those types it should be possible to realize a translation and not falsifying the result by using a wrong encoding of a list. Once again the tree was analyzed until every node had got its appropriate type. While doing this, type checks for certain nodes have been done. For example if a method needed a vertically encoded list as input, it had to be ensured, that only if that was the case, the evaluation went on. Otherwise a faulty program would result. If one of the type checks of the underlying lists failed, there was the possibility to change the encoding of that list by inserting a function call, that changed the encoding. To realize this, a new node has to be inserted, representing the function call, talked of.

Finally the modified class tree could be presented as tree visualization, as well as a retransformation into Ruby source code could have been done, resulting in a new source code, adapted to the changes, made during the working steps.

The goals of this thesis have been reached and the next steps to get a database support for some Ruby methods can be made. In future one of these steps will be the actual translation from the modified facilitated Ruby code into the relational algebra. As a translation from relational algebra into database sequel code already exists (implemented by the Lehrstuhl of Datenbanksysteme, Wilhelm-Schickard-Institut für Informatik, Eberhard Karls Universität Tübingen), an evaluation of the database support could follow. Also an interesting thing to go on, is doing the work the other way round. Instead of analyzing Ruby code and translating it into relational algebra to compute the results of some Ruby methods using a database, it would be nice to have a mechanism to catch these results, pass them back to the Ruby program and store them on the heap. One last possible step is to use Ruby's meta programming abilities to implement self modifying code, meaning a program can alter and prepare its source code itself for database support, by searching and handling the methods, that need a modification treatment, like it was done in this thesis by another program.

List of Figures

2.1	Example of an AST created by Ripper	10
2.2	The working steps of this thesis	11
3.1	Visualization of the class tree	23
4.1	Class tree with replaced blocks	32
4.2	Class tree with replaced blocks and methods	45
5.1	Listing 5.1 encoded horizontally	48
5.2	Listing 5.2 encoded vertically	49
5.3	Class tree with evaluated lengths	65
5.4	Class tree with evaluated lengths and types	82
5.5	Class tree with evaluated lengths, types and inserted 'vlist'-calls	85

Bibliography

- [FM08] David Flanagan and Yukihiro Matsumoto. *The Ruby programming language*. O'Reilly, first edition, January 2008.
- [GKN06] Emden Gansner, Eleftherios Koutsofios, and Steffen North. *Drawing graphs with dot*. Graphviz - Graph Visualization Software, 2006.
- [Gru] Prof. Dr. Thorsten Grust. *tpch.rb*. Eberhard Karls Universität Tübingen, Wilhelm-Schickard-Institut für Informatik, Lehrstuhl für Datenbanksysteme.
- [Jon03] Joel Jones. *Abstract syntax tree implementation idioms*. Department of computer science, University of Alabama, 2003.
- [RDOa] RDOC. Ruby documentation system - ripper class.
- [RDOb] RDOC. Ruby documentation system. <http://rdoc.sourceforge.net/doc/index.html>.
- [sRpa] Rubyforge: Open source Ruby projects. <http://ruby-asp.rubyforge.org/ruby-graphviz/files/readme.html>.
- [sRpb] Rubyforge: Open source Ruby projects. <http://rubyforge.org/projects/parsetree/>.
- [TFH09] Dave Thomas, Chad Fowler, and Andy Hunt. *Programming Ruby 1.9*. The Pragmatic Programmers, April 2009.
- [Ven03] Bill Venners. The philosophy of Ruby - A conversation with Yukihiro Matsumoto, part 1. *Artima Developer Community*, September 2003.

Copyrights

Titlepage:

Iron Ore Mining, Bell Island, ca. 1900. Drawing by South Pub. Co., NY. From M. Harvey, Newfoundland in 1900 (St. John's, Nfld.: S.E. Garland, 1900)

Ruby logo by Ruby visual identity team. From Yukihiro Matsumoto, official Ruby logo.
<http://rubyidentity.org/>