Bachelor Informatica
Universiteit van Amsterdam

Compiling Ruby to Program Algebra

Gijs Kunze

August 24, 2006

Supervisor(s): Bob Diertens (UvA) & Inge Bethke (UvA)

Signed:

Abstract

The program algebra toolset contains several projections and embeddings between instruction sets. A thesis by Ruben Geerlings[1] introduces a new projection from a subset of Ruby. A projection from a more complex high-level language like Ruby requires a complex parser. This paper introduces an alternative method for the creation of projections using traditional compiler construction techniques. Additionaly, the projections from the Ruby subsets are implemented using these techniques.

Contents

List of Figures

CHAPTER 1

Introduction

The purpose of this paper is twofold. It adds support for a new projection to the PGA Toolset[1] and discusses an alternate technique for implementing these projections. The projection is based on the work done by Ruben Geerlings[1] whose thesis describes a projection of a subset of Ruby[6] to program algebra. This paper will show that for high-level languages the techniques I use here are better suited than the ones used for the relatively easy to parse languages used in the toolset so far.

1.1 Program algebra and Ruby

In the PGA hierarchy of primitive instruction sets build on one another, any valid program in one of the instruction sets can be projected down the hierarchy to the least complex instruction set PGLA[2]. Aside from primitive instruction sets which represent the logic and flow of a program, PGA also has basic instruction sets. Basic instruction sets represent operations on resources within a computer.

The basic instruction set MPP[2] and its family allow an abstracted and object-based view of system memory. This is exceedingly well suited for the projection of object oriented concepts.

Ruben Geerlings' thesis added five new languages (or instruction sets) to the hierarchy, an intermediate language between the Ruby subset and existing instruction sets and four subsets of Ruby, where each subset of Ruby improves the previous one with new features:

IPL The Intermediate Projection Language (a basic instruction set) was created to provide the features missing from the PGA toolset at the time. It provided features such as the *returning goto* and *variable goto*. IPL was built to project to PGLEcw[3] (see figure 1.1). IPL, and its replacement PGLEcrv will be discussed in section 2.1.1.

RC1 Ruby Core One is the first subset of Ruby. It is the most basic in that it supports only the basic features of object-oriented programming. Section 2.4 discusses the projection for RC1.

RC2 Ruby Core Two builds on RC1 by adding support for different kinds of methods. See section 2.5 for more details.

RC3 Ruby Core Three adds class variables, a feature similar to static properties in languages like Java and C++. Section 2.6 goes into more detail.

RC4 Ruby Core Four improves the projection by adding support for integers. This is elaborated on in section 2.7.

[1] Program Algebra[2] Toolset[5], see: http://www.science.uva.nl/research/prog/projects/pga/toolset
[2] Molecular Programming Primitives[3]
[3] A program algebra instruction set supporting, amongst other things if-then-else and while blocks

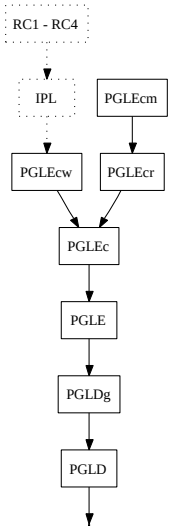


Figure 1.1: Partial PGA hierarchy with Ruby support, old situation

1.2 Projecting

Ruby is a high-level scripting language. These languages are known for the ease at which humans can work with them. This ease of use often comes at a price: easier programming languages are generally harder to parse for a computer. The projections currently in the PGA toolset are all using simple instructions easily parsable using regular expressions. The Ruby subsets used in this thesis are not as easily parsed. Several language constructs such as nested if-then-else blocks make Ruby a more complex language than regular expressions are capable of parsing.

Parsing computer languages, a task compilers are built for is one of the oldest and most mature subjects in computer science[7]. And since projecting a language such as Ruby Core One to program algebra *is* compiling, using compiler techniques to do this seems appropriate.

Many tools exist to help create compilers. These so-called 'compiler-compilers' assist in the creation of various parts of a compiler. Most tools focus on the front-end tasks of the compiler: lexical analysis, syntactic analysis and semantic analysis. Some tools perform only one task, for example the well-known lex only performs lexical analysis while its partner program yacc performs syntactic analysis. The tool I decided to use, ANTLR[4], performs all three of the tasks.

Most 'compiler compilers' take input from a file describing the grammar of the language and turn it into code in a specific language which when run will parse the described language. ANTLR allows the user to select a variety of languages for output. Unfortunately, a version capable of emitting Ruby parser code is not yet available. Therefore the implementation of the projection will be done in the only scripting language supported[5]: Python[6].

A feature which makes ANTLR eminently suitable for the projection of Ruby Core One through Four is that it supports grammar inheritance. Grammar inheritance allows us to easily add new features to an existing grammar. This is exactly what the Ruby Cores do to their preceding versions.

[4] http://antlr.org and see [8]
[5] The current stable version 2.7.x supports Java, C++ C# and Python
[6] http://www.python.org

CHAPTER 2

Improving the projection

The details of the projection of Ruby to PGA are explained in [1] but as it is essential to this program, and several changes have been made, I will list it here in its entirety. Because the original projection was made in 2003 and several new instruction sets have been developed since that time, I have updated the projection to make use of the more useful features of the new instruction sets.

2.1 The projection language

The Ruby subsets need a projection language to project to, this section discusses the old projection language IPL and its replacement PGLEcrv.

2.1.1 IPL

The IPL was made by Ruben Geerlings to serve as glue between the projections of Ruby and PGLEcw. The following instructions were provided by the IPL:

Variable Goto A jump instruction where the destination label is contained within a variable.

Returning Goto Similar to the `gosub` instruction in Basic, the *returning goto* jumps to a certain label and after it encounters a *return* instruction the program will jump back to the instruction following the original jump instruction.

Return A simple statement that jumps to the instruction following the last executed *returning goto* instruction.

Variable Returning Goto A combination of the *variable goto* and the *returning goto*, this allows one to jump to a variable label and return once it has been completed.

In addition to the instructions described here, IPL also added support for string labels as opposed to the conventional integer labels used in mode instruction sets. I have removed this support for string labels in the new projection as it was used inconsistently in the old one.

2.1.2 PGLEcrv

Since the publication of [1] PGLEcr has been introduced. PGLEcr already added support for the *returning goto* and *return* instructions. PGLEcr is not built upon PGLEcw which contains the *while* instruction needed in a few places in the old projection, but since a while loop is easily recreated using a jump instruction combined with an if-then-else block PGLEcr seems a better destination for our projection.

This still leaves out the *variable goto* and *variable returning goto* instructions. Since these are not available in the toolkit, I made a new instruction set PGLEcrv built upon PGLEcr introducing these two instructions. The details of the projection to PGLEcr are in Appendix A. Figure 2.1 shows the new situation.

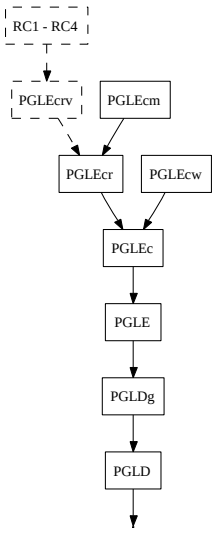


Figure 2.1: Partial PGA hierarchy with Ruby support

2.2 MPP and MSP

MPP[1] gives us the ability to create complex data structures. These data structures are usually represented as molecules floating in a liquid. In this section I will give a succinct introduction to the terminology of MPP and I will introduce several new basic instruction sets building upon it.

2.2.1 Terminology

MPP, the first of the molecules-in-fluid instruction sets introduces several terms used thoughout this chapter.

fluid The *fluid* contains all molecules created. This could be seen as the entire memory of a system.

atom A single object, it can contain any number of *fields*. In graphs *atoms* are represented by black dots.

field A named 'property' of an atom, which can point to any atom. In graphs *fields* are represented by named arrows.

molecule A group of *atoms* and their connections (through *fields*).

focus *Foci* are pointers to *atoms*. In other words, foci are the 'global' variables. A focus in graphs is represented by a named curly arrow.

Figure 2.2 is an example fluid with a single molecule, one focus x and two atoms with various fields.

2.2.2 MPPV

MPP with values adds support for booleans and integers to MPP, a field can now no longer just point to atoms but also to typed values. The original projection used an extension to MPPV called MPPVs which added a third type: strings. However MPPVs has never been implemented and we now have access to even better alternatives.

[1] Molecular Programming Primitives

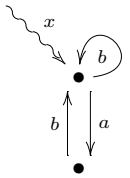


Figure 2.2: A simple fluid

2.2.3 HMPPV

High-level MPPV[4] is a major addition to the MPP family. First of all it adds the string type, and more importantly it adds – as its name suggests – higher level instructions for the manipulation of the fluid. This can shorten the projection somewhat as programming with MPP occasionally requires several instructions to do a simple task. Besides these additions HMPPV adds support for garbage collection[2].

2.2.4 MSP

Molecular Scripting Primitives[4] adds support for the manipulation of strings and integers introduced in HMPPV and MPPV such as addition and subtraction for integers, concatenation for strings and conversions between the two. Since Ruby Core Four introduces integers, the integer operations are quite useful. Therefore the new projection uses MSP as the destination basic instruction set.

2.3 Large changes

Since some changes have had an effect on a large part of the original projection. This section contains several of the drastic changes that have been made to the projection.

2.3.1 PGLEcr and the `stackframe`

PGLEcr introduces the *returning goto*. This instruction is used extensively in the projection. The projection of PGLEcr to PGLEc is implemented using a stack with MPP, specifically the molecule pointed to by focus `stackframe`. This implementation is exactly the same as the one described for IPL. Unfortunately in the original projection the focus `stackframe` is used for other purposes as well. It contains a field pointing to the current object (*self*) and the variables in the current scope. The problem this introduces is that because of PGLEcr the `stackframe` focus is reserved and cannot be used in languages built on top of PGLEcr. A `stackframe` focus used in PGLEcrv would behave completely different once it had been projected to a language above PGLEcr (like PGLEc) since the projection makes use of the focus.

A solution to the problem is to create an alternative focus to a stack-like molecule behaving similarly to the focus `stackframe`. I have named this focus `locals`. The end result is that the `lv`[3] and `self` fields both moved to the `locals` stack and that the stack must be manipulated manually when scope changes.

2.3.2 Movement of several foci to fields

MPPV defines the boolean foci `true` and `false`. The original projections uses these foci as well which creates a conflict. To solve this conflict I have moved all constants as fields to the focus `constants`. Similarly I moved all classes out of the local 'scope' to the focus `classes`.

[2]The removal of atoms no longer referenced.
[3]local variables

2.3.3 Methods as arguments

In the original projection a method call used the foci `x` and `arg`*1*...`arg`*N* to specify its object and arguments respectively. This made it impossible for method calls to appear as arguments in another method call as the first would overwrite the arguments and `x` of the other before it could be called. I solved this by using a similar stack as the newly introduced `locals` stack(Section 2.3.1) called `callstack`. This stack contains the fields `x` and `arg`*1*...`arg`*N*. The small but significant difference between `locals` and `stackframe` is the moment during the method call at which the stack is altered, this makes it possible for arguments to a method to still access variables local to the current method.

2.4 Ruby Core One

This section describes the updated version of the projection of Ruby Core One to PGLEcrv. This projection is the biggest of the fours since it introduces all of the object oriented concepts in Ruby and the Ruby syntax itself.

2.4.1 Notation

The notation for this projection and the others is the same as it is in [1]. Here is a short list of the projection functions:

$\psi(X)$ Projects the Ruby instructions X to PGLEcrv.

$\psi_{x_1,...,x_n}(X)$ Projects the Ruby instructions X to PGLEcrv, with the instructions restricted to a certain context.

$\varphi_x(a_1, ..., a_n)$ A PGLEcrv macro named x. This is only used for clarity, whenever a macro is encountered in the projection, the contents of the macro can be substituted in place with the arguments of the macro substituted as well.

A *context* restricts the allowed instructions in the program. This is used to ensure that language constructs like `return` statements are only placed within methods.

2.4.2 Programs

We start with the complete projection function `rc2pglecrv` projecting a Ruby program to a PGLEcrv program. There are no big changes to this function. It has been renamed (the original name was `rc2ipl`) and the $\varphi_{init-stackframe}$ macro is replaced by $\varphi_{init-stacks}$.

`rc2pglecrv`$(u_1; ...; u_k) =$
$\varphi_{init-classes}$;
$\varphi_{init-methods}$;
$\varphi_{init-stacks}$;
$\psi_{main}(u_1); ...; \psi_{main}(u_k)$

Macros

The $\varphi_{init-classes}$ macro initializes the basic class structure. It also initializes the constants `main`, `true`, `false` and `nil`. Figure 2.3 shows a part[4] of the fluid after the macro has run. There are no real changes to the projection except for the movement of the molecules to fields under a new focus (as per section 2.3.2).

$\varphi_{init-classes} =$
`classes = new;`

[4]Several atoms were removed for clarity

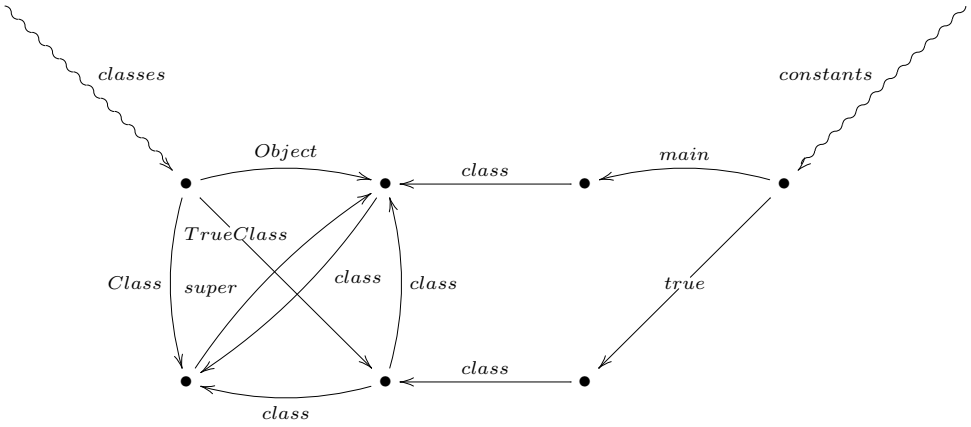


Figure 2.3: Part of the fluid after the $\varphi_{init-classes}$ macro has run.

```
classes.+Object = new;
classes.+Class = new;
classes.Class.+super = classes.Object;
classes.Class.+class = classes.Class;
classes.Object.+class = classes.Class;
classes.+TrueClass = new;
classes.TrueClass.+super = classes.Object;
classes.TrueClass.+class = classes.Class;
classes.+FalseClass = new;
classes.FalseClass.+super = classes.Object;
classes.FalseClass.+class = classes.Class;
classes.+NilClass = new;
classes.NilClass.+super = classes.Object;
classes.NilClass.+class = classes.Class;
constants = new;
constants.+true = new;
constants.true.+class = classes.TrueClass;
constants.+false = new;
constants.false.+class = classes.FalseClass;
constants.+nil = new;
constants.nil.+class = classes.NilClass;
constants.+main = new;
constants.main.+class = classes.Object
```

The $\varphi_{init-begin}$ and $\varphi_{init-end}$ macros are used for the methods of the classes initialized in the $\varphi_{init-classes}$ macro. They are placed around the *body* of the method, as you will see on page 12.

This was the only place in which string labels were used in the original projection, and since the $\varphi_{im-begin}$ macro is only used for a few methods (only those in the $\varphi_{init-methods}$ macro) they were discarded in favor of numerical labels. Here i and j are two unused labels, different for every macro invocation.

$\varphi_{im-begin}(C, m) =$

```
-classes.C/im {; classes.C.+im = new; };
classes.C.im.+m = new;
classes.C.im.m.+label:int = i;
```

```
   ##Lj; Li
φim−end =
   R; Lj
```

Since two new stacks have been added to the projection, we need a way to manipulate them, these four macros do just that.

```
φlocals−up =
   locals.+next = new; locals.next.+prev = locals; locals = locals.next
φlocals−down =
   locals = locals.prev; locals.-next
φcallstack−up =
   callstack.+next = new; callstack.next.+prev = callstack
   callstack = callstack.next;
φcallstack−down =
   callstack = callstack.prev; callstack.-next
```

The $\varphi_{init-methods}$ macro initializes several methods for the base classes of the system. The `Class.new` method is interesting in that it reuses the existing `callstack` instead of adding a new one to the stack. This allows the method to call the `initialize` method on the instantiated object with the same arguments it was called with. The biggest change over the old projection is probably the addition of the `!=` method. The `!=` method calls the `==` method on the object and inverses the result, this causes `!=` to automatically work correctly when the `==` method is overridden.

```
φinit−methods =
   φim−begin(Object, initialize);
   φim−end;

   φim−begin(Object, class);
      result = self.class;
   φim−end;

   φim−begin(Object, ==);
      +self == callstack.arg1{; result = constants.true; }{;
      result = constants.false; };
   φim−end;

   φim−begin(Object, !=);
      φlocals−up; locals.lv.+arg1 = callstack.arg1;
      ψ(self == arg1);
      +result == constants.true {; result = constants.false; }{;
      result = constants.true; };
      φlocals−down;
   φim−end;

   φim−begin(Class, initialize);
      self.+super = callstack.arg1;
   φim−end;

   φim−begin(Class, superclass);
      +self/super{; result = self.super; }{; result = constants.nil; };
   φim−end;

   φim−begin(Class, new);
      callstack.x = new; callstack.x.+class = self;
```

$\varphi_{search-instance-method}$(initialize);
self = callstack.x; label = method.label; R##L[label];
result = callstack.x;
φ_{im-end}

The $\varphi_{init-stacks}$ macro replaces the $\varphi_{init-stackframe}$ macro of the old projection since the `stackframe` is now used implicitly[5], while there are two new stacks which need initializing.

$\varphi_{init-stacks}$ =
locals = new; callstack = new;
locals.+self = constants.main; locals.+lv = new

2.4.3 Classes

The projection for a class definition is quite simple. First it checks if the class we are trying to declare already exists[6]. If the class does not exist, we create a new Class object (classes in Ruby are instances of the `Class` class) and assign it to the appropriate field. When that is complete, open a new scope and execute the statements inside the class declaration. In this projection C is the classname, P is the name of the parent-class and u_1 through u_k are the Ruby instructions inside the class definition.

The projection for a class definition has not changed much compared to the original projection. The only real change is the check whether a parent class exists. This check is not necessary for the execution of well-written code, but should one try to inherit from a non-existant class it could be possible the code would attempt to inherit from a local variable. The compiler (see chapter 3) will automatically set the parent to `Object` if no parent is explicitly defined, therefore in the new projection P is no longer optional.

ψ_{main}(class $C < P; u_1; ...u_k;$ end) =
-classes/C {; -classes/P {; !; }; ψ(Class.new(P)); };
$\varphi_{locals-up}$;
locals.+self = classes.C; locals.+lv = new;
$\psi_{class}(u_1)$;...;$\psi_{class}(u_k)$;
$\varphi_{locals-down}$

2.4.4 Methods

Method Definitions

Method definitions can occur inside the global scope and in class definitions. The first part checks which of the two cases is true at the moment. The method declaration then goes on to add a new method entry to the respective object with an integer value to the label the method body starts at. This is followed by a jump past the method body (a declaration obviously shouldn't execute the method body). The method body itself increases the scope and assigns all the arguments to the associated local variables. Before the method returns back to the caller the scope is lowered again. In this projection i and j are two unused labels.

$\psi_{class,main}$ (def $m(p_1, ..., p_n); u_1; ...; u_k;$ end) =
+locals.self == constants.main {; cl = classes.Object; }{;
cl = locals.self; };
-cl/im {; cl.+im = new; };
cl.im.+m = new;
cl.im.m.+label:int = i;

[5]The stackframe focus is introduced in the projection from PGLEcr to PGLEc
[6]Ruby allows one to reopen a class definition to add additional functionality.

```
##Lj ; Li ;
φ_locals−up ;
locals.+lv = new; locals.+self = self;
locals.lv.p_1 = callstack.arg1;...;locals.lv.p_n = callstack.argn;
ψ_method(u_1);...;ψ_method(u_k);
φ_locals−down ;
R; Lj
```

Return Statement

The return statement, which makes the current method return the value in *expr* can also be invoked without an expression in which case the function should return the value `nil`. Similarly to the parent-class in the class declaration, the semantic analysis phase will automatically insert the implicit `nil` value.

ψ_{method}(`return` *expr*) =

```
ψ(expr);
φ_locals−down ;
R
```

Method Call Macros

The method call projection makes use of a pair of macros to find the correct method to call. The $\varphi_{search-instance-method}(m)$ macro searches the class of the current object and its ancestors for method m. To do that, it invokes the $\varphi_{search-supers}$ macro, which will be explained next.

$\varphi_{search-instance-method}(m)$ =

```
φ_search−supers(callstack.x.class, im, m);
+found == false {; !; }{; method = res; }
```

The $\varphi_{search-supers}$ macro searches an atom x and all its ancestors[7] for an atom with a field s which points to an atom with a field i. This is a general method for finding *things* in the class tree. The method call projection uses it to search the class tree for the first ancestor which supports the method it wants to call. Since we can no longer use the while-loop construction of PGLEcw, there was a need to use an if-then-else statement in combination with a jump. In the projection a is an unused label, different for every macro invocation.

$\varphi_{search-supers}(x, s, i)$ =

```
loop = constants.true; found = constants.false;
sp = x; La;
+loop == constants.true {;
   +sp/s{;
      br = sp.s ;
      +br/i{;
         loop = constants.false; found = constants.true; res = br.i ;
      };
   };
   +sp/super{;
      sp = sp.super;
   }{;
      loop = constants.false;
   };
   ##La ;
}
```

[7] atoms pointed to by the *super* field

Method Calls

When a method call is encountered, first the object on which it is executed is evaluated. Again, the compiler implicitly adds `self` as the object if none is given. As stated in section 2.3.3, the `callstack` focus is now used to store the resulting object as well as the evaluated arguments. One should also note that since the `callstack` stack has to be increased to store this data at the beginning of the method call it is not possible to increase the `locals` stack as the evaluation of the arguments quite possibly need to use local variables which would otherwise no longer be available. Once the `callstack` structure has been filled, the projection searches for the correct mathod to call using the $\varphi_{search-instance-method}(m)$ macro. All that is left is to actually jump to the found label using the *returning goto* instruction.

$\psi(exp_0.m(exp_1, ..., exp_n)) =$
$\psi(exp_0)$;
$\varphi_{callstack-up}$;
`callstack.+x = result;`
$\psi(exp_1)$; `callstack.+arg1 = result;...;`$\psi(exp_n)$; `callstack.+argn = result;`
$\varphi_{search-instance-method}(m)$;
`self = callstack.x; result = constants.nil; label = method.label;`
`R##L[label];`
$\varphi_{callstack-down}$

2.4.5 Expressions

Most of the projections in this section are relatively simple and should speak for themselves.

Local Variables

The Ruby interpreter uses heuristics to determine whether an identifier is a class identifier or a variable identifier depending on capitalization and the current scope. I have simplified this in the new projection: if an identifier is encountered, and if there is a class with the same name, it is used otherwise the local scope is searched for a variable.

$\psi(x) =$
```
+classes/x {;
   result = classes.x;
}{;
   -locals.lv/x {;
      !;
   }{;
      result = locals.lv.x;
   };
}
```

Instance Variables

Instance variables are even simpler than local variables. Since classes can't be prefixed by an @ character there is no need to search for possible matching classes. A small change in behaviour between local and instance variables is how non-existant identifiers are handled. Undefined instance variables have the value `nil` while undefines local variables cause an error.

$\psi(@x) =$
```
-locals.self/iv {; result = constants.nil; }{;
   -locals.self.iv/x {; result = constants.nil; }{;
      result = locals.self.iv.x;
```

```
    };
}
```

Constants

The Ruby subsets have three real constants, `true`, `false` and `nil`. Because `self` changes depending on where it is evaluated, it technically isn't a constant. However since the programmer can not change `self` it does have similar properties. In the original projection, declared classes were also constants, in essence every class had its own focus in the fluid. This could lead to dangerous situations: a class called `result` could destroy the validity of an entire program.

$\psi(self)$ =`result = locals.self`
$\psi(true)$ =`result = constants.true`
$\psi(false)$ =`result = constants.false`
$\psi(nil)$ =`result = constants.nil`

Assignments

In expressions, assignments are treated differently than other expressions since assigning to a variable can create a new identifier, while using a previously undefined identifier in any other situation (besides the `defined?` test discussed next) causes an error.

$\psi(x = expr)$ =
```
    ψ(expr);
    locals.lv.+x;
    locals.lv.x = result
```

The assignment to an instance variable is not really any different than the assignment to normal variable, except for the place the variable is added to of course.

$\psi(@x = expr)$ =
```
    ψ(expr);
    -locals.self/iv {; locals.self.+iv = new; };
    locals.self.iv.+x;
    locals.self.iv.x = result
```

Tests

The `defined?` tests have changed somewhat: the original projection returned boolean values depending on the existence of the variable. However, in Ruby `defined?` returns `nil` when the variable doesn't exist and a *string* containing its description if it does exist. Unfortunately, our Ruby subset doesn't support strings. To maintain closer compatibility, the new projection returns `true` and `nil` for existance and non-existance respectively.

$\psi(defined?\ x)$ =
```
    -locals.lv/x {; result = constants.nil; }{; result = constants.true };
```

$\psi(defined?\ @x)$ =
```
    -locals.self/iv {; result = constants.nil; }{;
        -locals.self.iv/x {; result = constants.nil; }{; result = constants.true };
    }
```

2.4.6 Conditional Statements

Macro

The simple $\varphi_{nil \to false}$ macro changes the latest result to `false` if it is `nil` otherwise it does nothing. Since `false` and `nil` are the only values which should evaluate to false, this macro makes it easy to test for `false`ness in the conditional statements.

$\varphi_{nil \to false} =$
```
+result == constants.nil{ result = constants.false; }
```

If-Then-Else Statement

The `if-then-else` is a well known construct available in practically every imperative language. The compiler adds an implicit `else` block for any if-then statement it encounters. Therefore a second projection is necessary no longer.

$\psi_X($`if` $expr; u_1; ...; u_k;$ `else`$; u_{k+1}; ...; u_l;$ `end`$;) =$
```
ψ(expr);φ_{nil→false};
-result == constants.false {;
    ψ_X(u_1);...;ψ_X(u_k);
};{;
    ψ_X(u_{k+1});...;ψ_X(u_l);
}
```

While Statement

In the original projection, the while loop utilized the `while` instruction of PGLEcw which is no longer available to us. Fortunately it turned out the projection is actually smaller when it uses an `if` instruction combined with a jump as it requires only a single place in which to evaluate *expr*. In this projection j is an unused label.

$\psi_X($`while` $expr; u_1; ...; u_k;$ `end`$;) =$
```
Lj;
ψ(expr);φ_{nil→false};
-result == constants.false {;
    ψ_X(u_1);...;ψ_X(u_k);
    ##Lj;
}
```

2.5 Ruby Core Two

Ruby Core Two extends the projection with support for *singleton* methods and *class* methods.

Singleton Methods A method which exist on one object, it is not associated with a class like normal methods.

Class method Is a method defined on a class itself. It can be compared to the static methods in Java and C++.

Basically the two new method types are one and the same. Both are a method on an object which results from an expression. In the case of class methods that object is the instance of the Class instance for class the method is defined on. In the case of singleton methods, the object is any other sort of object.

2.5.1 Methods

Singleton/Class Method Definitions

The declaration of a class or singleton method is done quite similarly to ordinary (instance) method declarations, except the *expr* is evaluated first and the method is declared on that object instead of `self` or `main`. In this projection i and j are unused labels.

$\psi_{class,main}$(`def` $expr.m(p_1, ..., p_n); u_1; ...; u_k;$ `end`) =
    $\psi(expr)$;
    `-result/sm { result.+sm = new; };`
    `result.sm.+`m `= new;`
    `result.sm.`m`.+label:int =` j`;`
    `##L`i`; L`j`;`
    $\varphi_{locals-up}$;
    `locals.+lv = new; locals.+self = self;`
    `locals.lv.`p_1 `= callstack.arg1;...;locals.lv.`p_n `= callstack.argn;`
    $\psi_{method}(u_1)$;...;$\psi_{method}(u_k)$;
    $\varphi_{locals-down}$;
    `R; L`i

Method Call Macros

When a method call is made in RC2, we now have to search for all three types of methods. The $\varphi_{search-method}(m)$ macro searches for the new method types using the $\varphi_{search-supers}$ macro defined in section 2.4.4. If no method is found, it searches for ordinary instance method using the $\varphi_{search-instance-method}(m)$ macro.

$\varphi_{search-method}(m)$ =
    $\varphi_{search-supers}$(`callstack.x, sm,` m);
    `+found == constants.false {;`
        $\varphi_{search-instance-method}(m)$;
    `}{;`
        `method = res;`
    `}`

Method Calls

Since the method for a method call is found using a single macro, the only thing that has to change in the new method call projection for RC2 is the name of the macro:

$\psi(exp_0.m(exp_1, ..., exp_n))$ =
    $\psi(exp_0)$;
    $\varphi_{callstack-up}$;
    `callstack.+x = result;`
    $\psi(exp_1)$; `callstack.+arg1 = result;...;`$\psi(exp_n)$; `callstack.+argn = result;`
    $\varphi_{search-method}(m)$;
    `self = callstack.x; result = constants.nil; label = method.label;`
    `R##L[label];`
    $\varphi_{callstack-down}$

2.6 Ruby Core Three

Ruby Core Three has a single additional feature over Ruby Core Two, namely class variables. Class variables can be compared to static variables in classes from languages such as C++ and Java. Since classes themselves are instances too, using instance variables[8] in class methods gives us functionality close to static variable. However these 'fake' class variables are only available in class methods. Class variables are accessible in any method of the class in question.

2.6.1 Expressions

Macro

The $\varphi_{find-class}$ macro gives us the class object for the current `self`. Should the current object already be an instance of the `Class` class (we're in a class declaration or class method) the macro just gives `self`. This way, no matter the location in the code, we always get the current object class variables should exist on.

$\varphi_{find-class} =$
```
cl = locals.self.class;
+cl == classes.Class {; cl = locals.self; }
```

Variables

When a class variable identifier is encountered, the $\varphi_{search-supers}$ macro is utilized to find the class variable in any of the ancestor of the current class. Should the variable not be found, an error occurs, otherwise the first variable that is found is returned.

$\psi(@@x) =$
```
φ_find-class;
φ_search-supers(cl, cv, x);
+found == constants.false {; !; }{; result = res; }
```

Assignments

The assignment to class variables is an interesting case: as with ordinary class variables (not assignments) the ancestors are searched for the class variable, if it is not found, it is added to the current class. However if the class variable is found, the class variable is updated *on the class it was found on.*

$\psi(@@x = expr) =$
```
ψ(expr);
φ_find-class;
φ_search-supers(cl, cv, x);
+found == constants.false {;
   cl.+cv = new; cl.cv.+x = result;
}{;
   br.+x; br.x = result;
}
```

Tests

The `defined?` instruction for class variables, behaves similarly to the tests in section 2.4.5. The only difference is that several classes are searched, and only one of the ancestors needs to have

[8] A single @ character, as opposed to two for class variables

it defined.

$\psi(defined?\ x) =$
```
φ_find−class;
φ_search−supers(cl, cv, x);
+found == constants.false {;
result = constants.nil;
}{;
result = constants.true;
}
```

2.7 Ruby Core Four

Ruby Core Four adds support for integers. While integers could be represented using only classes, it would be inefficient and very usable. Support for integer literals and operations upon them. The only supported operations are addition and the equality test[9].

2.7.1 Programs

An RC4 program requires additional initialization of the classes and methods which represent integers.

$\texttt{rc2pglecrv}(u_1; ...; u_k) =$
```
φ_init−classes;
φ_init−methods;
φ_init−stacks;
φ_init−integer;
ψ_main(u_1); ...; ψ_main(u_k)
```

Macros

The $\varphi_{init-integer}$ macro initializes the `Integer` and `Fixnum` classes. The `Integer` class is the parent class for all integer types. One of these we (partially[10]) implement, namely the `Fixnum` integer type. This type supports fixed size integers (no arbitrarily large integers). Figure 2.4 shows what an integer value looks like.

$\varphi_{init-integer} =$
```
classes.+Integer = new;
classes.Integer.+class = classes.Class;
classes.Integer.+super = classes.Object;
classes.+Fixnum = new;
classes.Fixnum.+class = classes.Class;
classes.Fixnum.+super = classes.Integer;

φ_im−begin(Fixnum, ==);
+self.value == callstack.arg1.value {;
   result = constants.true;
}{;
   result = constants.false;
};
φ_im−end;

φ_im−begin(Fixnum, +);
```

[9] Support for the equality test implies support for the inequality test as well
[10] Many operations are absent as well as negative number support

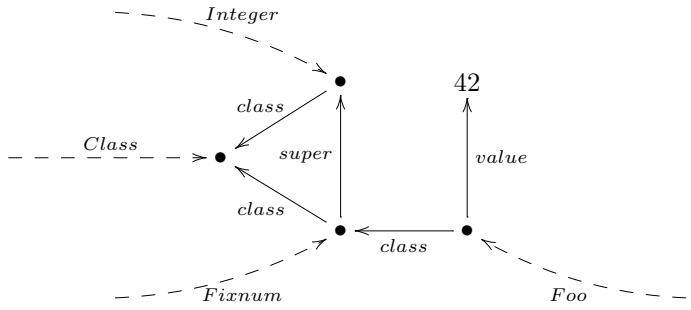


Figure 2.4: Fluid containing an integer object `Foo`

```
result = new; result.+class = classes.Fixnum; result.+value:int = 0;
incr result.value self.value;
incr result.value callstack.arg1.value;
```
φ_{im-end}

2.7.2 Expressions

Literals

Integers are immutable[11], therefore when an integer literal *int* is encountered, a new `Fixnum` instance is created.

$\psi(int) =$
```
result = new; result.+class = classes.Fixnum; result.+value:int = int
```

[11] Every operation generates a new object, values can not be changed.

CHAPTER 3

Compiling Ruby

As stated in the introduction, I use ANTLR for the creation of the Ruby parser. ANTLR is a tool written in Java which generates LL(k)[1] parsers. As opposed to other well known tools such as Yacc which generates LALR[2] parsers, the code generated by ANTLR is easily readable as it looks similar to a recursive descent parser written by hand.

ANTLR is a full featured tool providing lexical, syntactic and semantic analysis of code. Since this is all done in a single package, the different parts interact with each other seamlessly. The tool uses Extended Backus-Naur Form (ENBF) for all three of the tasks. This leads to exceedingly powerful and readable parsers. As stated in the introduction, ANTLR allows grammar inheritance which enables us to inherit from an existing grammar and extend it in some way which is exactly what we need for the four subsets. This system of inheritance also greatly increases the extendability of what I have made. It is quite easy to make a Ruby Core Five with support for some new language feature by simpy inheriting from the RC4 grammar.

This project was my first experience with ANTLR, and as such during the creation of the parser I made some mistakes. Due to the power ANTLR gives the developer and the fact that it allows one to approach a task from many directions, this was to be expected.

3.1 Lexical analysis

Lexical analysis or *lexing* is the process of transforming a series of characters into a series of tokens. Most tools for creating a lexical analyser or *lexer* allows one to match the input using a series of regular expressions. ANTLR allows the user to use the power of not only regular expressions but also EBNF, syntactic predicates[3] and semantic predicates[4]. The end result of this is that an ANTLR-built lexer is powerful enough to parse context-sensitive languages, albeit sometimes with a little difficulty.

To illustrate the workings of the lexer, I will walk through a part of the lexing code for the Ruby subset lexer. Specifically the part shown in figure 3.1.

Line 1 declares a token rule, in this case IDENTIFER, which matches identifiers such as variables, class names, method names and literals[5].

Line 2-4 specify an option specific to this token-rule, 'testLiterals'. This options enables the checking for literals. In the parser (see section 3.2) literals are defined simply by using them as a string. The lexer checks the associated text to this token, and if it matches one of the literals, it will automatically change the token type to the literal's token type.

On line 5 the rule starts (denoted by the colon) and says the first part of the match for this token must either match an underscore ('_'), or the LETTER token (or rule). This shows the

[1] Left-to-right Leftmost derivation with k tokens lookahead.
[2] Look Ahead Left-to-right Rightmost derivation
[3] A method which allows temporary infinite lookahead using backtracking
[4] Arbitrary piece of code which determines is a path may be chosen
[5] Literals in this case are words used by the language such as `if` and `true`

```
01: IDENTIFIER
02: options {
03:     testLiterals = true;
04: }
05:     : (LETTER|'_') (LETTER|DIGIT|'_')*
06:     (
07:     | ("?"|"!") { $setType(METHOD_IDENTIFIER); }
08:     )
09:     ;
10:
11: protected
12: LETTER   :    ('a'..'z'|'A'..'Z');
13:
14: protected
15: DIGIT    :    ('0'..'9');
```

Figure 3.1: Part of the Ruby Subset Lexer

power of EBNF in the lexer, as rules can also reference to themselves making recursive rules. The LETTER token is defined later, but suffice to say it matches a single letter of the alphabet. The last part of line defines a closure of a LETTER, a DIGIT or an underscore. This is a definition for an identifier as seen in many languages, an underscore or letter followed by zero or more letters, digits and underscores. But the definition does not stop here...

The parenthesis on line 6 start a subrule, and the first option is empty. Basically this means that unless the second option (on line 7) is matched[6] this subrule can match nothing.

The pipe on line 7 starts the second alternative for the subrule. In our subset (and in Ruby itself as well) method identifiers may end in either an exclamation point or a question mark. Therefore on this line it tries to match either character. Following the match is a so-called semantic action. If either character is matched the code between the curly braces is executed. Since ANTLR can output multiple languages, several macros have been added for common tasks which are automatically translated to the chosen output language. This particular macro changes the token type to METHOD_IDENTIFIER instead of IDENTIFIER (the name of this rule) since when an identifier ends in either character it can not be used for anything other than methods.

Line 9 ends the current rule (IDENTIFIER).

Line 11 tells us the following rule is a helper rule, it can be called from other rules but it will not generate tokens by itself.

Line 12 defines the LETTER helper rule. It matches a single alphabetic character. The '..' operator is used for ranges. Therefore a LETTER is between 'a' through 'z' inclusive or between 'A' through 'Z' inclusive.

Line 14-15 contains another helper rule DIGIT which matches a single numerical character.

All in all, the lexical analysis of the Ruby subset was quite easy to implement. The complete lexer definition which matches all the tokens needed for the four Ruby subsets is around 50 lines. In the end I decided to use the same lexer for all four of the parsers since while the syntax changes between every Ruby Core, the lexicon hardly changes.

3.2 Syntactic analysis

Syntactic analysis is often referred to as *parsing* even though the lexer and the semantic analysis also do their part in parsing code. The syntactic analyser or *parser* finds a hierarchical structure in the token stream provided by the lexer. The parser turns this hierarchical structure into an abstract syntax tree. The details of the creation of this tree are discussed in section 3.2.3. Like

[6] By default lexing is greedy

the lexer, the parser consists of a set of rules. A parser also has a starting rule[7] which is obviously where parsing begins.

An ANTLR parser supports the same features as it did in the lexer (EBNF, syntactic and semantic predicates). To illustrate the workings of parsing in ANTLR I will walk through a small example shown in figure 3.2.

```
01: program
02:   : (statement)* EOF
03:   ;
04:
05: statement
06:   : ("if" expression "then" (statement)* "else")=> ifthenelse
07:   | ifthen
08:   | expression
09:   ;
10:
11: expression
12:   : NUMBER
13:   | {foo == True}? methodcall
14:   ;
15:
16: ifthen
17:   : "if" expression "then"
18:     { foo = True } (statement)*
19:     { foo = False } "end"
20:   ;
21:
22: ifthenelse
23:   : "if" expression "then"
24:     (statement)* "else" (statement)*
25:     "end"
26:   ;
27:
28: methodcall
29:   : IDENTIFIER LPAREN RPAREN
30:   ;
```

Figure 3.2: Simple parser example

Line 1-3 defines our starting rule. A starting rule is not determined in the grammar, but by the the way you invoke the parser[8]. A `program` consists of zero or more `statement`s followed by an EOF. EOF is a predefined token found at the end of a token stream.

Line 5 starts the definition of a new rule `statement`.

Line 6 begins with the first alternative for a `statement`, a `()=>` construct. This is a syntactic predicate. The parser will try to match what is between the parentheses and should it match, the `ifthenelse` rule is tried, otherwise it is skipped. The code between the parantheses tries to match an if-then-else-end statement up to the `else` literal to determine what kind of if statement this is. Since ANTLR has finite lookahead (usually no more than 4 tokens) and there are a potentially unlimited tokens between the `if` and the `else` literals, a syntactic predicate is needed.

The second alternative for a `statement` on line 7 attempts to match the `ifthen` rule. While the third alternative rule on line 8 attempts to match an `expression`.

Line 12 tells us the `expression` rule can match a NUMBER. Presumably a NUMBER is a token containing digits. This NUMBER should correspond to a NUMBER rule in the lexer.

[7] Usually just one, unless you are parsing different things with the same parser.

[8] every rule becomes a method on your parser object, and you call the starting rule

Statement	Allowed when?
class definition	Main context
method definition	Main and Class context
return	Method context anywhere in stack
if	Anywhere
while	Anywhere
expression	Anywhere
empty statement	Anywhere

Table 3.1: Statements and when they are allowed

Line 13 contains a semantic perdicate. The code `foo == True`[9] is executed, and should it yield `True` the `methodcall` rule is tried, otherwise it is not.

On line 17 (excluding the syntactic predicate on line 6) we encounter our first literal. Literals are automatically added to the list of token types. This allows lexer rules with the 'testLiterals' option set to true to look for these literals.

On line 18 and 19 the statement closure is surrounded by two semantic actions setting the variable `foo` to `True` and `False` respectively. This means that for any statement matched inside an if-then-end block the `foo` variable is set to `True` (unless something else changes it). Thus when inside an if-then-end block, we are allowed to perform `methodcall`s.

Lines 22-26 define the if-then-else-end statement. It is essentially the same as the if-then-end statement with the addition of an else-block and without the semantic actions setting the `foo` variable.

Lastly lines 28-30 contain the `methodcall` rule. We assume IDENTIFIER is a lexer rule representing a series of alphanumeric characters while LPAREN and RPAREN stand for '(' and ')' respectively.

3.2.1 The Ruby parser

During the creation of the parser for Ruby Core One specifically in the rule matching statements[10], I used semantic predicates the check whether a statement was allowed in the current context. While this seemed to be a neat solution, I did have to maintain a stack maintaining the current context. This required quite a few semantic actions, making the parser somewhat less readable. The stack starts with a single value 'MAIN' representing the main context, and for any statement which can contain other statements the context is pushed onto the stack. Even though it is not technically necessary for while and if statements at this time it can be should a statement like break[11] be implemented. Table 3.1 shows when a statement is allowed.

Probably the hardest part of making a well working parser is the parsing of expressions. Expressions have many different aspects: the precedence of operators, associativity of operators, postfix operators like method calls and the various constants. Parsing expressions with LALR parser tools like yacc or bison is quite easy, ANTLR does not have this ease of use in this case. Without any doubt the expression rules were the most complex to setup correctly even though since the number of operators in the four Ruby subsets is rather small relatively few rules were needed. Table 3.2 shows all expressions and their precedence.

To actually implement a parser for expressions using ANTLR a parser rule is required for every group of operators with the same precedence. The expression rule starts with the rule corresponding to the lowest precedence, in this case the assignment operator. The format for binary operators[12] usually takes one of two forms as displayed in figure 3.3. These rules create very different trees (Section 3.2.3).

[9] In Python `True` and `False` start with an uppercase letter

[10] There are 7 different statements: class definition, method definition, return, if, while, expression and the empty statement.

[11] terminates execution of the current loop and continues at the first statement following it.

[12] operators with two operands like +, × and ==

Type	Examples
defined test	`defined? @foo`
constants	`true self`
parenthesized expressions	`(mymethod() == true)`
variables	`myvar`
dot operator	`foo.bar`
method calls	`foo() foo.bar(baz, ban)`
addition (RC4 only)	`foo + 42`
tests	`foo == bar foo != false`
assignment	`@bar = foo baz = nil`

Table 3.2: Operator precedence, from high to low

```
operator_rule:  next_operator_rule (OPERATOR next_operator_rule)*
```

(a) Left associativity

```
operator_rule:  next_operator_rule (OPERATOR operator_rule)?
```

(b) Right associativity

Figure 3.3: Syntax for operator associativity

3.2.2 Grammar inheritance

For the Ruby Core parsers I used grammar inheritance to implement the four languages. The RC1 grammar is the largest grammar defining all the rules for the program. Because the subsequent subsets only introduce a couple of new language features, through grammar inheritance the grammars for RC2 through RC4 contain only the altered and new rules. In the end this had the effect that those grammars contained only a few rules.

3.2.3 Creating an AST

An abstract syntax tree (AST) contains structured information about the source program and it is the data structure used for semantic analysis. ANTLR provides the user with a simple but powerful syntax for the creation of an abstract syntax tree. Figure 3.4 shows an example AST for a simple expression.

By default ANTLR creates a tree where the first item in a rule becomes the parent node of all the following items. Special syntax is required for each rule to add meaningful structure.

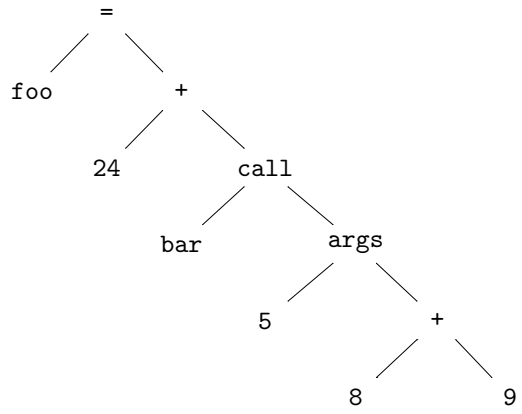
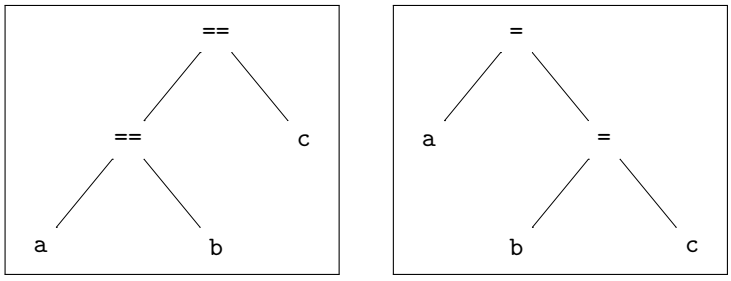


Figure 3.4: AST for `foo = 24 + bar(5, 8 + 9)`



(a) Left associativity `a == b == c` (b) Right associativity `a = b = c`

Figure 3.5: Associativity in AST

If an item[13] in a parser rule is given a caret (`^`) suffix, that item will become the root of the current rule's subtree. If for example, the 'OPERATOR' token in figure 3.3 is replaced by 'OPERATOR^', the rules will create trees as shown in figure 3.5.

Similarly to the caret suffix, an item suffixed by an exclamation point also alters tree construction. Any item suffixed like this will be removed from the resulting tree. This is useful for tokens no longer necessary in the syntax as the structure is already explicitly defined by the tree. For example, since statements are now nodes in the tree the semicolon seperator is no longer needed. Similarly the `end` statement is no longer required as the statements within a compound statement are child nodes of the compound instruction.

These two suffixes allow us to transform most of the rules correctly to trees. Sometimes more power is needed to create the exact tree required and ANTLR provides is with a multitude of macros we can use in semantic actions to manually control tree creation. The exact syntax for the creation of these trees is described in the ANTLR-manual[9].

The creation of the AST for the Ruby subset parsers was quite easy. Due to my inexperience with ANTLR I used manual tree creation when the simpler method of tree construction would have sufficed. This gave me greater insight into the workings of tree construction. Additionally an article on the ANTLR-website[10] helped greatly in this understanding.

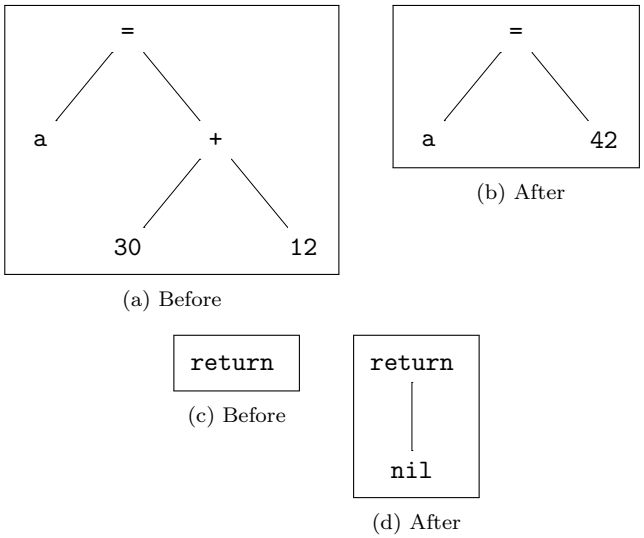
While I did make some mistakes[14], most rules did turn out well. During development I transformed the tree to a graphic using the Graphviz toolset. The way I implemented this transformation is described in section 3.3.2.

3.3 Walking the tree

Most 'compiler compiler' tools assist you up until the syntactic analysis phase has been completed. ANTLR goes beyond that and besides helping you in creating an AST also helps you to do *something* with it. Most people[15], when first encountering an AST want to write a method that walks the tree and extracts the necessary information. Another article on the ANTLR-website[11] describes three different methods. The first is the tree walker mentioned above, the second uses a heterogeneous AST[16] where each node knows how to handle itself and is responsible for its child nodes.

The third method, the one Terence Parr[17] advocates is a so-called *tree parser*. Tree parsers use techniques similar to the ones used in lexers and parsers to walk a 2-dimensional tree structure. You can do two things with tree parsers, you can transform a tree (AST-to-AST) and you can give no explicit output. The first is often used for optimization and unification of similar structures. For example in expressions one could collapse arithmetic nodes when all operands are constants as seen in figure 3.6a and 3.6b. Another example, from the alterations done in this project is shown in figure 3.6c and 3.6d. This adds the explicit `nil` to the `return` statement. In

[13] A literal, parser rule or token
[14] Such as adding a superfluous *EXPRESSION* node as the parent of all types of expressions
[15] myself included
[16] ANTLR supports the creation of heterogeneous trees
[17] Professor of CS at the university of San Francisco, the author of ANTLR



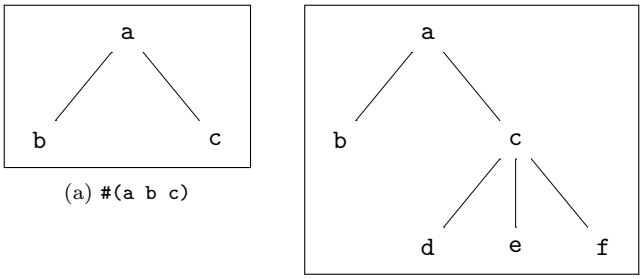
(a) Before

(b) After

(c) Before

(d) After

Figure 3.6: Example of AST alteration



(a) `#(a b c)`

(b) `#(a b #(c d e f))`

Figure 3.7: Tree representation in ANTLR

this project I used a single AST-to-AST tree parser to perform some semantic analysis and add implicit values where applicable.

Parsers which do not transform a tree, usually perform output via semantic actions. An expression evaluator may for example calculate the value of every node until it reaches the root node and print the results. In this project I used two of these kinds of parsers, one for the visualization of the AST and another for the output of program algebra.

Matching rules for trees are somewhat more complex than those for character streams (lexers) or token streams (parsers) since we are now matching two-dimensional data. While the basic syntax is still EBNF, a lisp-like syntax is added to match tree structures. ANTLR represents tree structures as shown in figure 3.7.

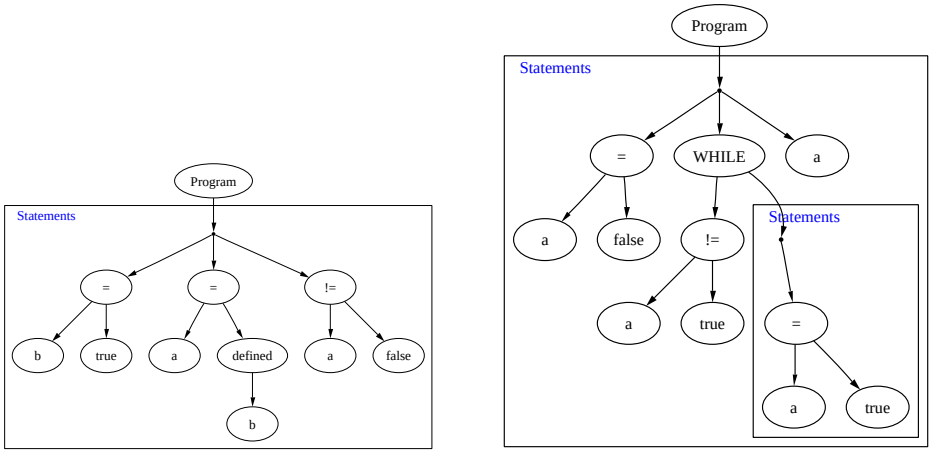
Matching a tree structure in ANTLR is similar to its string representation, for example to match a STATEMENTS node with zero or more statements as children would be matched by `#(STATEMENTS (statement)*)` or a Ruby class statement is matched as follows:

```
#(CLASS IDENTIFIER statements)
```

where CLASS is a token containing the class name, IDENTIFIER a token containing the parent class' name and the multiple statements are the class body (the statements rule works like the matching of a STATEMENTS node as in the first example).

3.3.1 Cleaning the tree

The tree parser which modifies the tree obtained from the syntactic analysis stage was named the *cleaner*, a misnomer but due to lacking inspiration I kept the name. In the end the tasks



(a) (b)

Figure 3.8: Example AST

this parser performs are few:

1. Several more stringent checks on the source language trying to capture semantic and syntactical errors.
2. Adds `Object` parent to a parentless class definition.
3. Adds `nil` to a `return` statement without explicit return value.
4. Removes the superfluous EXPRESSION node from the tree (See section 3.2.3).
5. Adds `self` to method calls without explicit destination object. (`foo(12)` becomes `self.foo(12)`).

3.3.2 Visualizing the AST

The second tree parser generates dot-files[18] via semantic actions. Several nodes in the AST generate different patterns in the output. Figure 3.8 shows the graphs generated from two simple programs.

The manner in which the graph visualizer emits the dot file source can be seen as a first trial for the emitting of program algebra. The code which emits the algebra is entirely embedded in the grammar file. This turned out to be a less than ideal choice since this made it impossible to use the same grammar for another purpose (such as emitting program algebra).

3.4 Emitting algebra

For the emitting of program algebra I decoupled the grammar from the methods generating the code. This way I was able to use a single grammar for all four of the subsets. A seperate Python module provides four classes: one for each of the Ruby subsets. Each class exposes methods corresponding to the different macros and projection functions described in chapter 2. The grammar calls the methods on an instance of one of these classes. Depending on the used class one of the four subsets is emitted. These classes inherit from one another only adding or redeclaring the methods which need to yield a different projection. The way the classes work is simple, every method corresponding to a macro or projection function returns a string constructed with the values given by the grammar containing the arguments needed.

Since each grammar rule in ANTLR corresponds to a method on the (tree)parser class ANTLR allows the grammar author to give rules both arguments and return values. The arguments can

[18] `dot` which is a part of the Graphviz toolset is a tool for the visualization of directed and undirected graphs. http://www.graphviz.org

be provided when a rule is invoked by another rule or when the parser is invoked by another program. Similarly the return values of rules can be used in semantic actions of the calling rule. The program algebra emitting tree parser uses return values to return a string containing the segment of program algebra the rule represents.

As an example, a `while` node contains two children: an `expression` node which is evaluated every iteration to determine whether the loop should continue and a `statements` node which contains the body of the loop. The rule for the `while` node invokes the rules for these two children to get their respective program algebra fragments. Afterwards a semantic action calls the `stmt_while` method on the instantiated class with the two fragments as arguments. The method inserts these fragments into the appropriate place of the `while` projection. When this is completed, the return value of the method is set as the return value of the `while` node and it is used as a fragment in whatever parent the node has.

3.4.1 A better way

ANTLR 3.0, a new version which provides several new features also includes an extra module which was previously not part of the ANTLR toolset. This module – called *StringTemplate* – allows us to generate formatted text-based output easily. This is exactly what the code emitter needs to do. StringTemplate is available in the ANTLR version I have used as a seperate library and usable via semantic actions. Unfortunately I did not discover the existence of this library until after the current implementation was practically complete, therefore I have not used it.

3.5 Testing the projection

While the development of most of the program went without any major problems, the resulting projection had to be tested. To easily test the translation I wrote several Ruby scripts with increasing complexity using every feature available in the four subsets. All these scripts had the same behaviour: if they ran correctly the last evaluation in the program yields `true`. Figure 3.9 shows an example test file. To test these scripts they had to be run through both Ruby and the PGA Toolset. To run them through the PGA toolset they first had to be projected to PGLEcr via PGLEcrv. I wrote a script which ran both the Ruby parser and the PGA simulator on all test files and checked the output. This allowed me to make changes to the projection and quickly test the effectiveness of them.

Debugging the projection was a rather difficult job as even relative simple programs can generate quite a bit of PGLEcr code. Many of the scripts resulted in over one thousand PGLEcr instructions, a single script even went far beyond that with 13394 PGLEcr instructions. Luckily fixing bugs in simple scripts often caused more complex scripts to also start working correctly. Debugging the entire projection took quite a bit of time. Had I not done so and just left the code as is, the development time would have been cut in half.

```
01: class Pair;
02:   def initialize(a,b);
03:     @e0 = a;
04:     @e1 = b;
05:   end;
06:
07:   def first();
08:     return @e0;
09:   end;
10:
11:   def second();
12:     return @e1;
13:   end;
14:
15:   def ==(p);
16:     if first() == p.first();
17:       if second() == p.second();
18:         return true;
19:       end;
20:     end;
21:     return false;
22:   end;
23:
24: end;
25:
26: a = true;
27: b = false;
28: foo  = Pair.new(a,b);
29: bar = Pair.new(b,a);
30:
31: res = false;
32:
33: if foo != bar;
34:   if foo.first() == bar.second();
35:     if foo.second() == bar.first();
36:       res = true;
37:     end;
38:   end;
39: end;
40:
41: res;
```

Figure 3.9: A Ruby test file

CHAPTER 4

Conclusion

In the end I am most satisfied with the results: the projection passes all tests and modifications to many aspects of the programs are easy. The ANTLR grammar files are easily readable and so far modifications have not required significant change elsewhere. ANTLR is a well designed tool and fun to use, I have already been looking for excuses to use it elsewhere.

4.1 Program algebra

The Ruby subsets are an interesting addition to program algebra. They have removed the distinction between primitive instruction sets and basic instruction sets. This is similar to one of the aspects of the object oriented paradigm,: the combination of behaviour (primitive instructions) and data (basic instructions) into a single entity (objects).

The projection proves program algebra is powerful enough to allow object oriented programming. But it can be argued that the Ruby subsets do not belong in the program algebra hierarchy: the step from Ruby to PGLEcr is large and somewhat complex. While most instruction sets in the PGA toolset introduce a single new concept, the projection from the Ruby Cores introduces an enormous amount of related concepts which could have been seperated into several new instruction sets.

4.2 Extensibility

The compiler is most extensible, Ruby Cores two through four prove this: minimal work was required to implement these once Ruby Core One was implemented. All parts of the of the compiler related to the different subsets inherit from a previous version one way or another. Implementing a Ruby Core Five – whatever it may do – should therefore not take too much effort.

4.3 Further improvements

Both the projection and the compiler implementing the projection are easily altered or extended. Here I mention a few interesting possibilities.

4.3.1 More language features

A lot of the language features of Ruby have been left out of the four subsets. One could add new Ruby Cores implementing more functionality of the Ruby language, for example:

Integers negative integers, more arithmetic operations, support for Bignum as well as Fixnum.

Other types strings, floating point numbers.

Mixins mixins are a language construct in Ruby providing a form of multiple-inheritance.

Modules support for multiple source files.

4.3.2 Improved compiler

ANTLR 3.0 uses a new algorithm for matching rules[1]. This makes several syntactic predicates in the current grammar redundant making the grammar files even more readable. Furthermore the StringTemplate functionality can be used for the generation of PGLEcrv instead of the current method.

4.3.3 Alternate Projection

The current projection maintains the complete object oriented structure. This is what dynamic languages like Ruby do. Static object oriented languages like C++ discard much of the objects meta data at compile time[2]. Creating a projection with similar behaviour could be interesting.

4.4 Acknowledgements

I would like to thank my supervisors Bob Diertens and Inge Bethke for their help during the weekly meetings and for the proofreading of my thesis. Further thanks go to Stephan Schroevers, who worked on a related thesis[3] and helped by proofreading and discussing program algebra. Sybren Stüvel found some typos in my thesis as well and I would like to thank him for taking the time to read it.

[1] Instead of finite lookahead parsing ($LL(k)$) ANTLR 3.0 supports infinite lookahead ($LL(*)$)
[2] Less so if Runtime Type Identification (RTTI) is enabled.
[3] Coincidentally, his thesis has the same page count as mine.

APPENDIX A

PGLEcrv

PGLEcrv adds two instructions to PGLEcr, the *variable goto* instruction and the *variable returning goto*. PGLEcrv enables us to have jumps in a program where the destination label is not known in advance. In [1] these two instructions are introduced as part of the intermediate projection language. In the original projection, the *variable returning goto* is projected using the `stackframe` focus. Since this focus is not introduced until PGLEcr is projected to PGLEc, it was needed to alter the projection somewhat.

A.1 Variable Goto

The *variable goto* instruction `##F[f]` did require a single modification. The projection is as follows:

$\psi($`##L[`f`]`$) =$
`+`f `== 1; ##L1; ... ; +`f `==` n`; ##L`n`; !; !`

where n is the highest label in the program. The projection simply walks through all possible destinations and checks the given focus against each, when it is found the jump is made. The single modification I made was the addition of two termination instructions to the end of the projection. This way, should the given focus have taken a value not part of the set of label values, the program terminates instead of exhibiting unexpected behaviour.

A.2 Variable Returning Goto

The *variable returning goto* instruction needed to have a major overhaul, due to the fact that the `stackframe` focus is not available. The projection is as follows:

$\psi($`R##L[`f`]`$) =$
`+`f `== 1 {; R##L1; ##L`i`; }; ... ; +`f `==` n `{; R##L`n`; ##L`i`; }; L`i

where n is the highest label in the program, and i is a new label every time this projection is invoked. The jump to label i is needed because otherwise after a jump returns, the following `+`f `==` x `{;` instructions are executed which could yield true as the focus may have changed during the execution following the returning jump.

A.3 Projecting PGLEcrv

I used Antlr for the projection of PGLEcrv to PGLEcr. There are only three instructions the lexical analyzer is interested in: the two new ones obviously and the label instruction[1]. During syntactic analysis, every instruction is parsed and turned into a string containing the new program. The projection does not use the highest used label plus one when a new label is needed, instead it maintains a set of the used labels and uses the first unused label as it is encountered.

[1] To determine the maximum label, n in the projections.

APPENDIX B

Formal grammar

The grammar of the four subsets of Ruby used in this thesis needed to be completely specified so parsing could be done. This appendix contains the specification of the grammars in EBNF. Since RC2-RC4 only add some new language features, only the altered/added rules will be listed.

B.1 Ruby Core One

```
program :== (statement)*

statement :== class_definition
            | method_definition
            | return_statement
            | if_statement
            | while_statement
            | expression ';'
            | ';'

class_definition :== 'class' IDENTIFIER ('<' IDENTIFIER)? ';'
                     (statement)*
                     'end' ';'

method_definition :== 'def' method_identifier '(' parameter_list ')' ';'
                      (statement)*
                      'end' ';'

method_identifier :== '=='
                    | '!='
                    | IDENTIFIER
                    | METHOD_IDENTIFIER

parameter_list :== (IDENTIFIER (',' IDENTIFIER)*)?

return_statement :== 'return' (expression)? ';'

if_statement :== 'if' expression ';'
                 (statement)*
                 ( 'else' ';'
                   (statement)*
                 )?
                 'end' ';'
```

```
while_statement :== 'while' expression ';'
                    (statement)*
                    'end' ';'

expression :== expression '=' expression
             | expression '==' expression
             | expression '!=' expression
             | expression '(' (expression (',' expression)*)? ')'
             | expression '.' expression
             | IDENTIFIER
             | INSTANCEVAR
             | '(' expression ')'
             | 'defined?' IDENTIFIER
             | 'defined?' INSTANCEVAR
             | 'true'
             | 'false'
             | 'self'
             | 'nil'

IDENTIFIER :== ALPHA (ALPHANUM)*

METHOD_IDENTIFIER :== IDENTIFIER ('?'|'!')

INSTANCEVAR :== '@' IDENTIFIER

ALPHANUM :== ALPHA | DIGIT

ALPHA :== 'a'...'z' | 'A'...'Z' | '_'

DIGIT :== '0'...'9'
```

B.2 Ruby Core Two

```
method_identifier :== '=='
                    | '!='
                    | IDENTIFIER
                    | METHOD_IDENTIFIER
                    | IDENTIFIER '.' method_identifier_simple
                    | '(' expression ')' '.' method_identifier_simple

method_identifier_simple :== '=='
                           | '!='
                           | IDENTIFIER
                           | METHOD_IDENTIFIER
```

B.3 Ruby Core Three

```
expression :== expression '=' expression
             | expression '==' expression
             | expression '!=' expression
```

```
              | expression '(' (expression (',' expression)*)? ')'
              | expression '.' expression
              | IDENTIFIER
              | INSTANCEVAR
              | CLASSVAR
              | '(' expression ')'
              | 'defined?' IDENTIFIER
              | 'defined?' INSTANCEVAR
              | 'defined?' CLASSVAR
              | 'true'
              | 'false'
              | 'self'
              | 'nil'

CLASSVAR :== '@' '@' IDENTIFIER
```

B.4 Ruby Core Four

```
expression :== expression '=' expression
              | expression '==' expression
              | expression '!=' expression
              | expression '+' expression
              | expression '(' (expression (',' expression)*)? ')'
              | expression '.' expression
              | IDENTIFIER
              | INSTANCEVAR
              | INTEGER
              | '(' expression ')'
              | 'defined?' IDENTIFIER
              | 'defined?' INSTANCEVAR
              | 'true'
              | 'false'
              | 'self'
              | 'nil'

INTEGER :== (DIGIT)+
```

Bibliography

[1] R. M. Geerlings (November 2003). *A Projection of the Object Oriented Constructs of Ruby to Program Algebra.*

[2] J.A. Bergstra and M.E. Loots (2002). *Program algebra for sequential code* Journal of Logic and Algebraic Programming, vol. 51, no. 2, pp. 125-156.

[3] J.A. Bergstra and I. Bethke (2002). *Molecular dynamics* Journal of Logic and Algebraic Programming, vol. 51, no. 2, pp. 193-214.

[4] B. Diertens (2004). *Molecular Scripting Primitives* electronic report PRG0401.

[5] B. Diertens (2003). *A Toolset for PGA* electronic report PRG0302.

[6] D. Thomas and A. Hunt (2001). *Programming Ruby: The Pragmatic Programmer's Guide.*

[7] A. Aho, R. Sethi and J. D. Ullman (1988). *Compilers: Principles, Techniques and Tools.*

[8] T. J. Parr and R. W. Quong (1995). *ANTLR: A Predicated*-LL(k) *Parser Generator* Software – Practice and Experience, Vol. 25(7), pp. 789-810.

[9] T. J. Parr Antlr *Reference Manual*
`http://antlr.org/doc/index.html`

[10] M. Barnett (2000). *Manual Tree Manual.*
`http://antlr.org/article/1137964510001/manual.tree.construction.txt`

[11] T. J. Parr (2004). *Translators Should Use Tree Grammars.*
`http://antlr.org/article/1100569809276/use.tree.grammars.tml`