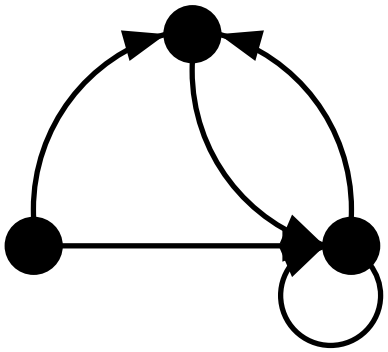
A Projection of the Object Oriented Constructs of Ruby to Program Algebra



Ruben M. Geerlings

Universiteit van Amsterdam

November 2003

Titel:	A Projection of the Object Oriented Constructs of Ruby to Program Algebra
Auteur:	Ruben M. Geerlings Van Nijenrodeweg 907, 1081 BJ, Amsterdam rgrlings@science.uva.nl
Studie:	Informatica Specialisatie Programmatuur
Faculteit:	Faculteit der Natuurwetenschappen, Wiskunde en Informatica Universiteit van Amsterdam
Begeleider:	Jan A. Bergstra Kruislaan 403, 1098 SJ, Amsterdam
Datum:	November 2003

Abstract

Ruby is a pure object-oriented (OO) programming language. It has many advanced features which are borrowed from languages such as Perl, Smalltalk and Scheme. Its OO constructs are simple and easy to learn. Ruby has no type checking. This paper describes a projection of a small subset of Ruby – focusing on the OO constructs – to Program Algebra (PGA). The projection to PGA can be seen as a theoretical compiler, optimized for readability instead of performance.

Keywords: Program Algebra; Object Oriented Programming; Ruby

Contents

List of Figures

Chapter 1

Introduction

Ruby is a relatively new object-oriented (OO) programming language. It has many advanced features which are borrowed from languages such as Perl, Smalltalk and Scheme. Although Ruby has a big number of language features, its OO core is simple and easy to learn. A number of features which make Ruby stand out:

- Ruby is a pure OO language: all data consists of objects, and all operations consist of method calls.
- Variables in Ruby are untyped: there is no type-checking, and they can hold any type of object. Variables do not need to be declared before they are used, they are created dynamically when they are assigned a value for the first time.
- Ruby is an interpreted language. Classes and methods are created dynamically. Methods can be added to a class or to a single instance of a class during run time.

This paper describes a projection of a small subset of Ruby to program algebra (PGA). The projection focuses on the OO constructs of the language, therefore many interesting features of the language are left out. But the subset can be considered the core of the language, and can be treated as a programming language on itself.

The projection to PGA can be seen as a theoretical compiler, optimized for readability instead of speed or size. Ruby is an interpreted language which means that there is no separate compile phase, instead Ruby programs are executed with the aid of an *interpreter* program. It is assumed that Ruby programs can (theoretically) be compiled into stand-alone programs which are behaviorally equivalent to their interpreted counterparts.

The first projection is of the most basic subset RC1 (Ruby Core One). It may be lacking important features which will be added gradually to RC1 in extensions RC2, RC3 and RC4. Although instructions in Ruby have many syntactic forms, only one form is used in the projections.

1.1 Motivation

The motivation for writing a projection of a subset of Ruby to program algebra is twofold.

On one hand, the development in program algebra reached a stage where MPP (molecular programming primitives) instructions combined with PGLEcm (control instructions and method calls) offers a language with objects and methods. But it lacks some essential features in order to be called an object oriented programming language according to the definition by Wegner [3]:

$$\text{OOP} = \text{Objects} + \text{Classes} + \text{Inheritance}$$

Classes and inheritance are missing in the previously mentioned language. The OO design of Ruby serves as a good role model for an OO language in program algebra.

On the other hand, Ruby is an interesting programming language in its own right. Its underlying principles are simple, which makes it relatively easy to learn. The interaction between classes and objects is not transparent however, and in this regard, a projection can serve as an aid to understanding the Ruby language.

The projection is based on the Ruby interpreter [1]. Assertions on the behavior of Ruby programs were tested by running experimental code. The goal was to find a simple model that explains all observed behavior.

An important source of documentation is [4], which covers the Ruby language in some detail. A special case of documentation is the source code of the Ruby interpreter itself. Besides the practical objection to reading through the interpreter code, it is the author's opinion that the actual implementation of the interpreter does not define the language. It would imply that any bugs in the interpreter are part of the language, and it would prevent the interpreter from improvement without an alteration to the language as well.

With the above thoughts taken into account, it must be stressed that this projection of part of the Ruby language has the goal to produce similar behavior of programs run on the interpreter as executed in program algebra. [2] The projection is therefore sometimes deliberately different from the interpreter when it favors readability.

[1] Ruby release 1.6.2

[2] The behavior of PGA programs is discussed in [1].

Chapter 2

The Projection Language

In [1] a program language is defined as a tuple (L, φ), where L is a collection of textual objects and φ a projection from L to PGA. A projection is a mapping of programs written in one particular language to another – lower level – language. [1] Every program in the projection framework can be projected to a PGA program. Projections of programs in high level languages are done in stages. If they can be projected to an intermediate language, it is more efficient to define the projection as a composition of projections.

This projection in stages also applies to real programming languages (for different reasons, such as platform independence). Instead of compiling programs to machine code, they output programs in an intermediate language, such as Java Bytecode or the Microsoft Intermediate Language.

This section describes the intermediate projection language (IPL) in the projection of Ruby subsets. It is assumed that the reader is familiar with the programming language PGLEcw [1] and the molecular programming instructions (MPP) as introduced in [2]. IPL extends the combination of PGLEcw and MPP with some advanced control instructions.

2.1 PGLEcw

PGLE programs are sequences of instructions, consisting of control instructions and an undefined collection of basic instructions. The goto instruction `##L`k, with k a natural number, moves control to the first occurence of the label instruction `L`k in the program. PGLE has test instructions which respond to a boolean value returned by a basic instruction. The boolean value determines whether to skip or execute the following instruction.

PGLEcw extends PGLE with conditional blocks (if-then-else instructions) and while loops.

[1] A projection to a higher level language is called an *embedding*.

2.2 MPPV

As stated above, PGLEcw only defines control instructions. The collection of basic instructions that is used is defined in MPPV. MPP stands for *molecular programming primitives*; a collection of instructions to create and manipulate objects and their connections, suitable for modeling the memory state of object oriented languages.

MPPV adds values to MPP, which is necessary to store integers and booleans.

2.3 Extensions

The combination of PGLEcw and MPPV is extended with a number of instructions which make the projection of Ruby subsets easier.

2.3.1 PGLEcws

To make labels more descriptive, a new label instruction `L`s is added that holds a string s instead of a number. The matching goto instruction `##L`s is added as well.

The projection of these string labels to numerical labels requires a dictionary D^X of all the strings in program X. The dictionary is an ordered set of all unique strings in the program. If the dictionary is of size n, D^X_1 through D^X_n denote all the entries. $D^X(s)$ returns the index i of s in the dictionary (when $D^X_i = s$).

The projection ψ of PGLEcws to PGLEcw applied to each instruction in program X is defined below:

- Suppose n is the maximal numerical label in program X.
  $\psi($`##L`$s)$ =
  `##L`$(D^X(s) + n)$
- $\psi($`L`$s)$ =
  `L`$(D^X(s) + n)$

ψ leaves all other instructions unchanged.

The projection ensures that all string labels will be projected to unused numerical labels.

2.3.2 MPPVs

MPPVs is an extension of MPPV that includes string constants. MPPV has two types: integers and booleans. Strings are a third type in MPPVs, the types of instructions in MPPVs do not change.

A string is a sequence of characters, [2] depicted by placing the characters between quotes. For example, the code `x.+f:string; x.f = "apple"` adds a

[2] The set of characters should include at least all lower case and upper case letters, all numericals and the ':' character.

string field f to atom `x` and assigns the value `apple` to that field. The default value of a string is the empty sequence of characters.

2.3.3 Variable Goto

Goto instructions in PGLEcws always jump to a fixed instruction in the program. In [2] a goto instruction is introduced which destination depends on the value of a focus. The focus must be set with instructions in MPPV. The syntax is as follows:

- `##L[`f`]` , where f is a focus.

The variable goto instruction checks the value of f upon resolution and jumps to the first occurence of the label with the same value.

The projection ψ of the variable goto instruction is done by simply selecting among all the possibilities. The definition is repeated below:

- Suppose n is the maximal label in program X:
  $\psi($`##L[`f`]`$) =$
  `+f == 1; ##L1; +f == 2; ##L2; ... ;+f == n; ##Ln`

The variable goto instruction can be adapted to an environment with strings when PGLEcws is combined with MPPVs. The focus must be compared to each string occuring in the program X:

- Suppose n is the maximal label number in program X, and D^X is the dictionary of string labels in program X of size m.
  $\psi($`##L[f]`$) =$
  `+f == 1; ##L1; +f == 2; ##L2; ... ;+f == n; ##Ln;`
  `+f ==` D^X_1`; ##L`D^X_1`; +f ==` D^X_2`; ##L`D^X_2 `; ... ;+f ==` D^X_m`; ##L`D^X_m

2.3.4 Returning Goto

In [2] the returning goto and the return instructions are defined with the following syntax:

- The returning goto instruction:
  `R##L`l
- The return instruction:
  `R`

The returning goto instruction `R##L`l performs a jump to label l. When it encounters a return instruction, the program continues at the instruction following the last executed returning goto instruction.

It is possible that multiple returning goto instructions are executed without a return instruction in between. Therefore, a stack is necessary to store the

return positions. A return instruction then jumps to the location stored on the top of the stack and pops the stack afterwards.

The stack datastructure is created by instructions in MPPV and is called the *stackframe*. There are two important operations: *stackframe-up* creates a new layer to store data on, *stackframe-down* removes the top layer. The operations are done by executing sequences of instructions in MPPV, as shown below. The `stackframe` is a dedicated focus that points to the top layer of the stack.

- $\varphi_{stackframe-up} =$
  ```
  aux = new; aux.+back; aux.back = stackframe;
  stackframe.+next; stackframe.next = aux;
  stackframe = aux
  ```

- $\varphi_{stackframe-down} =$
  ```
  stackframe = stackframe.back;
  stackframe.-next
  ```

Below is the translation ψ of the returning goto and the return instruction. [3] The translation of the returning goto requires a new label. To prevent clashes with user defined labels, the label numbers in the translation need to be raised by n, where n is the maximal label number in the program.

- $\psi_i(u)$ denotes the projection of the i^{th} instruction u in program X. n is the maximal label number in program X. l is a label constant, either a number or a string.
  $\psi_i($`R##L`$l) =$
  $\varphi_{stackframe-up}$`;`
  `stackframe.+label:int;`
  `stackframe.label =` $(i+n)$`;`
  `##L`l`;`
  `L`$(i+n)$

- The return instruction:
  $\psi($`R`$) =$
  `-stackframe/label;!;`
  `label = stackframe.label;`
  $\varphi_{stackframe-down}$`;`
  `##L[label]`

2.3.5 Variable Returning Goto

The last addition to the projection language is a combination of the returning goto and the variable goto: the variable returning goto.

- `R##L[`f`]`, where f focuses a value.

[3] The translation here is different from the one in [2]. In the referenced definition, returning goto instructions with the same label always return to the leftmost instruction. Whereas in the definition here, all returning goto instructions return to themselves.

The variable returning goto is defined in terms of the variable goto, which in turn can be projected to PGLEcws with MPPVs.

- Suppose n is the maximal label number in program X.
  ψ_i(`R##L[f]`) =
  $\varphi_{stackframe-up}$`;`
  `stackframe.+label:int;`
  `stackframe.label =` $(i+n)$`;`
  `##L[f];`
  `L`$(i+n)$

This last instruction allows the program to jump to a destination in the program which is not known in advance, and return afterwards. Exactly the requirements for the projection of method calls in an object oriented language where the applied method can not be determined in advance – as will be explained later on.

PGLEcws with MPPVs and these extra instructions make up the intermediate projection language (IPL).

Chapter 3

Ruby Core One (RC1)

The first subset of Ruby is Ruby Core One (RC1). RC1 introduces all the basic constructs of OO languages: classes, methods and objects.

These constructs in RC1 are not very extensive, there is only one type of each construct. For example, Ruby offers three different types of methods, but RC1 only has one type: the instance method. Only the most common types are in RC1.

Besides OO constructs, RC1 also contains some procedural programming features of Ruby: if-then-else statements and while loops.

Although the number of constructs in RC1 is limitied, it serves as a good point of departure for a simple and pure OO language.

All of RC1's constructs are explained, beginning with the most important concepts. The projection of an entire RC1 program is given at the end of this section.

3.1 Classes

It comes as no surprise that OO programming revolves around *objects*. Objects are abstract data elements which are compound of other data. The data can be basic such as integers or strings, or can consist of other objects. In OOP terminology, the named data belonging to objects are *fields* or *instance variables*.

A *class* is a type of objects. Every object belongs to a particular class; the object is called an *instance* of that class.

Traditionally, classes define fields and functions. Every instance of the class has the same fields. Functions belonging to a class are called *methods*. Often, these have exclusive access to the instance data, and are the only way to modify the state of objects.

Ruby breaks with the tradition of seperating data from methods. In Ruby there is no separate declaration of the instance variables, instead they are created dynamically during the lifetime of an object. Often the instance variables are created in the *initialize* method, immediately after the creation of an object,

but this does not need to be the case. Instance variables are created when they are assigned a value for the first time inside any method.

3.1.1 Example

A simple example of a class is `Pair`. When a new instance of `Pair` is created, the `initialize` method is called, which instantiates it with two elements in instance variables `@e0` and `@e1`.

After an instance of `Pair` is created, the only way to access the elements is by calling methods on the object. The methods `first` and `second` retrieve the elements.

The method `equals` has another `Pair` object as argument and checks whether the calling object and the argument contain the same elements.

Note that in this definition of `Pair`, it is impossible to change the elements of an instance after it has been created.

```
class Pair;

  def initialize(a,b);
    @e0 = a;
    @e1 = b;
  end;

  def first();
    return @e0;
  end;

  def second();
    return @e1;
  end;

  def equals(p);
    if first() == p.first();
      if second() == p.second();
        return true;
      end;
    end;
    return false;
  end;

end
```

3.1.2 Class Definition

Ruby class definitions can appear anywhere in the program. They do not need to be in separate files. Classes are created dynamically during the execution of a program. In Ruby, classes are objects like any other object. They are instances of the class `Class`. They are created when a class definition is encountered in the program. Class definition blocks can occur multiple times for the same class,

this way methods can be added or overwritten after the class has already been defined.

Block Blocks are consecutive parts of code which can be regarded as units. Examples are class definitions, method definitions and while loops. In Ruby, all blocks are delimited by a closing `end` instruction. The closing instruction should *match* the opening instruction. This means that they are in the same nesting level, as blocks can be nested inside other blocks. [1]

The space inside class declarations is not reserved for the declaration of methods. Every statement that can be put in the *main section* (code outside of class definitions) can be inside the class definition as well. Variables inside class definitions have their scope limited to that class definition.

Scope A scope is the visible range of a variable or constant. It is the part of the code where the variable or constant can be accessed.

3.1.3 Inheritance

An essential property of classes and of OOP in general is *inheritance*.

In the class definition a superclass relation can be defined with the `<` operator. For example: `class Bulldog < Dog` begins a class definition for the `Bulldog` class which is a *subclass* of class `Dog`. `Dog` is called the *superclass* of `Bulldog`. If the superclass operator is not present, the default superclass is `Object`.

Conceptually, a subclass is a specialization of the superclass. Objects of the subclass *inherit* all the methods of the superclass. The subclass can also define new methods. Methods already defined in the superclass can be given a new definition in the subclass, this is called *overriding* a method.

A class can have any number of subclasses. But each class has only one superclass. [2]

When a method is called on an object, first the object's own class is searched for a definition of that method. If it is not present, the superclass is searched in turn. Then the superclass of the superclass is searched and so on until the `Object` class is reached which has no superclass.

Because a subclass defines methods in addition to the inherited methods of the superclass, it is said to *extend* the superclass.

3.1.4 Example

In the `Pair` example, the contents of a pair object can not be changed after the object is created. Suppose it is desirable to have a class of pairs that can.

[1] In program code blocks are indented to make them more distinguishable.

[2] Ruby does not have *multiple inheritance*. It has *mixins*, a similar mechanic not discussed here.

A new class `MutablePair` can extend the old `Pair` class. Instances of `MutablePair` inherit the methods `initialize`, `first`, `second` and `equals` from `Pair` so they do not need to be defined again.

```
class MutablePair < Pair;

  def setFirst(x);
    @e0 = x;
    @changed = true;
  end;

  def setSecond(x);
    @e1 = x;
    @changed = true;
  end;

end
```

Instances of `MutablePair` which have their elements changed get a new variable `@changed`, which is set to true. This instance variable is created when it is assigned a value for the first time.

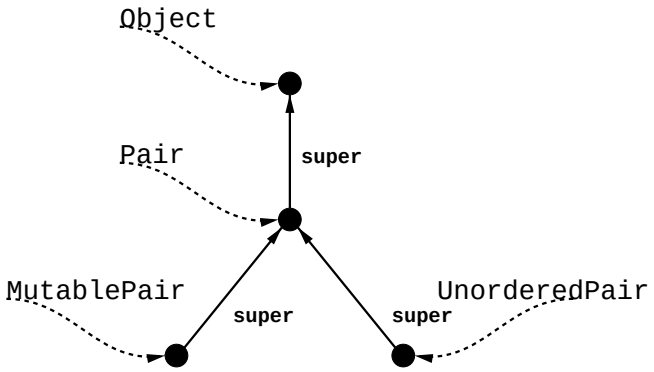
Another example is `UnorderedPair`. This class of pairs does not care about the order of its elements. The order of the instance variables `@e0` and `@e1` is only important when comparing pairs to other pairs with the `equals` method. `UnorderedPair` redefines `equals` and inherits the other methods of `Pair`.

```
class UnorderedPair < Pair;

  def equals(p);
    if first() == p.first();
      if second() == p.second();
        return true;
      end;
    end;
    if first() == p.second();
      if second() == p.first();
        return true;
      end;
    end;
    return false;
  end;

end
```

The class hierarchy of `Pair` and its subclasses is depicted in figure 3.1.

Class In Ruby, a class consists of a name and a collection of methods which can be applied on instances of the class. Methods can be inherited from one parent class. Instance variables are created dynamically inside methods.

Figure 3.1: The class hierarchy of Pair



3.1.5 Syntax

- Class definition:

  `class` C [`<` H]`;`u_1`;...;`u_k`;end`

  C is the name of the class, beginning with a capital. The optional part `<` H makes it extend a class H. If there is none, the default super class is `Object`. Instructions $u_1 \ldots u_k$ make up the body of the class definition.

3.1.6 Projection functions

- $\psi(X)$ maps a sequence of instructions X from RC1 onto the output language IPL.
- $\psi_{x_1,\ldots,x_n}(X)$ is a function restricted to contexts x_1 trough x_n. If there is no context subscript, the mapping doesn't care about the context.

Context A context is an environment in which the mapping takes place. The standard context is *main*. When the program enters a class definition block, it switches to the *class* context. When it enters a method definition, it switches to the *method* context.

- $\varphi_x(v_1, .., v_n)$ is a function without statements as input, but instead any number of parameters. It projects to a block of code in the output language. This type of projections can be seen as macros, because it expands to a piece of code and substitutes the parameters literally inside the code.

3.1.7 Projection of Classes

To implement the scope of variables inside class definitions, the *stackframe* mechanism is used. This is the same stackframe as used in the *returning goto*

translation. A *stackframe-up* section is executed before the program enters the class definition block to create a new layer on the stackframe. The `self` reference is set to the class object. (Inside methods `self` refers to the calling object, as is usual in other OO languages. [3]) A field `lv` is created to store all the local variables inside the class definition block.

Because class definition blocks can be repeated for the same class, it checks whether a class object has already been created – the presence of a `class` field reveals that it is an instance of a class and has already been created. If it is, the code that creates a new class is skipped. The call C `= Class.new(`H`)` creates a new `Class` object that has an instance variable `super` pointing to the superclass H.

- C and H are variables starting with a capital. The part between brackets (`[` and `]`) is optional.

  ψ_{main}(`class` C [`<` H]`;`u_1`;...;`u_k`;end` [4]) =
  [if there is no extension substitute `Object` in H]

  -C`/class {;`
  ψ(C `= Class.new(`H`));`
  `};`
  $\varphi_{stackframe-up}$`;`
  `stackframe.+ self; stackframe.self =` C`;`
  `stackframe.+ lv; aux = new; stackframe.lv = aux;`
  $\psi_{class}(u_1)$`;...;`$\psi_{class}(u_k)$`;`
  $\varphi_{stackframe-down}$`;`

Figure 3.2 shows a class being defined. The dots represent atoms in the projection language. The solid arrows are fields connecting the atoms and the dotted arrows represent foci in use by the program.

3.2 Method Definitions

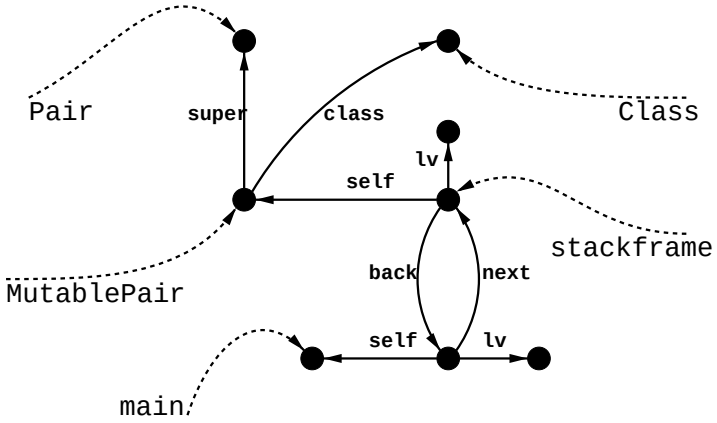
The only type of methods in RC1 is the *instance method*. Instance methods can be called on instances of the class (and extending classes) for which they are defined.

Method A method is a sequence of instructions which is executed as a unit. Methods are applied on objects. The instructions can manipulate the instance variables of the object. It has objects as input, called *arguments*. After execution, it supplies an object as result: the *return* object.

Methods can be defined outside of class definitions (in the main section), in that case the implicit class is `Object`. Every other class inherits methods from

[3] In some languages the *self* reference is called *this*.
[4] The end statement should always match the opening statement.

Figure 3.2: Program state durint definition of MutablePair



`Object`, thus these methods can be applied on objects of any type. Therefore, methods defined in the main section of the program are often called *functions*, [5] because there is no obvious relation to a particular class. [6]

An important difference between Ruby and other OO languages is that methods in Ruby are not typed. The arguments can be instances of any class, and the type of return object is not announced either.

When the last instruction of the method block is executed, execution of the program resumes at the position where the method was called. The method can return sooner if it encounters a `return` statement. An optional argument to this instruction is an expression that evaluates to a return object. The default return object is `nil`. [7] If the execution of the method is not ended by a `return` statement, the return object is provided by the last evaluated expression.

Expression An expression is an instruction which – when executed – yields an object. Expressions can occur inside other instructions, where an object is expected. After an expression is evaluated, the result is put in a dedicated focus `result`.

3.2.1 Example

The `equals` definition was defined in class `Pair`. Here is another way to define the method. Because `second() == p.second()` is an expression, it can be used

[5] In Ruby, these functions are labeled *private*, which means they can only be called on the *self* object. In RC1 the private modifier is ignored.

[6] But ignorance to this can lead to dangerous effects as all internal data of calling objects can still be manipulated inside of 'functions'.

[7] The nil object is an instance of the NilClass and its purpose is to to denote an unknown value, in tests it evaluates to a negative value.

as argument to the return statement. This method yields the same result as the previous definition.

```
def equals(p);
  if first() == p.first();
    return second() == p.second();
  else;
    return false;
  end;
end;
```

When the return statements are omitted, the method returns the last evaluated expression. The following definition demonstrates this idea.

```
def equals(p);
  if first() == p.first();
    second() == p.second();
  end;
end;
```

3.2.2 Syntax

- Method definition:

  `def` *m*`(`*p1*`,...,`*pn*`);`u_1`;...;`u_k `;end`

  m is the identifier of the method and *p1* to *pn* are the names of the arguments. Inside the method definition body these refer to the actual objects which are supplied by the method call.

  Instructions $u_1 \ldots u_k$ make up the body of the method defintion.

- A special statement which only occurs in method definitions is the return instruction:

  `return` *expr*

- Without an argument, it returns the `nil` object.

  `return`

3.2.3 Projection

The projection of methods is based on [2]. The projection is adapted to a setting where method names are not mapped one-to-one to their definition. There can be many methods with the same name in a class hierarchy. The method which should be called is only known at run time when the class of the calling object is determined. (This is called *polymorphism*).

When the method is defined in the main section, `stackframe.self` points to the `main` object, and consequently, the method is added to the `Object` class.

Methods are added to a branch of the class object. All instance methods are stored in C`.im`, where C is the class name and `im` is the field that points to the object that holds all the instance methods.

Whenever the method is called the program jumps to the label instruction L($2i$) in the definition below. The jump before it: ##L($2i+1$) is necessary to prevent the program from executing the method when it encounters the definition. The label number is created by numbering all instruction in the RC1 program and using the number of the method definition instruction as a unique label number.

After a jump to the method, before it executes the instructions of the method block, the local variables are initiated with the method arguments and the self reference is updated. The foci `arg1` to `arg`n and `self` are supplied by the method call.

- i is the position of the method definition instruction in the program.

  $\psi_{class,main}$ [8] (`def` *m(p1,...,pn)*`;`u_1`;...;`u_k `;end`) =
  `self = stackframe.self;`
  `+self==main {;cl = Object;}{;cl = self;};`
  `-cl/im {;cl.+ im; aux = new; cl.im = aux;};`
  `methods = cl.im;`
  `method = new;`
  `methods.+` m`;`
  `methods.`m `= method;`
  `method.+ label:int;`
  `method.label =` $(2i)$`;`
  ##L$(2i+1)$`;`

  L$(2i)$`;`
  `arguments = new; arguments.+` *p1*`;...;arguments.+` *pn*`;`
  `arguments.`*p1* `= arg1;...;arguments.`*pn* `= arg`n`;`
  `stackframe.+ lv; stackframe.lv = arguments;`
  `stackframe.+ self; stackframe.self = self;`
  $\psi_{method}(u_1)$`;...;`$\psi_{method}(u_k)$`;`
  `R;`

  L$(2i+1)$

The return instructions can appear with or without an expression as argument.

- *expr* represents an expression.

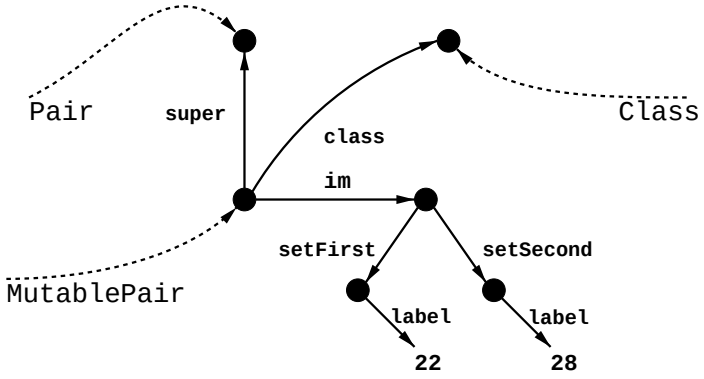
  ψ_{method}(`return` *expr*) =
  ψ(*expr*)`; R`

- Without an argument, `nil` is returned.

  ψ_{method}(`return`) =
  `result = nil; R`

[8] Methods can only be defined inside the main section or a class definition.

Figure 3.3: Program state after definition of methods in MutablePair



3.3 Method Calls

Method calls are messages sent to objects to perform some kind of computation with the data belonging to that object. When the method is not defined in the object's class, the object's superclasses are searched up the class hierarchy until the search reaches the `Object` class, which has no superclass. If the method isn't found there, the program must terminate.

The instance method `new` of class `Class` is defined on all classes. The method creates a new instance of that class, then it calls the `initialize` method on the newly created object to initialize the instance variables. The default `initialize` in class `Object` does nothing.

3.3.1 Example

An example of the creation of new objects is the following. Two new objects `a` and `b` are created and put in a `Pair` together. The last instruction checks if the first element of the pair is equal to `a` (and does nothing with the result).

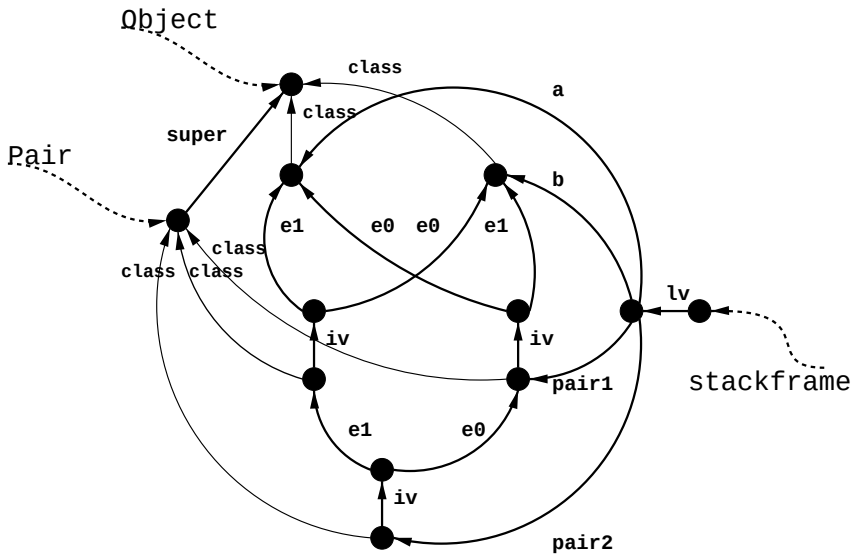
```
a = Object.new();
b = Object.new();
pair1 = Pair.new(a,b);
pair1.first() == a;
```

Method calls are expressions, because they always return one object. Therefore, method calls can also be placed inside other instructions, where an object is expected. This is shown in the examples above, where the method calls are used in assignment and test instructions.

Some other examples:

```
pair2 = Pair.new(pair1,Pair.new(b,a));
pair2.first().first() == a;
pair2.second().second() == b;
```

Figure 3.4: Program state after creation of objects



`pair2.first().first()` retrieves the first element of `pair2`, which is `pair1` and gets the first element of that object, which is `a`. The second test fails, because the second element of the second element of `pair2` is `a`.

The objects created in these examples are depicted in figure 3.4.

3.3.2 Syntax

- The method call without a prefix is sent to the object referred to by `self`. Expressions *exp1* to *expn* supply, when evaluated, the actual arguments.

  m`(`*exp1*`,...,`*expn*`)` (*expr*) [9]

- The prefixed version:

  exp0`.`*m*`(`*exp1*`,...,`*expn*`)` (*expr*)

  exp0, when evaluated, supplies the object to which the method call is sent.

3.3.3 Projection

The *stackframe* keeps track of the scope of variables. We have already seen its use in seperating variables in the main section from those in the class definitions. It is also an essential data structure to keep track of the local variables in (nested) method calls.

[9] Every instruction tagged with (*expr*) can be used as an expression in other instructions.

The variable returning goto instruction is used to jump to the body of the method. When it encounters an `R` instruction, it looks up the label in the stackframe, and jumps back to where the method was called. The stackframe also serves to store the `self` reference and the local variables – including the method arguments. The foci `arg1` to `arg`n are assigned to the method arguments which are subsequently put in the local variables when the program jumps to the method block. The same happens to `self`.

- A method call without a prefix is a shortcut for sending a call to `self`.

  $\psi($*m*`(`*exp1*, ..., *expn*`)`$) =$
  $\psi($`self.`*m*`(`*exp1*, ..., *expn*`)`$)$

- First, all the expressions are evaluated to objects. Then the macro *search-instance-method* is executed. If it does not find the method, the program terminates, otherwise the method is focussed by `method` which holds the jump label.

  The `result` focus is initialized to `nil` – if no expression is evaluated in the body of the method, `nil` is the return object.

  $\psi($*exp0*`.`*m*`(`*exp1*, ..., *expn*`)`$) =$
  $\psi($*exp0*$)$`;x = result;`
  $\psi($*exp1*$)$`;arg1 = result;...;`$\psi($*expn*$)$`;arg`n` = result;`
  $\varphi_{search-instance-method}($*m*$)$`;`
  `self = x;`
  `result = nil;`
  `label = method.label;`
  `R##L[label];`

- The actual method name m is substituted in the macro. Focus `x` is the object to which the method call is being sent.

  $\varphi_{search-instance-method}(m) =$
  `cl = x.class;`
  $\varphi_{search-supers}$`(cl,im,`*m*`);`
  `+found == false;!;method = res`

- This macro searches the class hierarchy for a method i in field s of class x. If there is no field s, the class is skipped, and the search continues with the *super*. If something is found, the result is put in focus `res`, and `found` is set to `true`.

  $\varphi_{search-supers}(x, s, i) =$
  `loop = true`
  `found = false`
  `sp =` x`;`
  `+loop == true{*;`
  `  +sp/`s{`;br = sp.`s`;`
  `    +br/`i{`;loop = false; found = true; res = br.`i`;};};`

```
  +sp/super{;sp = sp.super;}{;loop = false;};
*}
```

3.4 Local Variables and Instance Variables

Local variables are variables which exist during a method call or a class definition. When the program enters a method block or class definition block, all local variables in use are stored on the stack. They can not be accessed until the program returns from the block. New variables are only accessible inside the block, and are discarded when the program exits the block.

Variable A variable is an identifier with a certain scope, that points to an object.

Method arguments are local variables which are initialized at the beginning of a method call.

Recursive methods create new layers on the stack for every method call, each with its own distinct variables. This makes it possible to have many different variables with the same name in the same method during the execution of a program.

Recursive A method is recursive when it calls itself sometime during its execution.

Instance variables are variables which are bound to the scope of an instance object. Every method call on that object manipulates the same instance variables. It is not possible to directly access instance variables other than the instance variables of the object refered to by `self`.

Local variables and instance variables are both expressions and can occur inside other instructions.

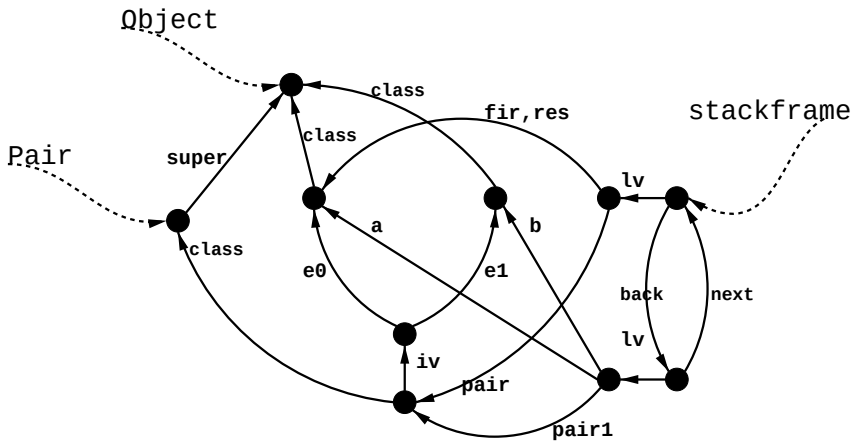
3.4.1 Example

Here is a definition of method `switch`.

```
def switch();
  temp = @e0;
  @e0 = @e1;
  @e1 = temp;
end;
```

When defined in class `Pair`, it changes the position of the elements in pair objects. The local variable `temp` is created every time the method is called. The instance variables `@e0` and `@e1` are created during initialization of the `Pair` object.

Figure 3.5: Program state during execution of recursiveFirst



The next method `recursiveFirst` is defined in the main section and expects an instance of class `Pair` as argument: [10]

```
def recursiveFirst(pair);
  fir = pair.first();
  if fir.class() == Pair;
    res = recursiveFirst(fir);
  else;
    res = fir;
  end;
  return res;
end;
```

`recursiveFirst` is an example of a recursive method. It takes the first element of a pair object and stores it in the local variable `fir`. If that is also an instance of class `Pair`, the method is called recursively on `fir`. [11] The `res` variable holds the result of the method. Every method call has distinct copies of the local variables `fir`, `res` and the method argument `pair`.

```
a = Object.new();
b = Object.new();
pair1 = Pair.new(a,b);
recursiveFirst(pair1);
```

[10] There is no type checking on method arguments, therefore the argument can be of any type, but when the program tries to call an undefined method on that object the program must terminate.

[11] Note that it is *not* possible to end up in an infinite loop with instances of class *Pair*. With *MutablePair* objects this *is* possible.

Figure 3.5 shows part of the program state during execution of the code above. Note that in this example the method does not call itself recursively to keep the size of the picture within bounds.

The argument `pair` is a local variable and holds the `pair1` object in this method call. `fir` is assigned to the first element of `pair`.

The first test `fir.class() == Pair` fails, and `res` is set equal to `fir`. When the method returns, the return value will be equal to `res`, which is the same object as `a` as can be verified in the picture.

3.4.2 Syntax

Variable names start with a lower case letter to distinguish them from class names.

- Local variables:

  x $\quad$ $(expr)$

- Instance variables:

  `@`x $\quad$ $(expr)$

3.4.3 Projection

Both local variables and instance variables are created when they are assigned a value for the first time.

To check whether a *local variable* exists, the field `lv` on top of the *stack-frame* is inspected. If there is a field with the same name as the local variable, the object referred to by that field is returned, otherwise the program must terminate.

For *instance variables* a field `iv` of the `self` object is inspected. If the variable has not been created, the `nil` object must be returned. Only instance objects with at least one instance variable have a field `iv`, so its existence must be tested first.

- $\psi(x) =$

```
localvars = stackframe.lv;
-localvars/x;!;result = localvars.x
```

- $\psi($`@`$x) =$

```
self = stackframe.self;
-self/iv{;result = nil;}{;instancevars = self.iv;
  -instancevars/x{;result = nil;}{;result = instancevars.x;};}
```

3.5 Constants

Constant A constant is an expression that always returns the same object.

`true` and `false` are boolean constants used by test expressions.

`nil` is the return value of a method which does not evaluate any expressions. It is also the value of an undefined instance variable.

All the constants above are objects like any other and are instances of classes.

Class names are also constants. Once a class has been created the class name represents the class object. Class names always begin with a capital letter to distinguish them from local variables.

`self` is a keyword denoting an object, like the previously mentioned constants. The value of `self` can not be changed by the programmer. But it is not a real constant, because its value changes every time the program changes scope. In the main section of the program `self` refers to a special instance of `Object: main`. Inside class definitions, `self` refers to the class object. Inside methods `self` refers to the calling object.

3.5.1 Syntax

- `self` (*expr*)
- `true` (*expr*)
- `false` (*expr*)
- `nil` (*expr*)
- *C* is an identifier of a class, starting with a capital letter:
  C (*expr*)

3.5.2 Projection

The projection of constants is easy because the focus in MPP that refers to the constant object has the same name as the constant in RC1. The value of `self` is stored on the *stackframe* because it changes when the program changes scope.

- $\psi($`self`$) =$
  `result = stackframe.self`
- $\psi($`true`$) =$
  `result = true`
- $\psi($`false`$) =$
  `result = false`
- $\psi($`nil`$) =$
  `result = nil`
- $\psi(C) =$
  `result =` *C*

3.6 Assignments

Assignment instructions create or update a variable. Any variable can be created this way. This is not a surprising fact concerning local variables. But it is special to Ruby to have assignment instructions create *instance variables* where other languages reserve a space in the class definition to define them.

Because instance variables are created during a method call, it is possible that different instances of the same class have a different number of instance variables. This can be used as a way to distinguish instances of a class. For example, in the class `MutablePair`, only instances which have been modified have an instance variable `@changed`.

3.6.1 Example

The method `changed?`, when defined in class `MutablePair` checks whether the pair object has been altered by `setFirst` or `setSecond`. The variable `@changed` is only created in a call to one of these two methods.

```
def changed?();
  if defined? @changed;
    return true;
  else;
    return false;
  end;
end;
```

Because the value of an undefined instance variable is `nil`, which is considered *false*, the method can be simplified accordingly:

```
def changed?();
  @changed;
end;
```

3.6.2 Syntax

x is an identifier, starting with a lower case letter.

- Local variables:

  x `=` *expr*

- Instance variables:

  `@`x `=` *expr*

- Tests whether a local variable has been defined. Returns the boolean value `true` if defined, `false` otherwise:

  `defined?` x (*expr*)

- Tests whether an instance variable has been defined. Returns the boolean value `true` if defined, `false` otherwise:

  `defined? @`x (*expr*)

3.6.3 Projection

In MPP, a field introduction instruction `x.+ f` is ignored when the field already exists. This makes the projection easy. Every assignment in RC1 will be projected to code that introduces a new field, regardless of whether it is the first assignment or not. Only the first time will the field be added.

The `defined?` instruction in Ruby can be applied to every expression. In RC1 I restrict its use to variables.

- The assignment of an arbitrary expression to local variable x:

  $\psi(x$ `=` $expr) =$
  $\psi(expr)$`;`
  `localvars = stackframe.lv;`
  `localvars.+` x`;`
  `localvars.`x `= result;`

- The assignment of an arbitrary expression to instance variable x:

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

- $\psi($`defined?` $x) =$
  `localvars = stackframe.lv;`
  `-localvars/`x`{;result = false;}{;result = true;}`

- $\psi($`defined? @`$x) =$
  `self = stackframe.self;`
  `-self/iv{;result = false;}{;instancevars = self.iv;`
  `-instancevars/`x`{;result = false;}{;result = true;};}`

3.7 Conditional Statements and While Loops

Two well known statements in imperative programming languages are introduced in this chapter: the `if-then-else` statement, and the `while` loop.

Both types of statements evaluate a boolean expression to decide whether or not to execute a section. In Ruby the objects `false` and `nil` are considered negative. Every other object is considered positive.

3.7.1 Example

The conditional statement has been used in previous examples. Here is an example with the `while` loop. In a previous example the method `recursiveFirst`

was showed. This method receives a `Pair` object which elements can also be `Pair` objects. It repeatedly takes the first element of the first element until the element is no longer an instance of `Pair`. This example defines a method `iterativeFirst` which uses the `while` loop instead of a recursive method call to execute these steps an unbounded number of times.

```
def iterativeFirst(pair);
  fir = pair.first();
  while fir.class() == Pair;
    fir = fir.first();
  end;
  return fir;
end;
```

3.7.2 Syntax

- `while` *expr*`;`u_1`;...;`u_k`;end`

  The body of the `while` loop, $u_1 \ldots u_k$, is executed repeatedly as long as the test expression *expr* evaluates to a positive value.

- `if` *expr*`;`u_1`;...;`u_k`[;else;` u_{k+1}`;...;`u_l`];end`

  The first section after an `if` statement is executed if *expr* evaluates to a positive value. If there is an `else`-section, that section is executed when the `if`-section is not. [12]

- The object equality test expression:

  exp0 `==` *exp1* $\qquad$ (*expr*)

  Tests whether two expressions evaluate to the same object. Returns `true` if they are the same, otherwise `false`.

3.7.3 Projection

The conditional statement and the while loop are already present in the target language (as it includes instructions from PGLEcw), so it is easy to project them.

The test expressions are evaluated and determine whether the program enters the body or not. In case of the while statement, the test expression needs to be evaluated again after each iteration of the loop.

Because the `nil` object is also considered false. A test checks if the result of the expression is `nil` and sets it to `false` if it is.

The equality test `==` is present in the target language (as it includes instructions from MPP) and checks whether two foci point to the same atom. The equality test instruction in RC1 is implemented by defining a method `==` in the `Object` class. All objects can be tested for equality this way because every

[12] Conditional statements are expressions in Ruby, but I restrict them to statements in RC1 for simplicity.

object inherits methods from the `Object` class. By making the equality test a method, other classes can override the method to define equality on instances of those classes in a new way.

Conditional and while statements preserve context. This means that instructions inside a conditional or while block belong to the same context as the statement itself. This is done by using a variable X in the projection function ψ_X.

- This macro sets `result` to `false` when it equals `nil`.

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

- The `while` statement needs to evaluate the test expression after every iteration.

  $\psi_X($ `while` $expr; u_1; \ldots; u_k;$ `end` $) =$
  $\psi(expr);$ $\varphi_{nil \to false};$
  `-result == false{*;` $\psi_X(u_1); \ldots; \psi_X(u_k);$ $\psi(expr);$ $\varphi_{nil \to false};$ `*}`

- $\psi_X($ `if` $expr; u_1; \ldots; u_k;$ `end` $) =$
  $\psi(expr);$ $\varphi_{nil \to false};$
  `-result == false{;` $\psi_X(u_1); \ldots; \psi_X(u_k);$ `}`

- $\psi_X($ `if` $expr; u_1; \ldots; u_k;$ `else;` $u_{k+1}; \ldots; u_l;$ `end` $) =$
  $\psi(expr);$ $\varphi_{nil \to false};$
  `-result == false{;` $\psi_X(u_1); \ldots; \psi_X(u_k);$`}{;` $\psi_X(u_{k+1}); \ldots; \psi_X(u_l);$ `}`

- This instruction translates the equality test to a method call.

  $\psi($*exp0* `==` *exp1*$) =$
  $\psi($*exp0*`.==(`*exp1*$))$

3.8 Projection of Programs in RC1

In the previous sections, a projection function has been given for each of the different types of statements and expressions in RC1. Here the projection of an entire RC1 program to the intermediate projection language (IPL) is given.

The projection function `rc2ipl` is defined below.

Before any statements in the program are projected, five classes need to be created: the `Object`, `Class`, `TrueClass`, `FalseClass` and `NilClass` classes.

The `Object` class is the superclass of all other classes.

Every class is an instance of the class `Class` (including itself).

The last three classes are made for the keyword objects `true`, `false` and `nil`.

Methods for the `Object` and `Class` classes need to be defined. It would be easier to define these in RC1, than writing them out in IPL. But these methods access fields which are not accessible as instance variables, such as `class` and `super`. And the equality test method in `Object` must have access to the equality test in MPP. So these methods are written directly in IPL.

- `rc2ipl`$(u_1;...;u_k)$ =
  $\varphi_{init-classes}$;
  $\varphi_{init-methods}$;
  $\varphi_{init-stackframe}$;
  $\psi_{main}(u_1);...;\psi_{main}(u_k)$

- This macro creates predefined classes and constants.

  $\varphi_{init-classes}$ =
```
Object = new;
Class = new; Class.+ super; Class.super = Object;
TrueClass = new; TrueClass.+ super; TrueClass.super = Object;
FalseClass = new; FalseClass.+ super; FalseClass.super = Object;
NilClass = new; NilClass.+ super; NilClass.super = Object;

Object.+ class; Object.class = Class;
Class.+ class; Class.class = Class;
TrueClass.+ class; TrueClass.class = Class;
FalseClass.+ class; FalseClass.class = Class;
NilClass.+ class; NilClass.class = Class;

true = new; true.+ class; true.class = TrueClass;
false = new; false.+ class; false.class = FalseClass;
nil = new; nil.+ class; nil.class = NilClass;
main = new; main.+ class; main.class = Object;
```

- This macro begins a definition of an instance method. It creates a method with identifier m for class C. It sets up the jump labels. After this macro the body of the method should be given, and ended with the φ_{im-end} macro. Here it is possible to use string labels for the jump labels because the class is known in advance.

  $\varphi_{im-begin}(C,m)$ =
```
-C/im {;C.+ im; aux = new; C.im = aux;};
methods = C.im;
method = new;
methods.+ m;
methods.m = method;
method.+ label:string;
method.label = "C:m";
##L"C:m:skip";
L"C:m";
```

- This macro should close the definition of an instance method that begins with $\varphi_{im-begin}$.

  $\varphi_{im-end}(C,m)$ =
```
R;
```

L"C:m:skip"

- The bodies of these methods are written in IPL. Hence, the `result` focus should be used to point to the return value of the method. Method arguments are given in foci `arg1` to `arg`n. They are not stored as local variables on the stackframe. The same holds for `self`.

  the `new` method can have any number of arguments. After a new object has been created, the `initialize` method is searched in the object's class. The arguments to `new` are used in the `initialize` method.

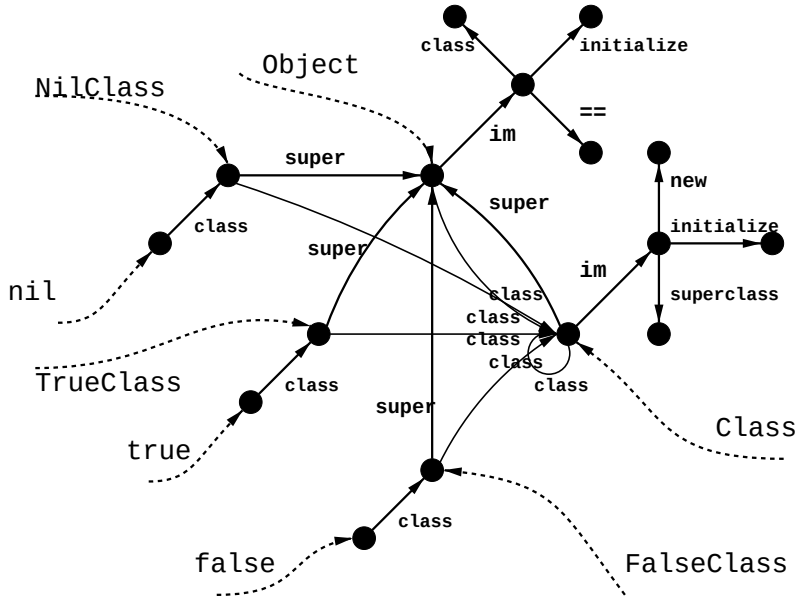
```
φ_init-methods =
φ_im-begin(Object, initialize);
φ_im-end(Object, initialize);

φ_im-begin(Object, class);
  result = self.class;
φ_im-end(Object, class);

φ_im-begin(Object, ==);
  +self == arg1{; result = true ;}{; result = false ;};
φ_im-end(Object, ==);

φ_im-begin(Class, initialize);
  self.+ super; self.super = arg1;
φ_im-end(Class, initialize);

φ_im-begin(Class, superclass);
  +self/super{; result = self.super ;}{; result = nil ;};
φ_im-end(Class, superclass);

φ_im-begin(Class, new);
  x = new; x.+ class; x.class = self;
  φ_search-instance-method(initialize);
  stackframe.+ x; stackframe.x = x;
  self = x;
  label = method.label;
  R#L[label];
  result = stackframe.x;
φ_im-end(Class, new);
```

- The stackframe needs to be created and `self` must be initialized to `main`.

```
φ_init-stackframe =
stackframe = new;
stackframe.+ self; stackframe.self = main;
stackframe.+ lv; aux = new; stackframe.lv = aux
```

The program state after initialization of RC1 is shown in figure 3.6.

Figure 3.6: Initial program state



3.9 Example Program

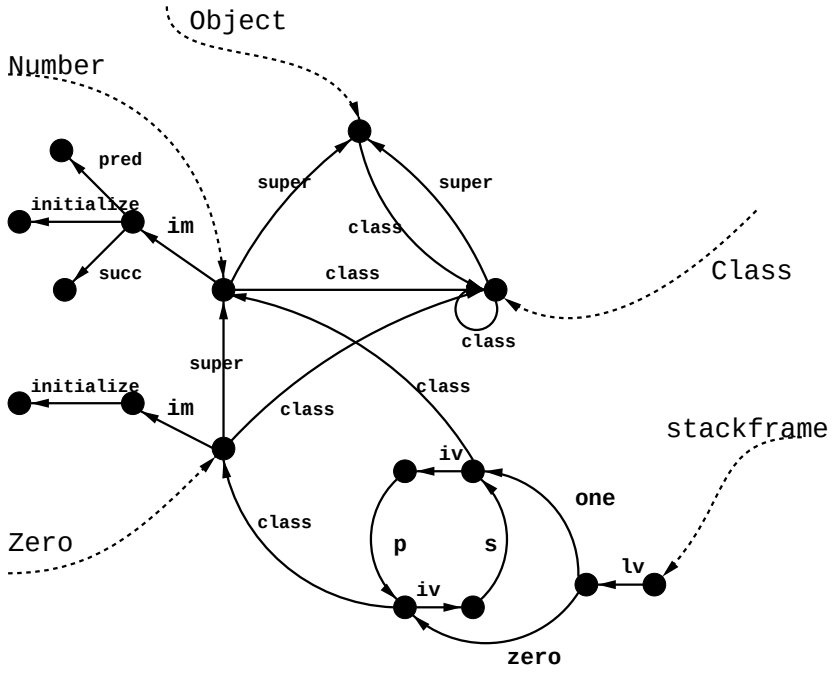
The last example is the class `Number`. Its instances represent all natural numbers starting with zero. The operations defined on them are `succ` and `pred` which return the successor and predecessor. These objects are stored inside the instance variables `@s` and `@p` respectively. Only the *zero* object has no predecessor variable. A special subclass has been made for *zero* which overrides the `initialize` method of `Number`.

When the successor of a number is accessed but not available, the successor object is created and returned. This way the collection of numbers grows during the execution of the program.

When the predecessor is accessed but not available, the method returns `nil`.

```
class Number;
  def initialize(p);
    @p = p;
  end;

  def succ();
    if defined? @s;
      return @s;
    else;
      @s = Number.new(self);
      return @s;
```

Figure 3.7: Program state after execution of the example program in RC1




```
    end;
  end;

  def pred();
    return @p;
  end;
end;

class Zero < Number;
  def initialize();
  end;
end;

zero = Zero.new();
one = zero.succ();
```

Figure 3.7 shows part of the program state after execution of the program above.

3.10 Summary

In regard to the amount of features, RC1 is a very small subset of the Ruby language. Ruby is designed to make programming easy, and it offers a vast amount of language features to remove the burden of writing repetitive code from the programmer. Although RC1 is very small, it does consist of all the important OO features: objects, classes and inheritance. Albeit with limitations.

In the following sections, RC1 will be extended with new instructions in subsets RC2, RC3 and RC4.

The only type of methods in RC1 are instance methods, which are methods defined on instances of a particular class. Other types of methods are class methods and singleton methods which are implemented in RC2.

The two types of variables in RC1 are local variables and instance variables. Ruby has three other types: global variables, user-defined constants and class variables. RC3 implements class variables.

What is probably most obviously missing to programmers are basic types such as *integers* and *strings*. Although it is possible to compute everything with objects (see the `Number` class example) it is very impractical. RC4 contains integers as a basic type.

Chapter 4

Ruby Core Two (RC2)

RC2 extends RC1 with two types of methods.

4.1 Class Methods and Singleton Methods

The only type of methods that RC1 has, is the instance method.

Instance method An instance method is defined on all instances of a class.

Singleton method A singleton method is defined on a single instance object.

Class method A class method is defined on the class itself.

A *singleton method* extends one individual instance object with a method. No other objects belonging to that object's class can call that method. This is useful when only one instance needs to have a particular method, or needs to override a particular method. In other languages, such as Java, this is accomplished with anonymous classes. An anonymous class is a nameless class that is created for one instance only. Singleton methods provide the same functionality but they do not change the class of the instance object.

A *class method* is not defined on instances of the class but the class itself. Class methods can be inherited from superclasses like instance methods can. Class methods are called *static* methods in Java and `C++`.

4.1.1 Comparison

Class methods and singleton methods are more similar than they seem: A singleton method is defined on a single object and a class method is defined on a single *class* object. (Remember that all classes in Ruby are instances of the `Class` class.) In Ruby, a class method *is* a singleton method defined on a class object.

Because classes are instance objects in Ruby, both method types can be implemented as one.

4.1.2 Syntax

In the definition of a singleton method the object on which the method is defined must be supplied. `def x.m()` begins a method definition for a method `m` on object `x`. Class methods are defined the same way but have the class name in front of the method: `def A.m()` defines a class method `m` on class `A`.

Because singleton methods can be defined on every object – a class object in case of a class method – the definition can receive any expression. All expressions return an object, so the definition is guaranteed to create a singleton method although instructions such as `def n(y).m()` look unusual. This instruction means: define method `m` on the object returned by the expression `n(y)`

The definition of a singleton method can occur in a class definition block or in the main section of the program. The class object is supplied in the definition of a class method, therefore class methods can also be defined outside the class definition block. Although it is usually good practice to put all class methods together inside the class definition.

- The singleton (class) method definition:

  `def` *expr*`.`*m*`(p1,...,pn);`u_1`;...;`u_k`;end`

  expr, when evaluated, supplies the object on which the method *m* will be defined.

  (When *expr* evaluates to an instance of a class, the new method will be a class method.)

4.1.3 Example

We will revisit the `Number` class example from RC1. A dedicated subclass `Zero` was used to implement the number *zero*. With class methods available in RC2, this can be done more elegantly. The `Number` class will have a class method `zero` that returns the zero object. The zero object can be stored in an instance variable of of the `Number` class. Instance variables for classes were available in RC1, but with no way to access them in methods they were rather useless. Class methods can access instance variables which are either created inside the class definition block or inside a class method.

Creating the zero object *before* defining the `initialize` method in class `Number` causes the default initializer to be applied. The `initialize` method in class `Object` does not create any instance variables. This way the zero object can be created without a predecessor object available.

```
class Number;
  @zero = Number.new();

  def Number.zero();
    return @zero;
  end;

  def initialize(p);
```

```
    @p = p;
  end;

  def succ();
    if defined? @s;
      return @s;
    else;
      @s = Number.new(self);
      return @s;
    end;
  end;

  def pred();
    return @p;
  end;
end;

two = Number.zero().succ().succ();
```

Note that `zero` is a class method because it is prepended with its class name in the method definition.

4.1.4 Projection

Singleton methods are added to a field `sm` of the object. This way class methods can be distinguished from instance methods which are added to a field `im`. [1] The projection of the singleton method definition does not differ much from the instance method definition. Only the location where the method object is stored is different.

- i is the position of the method definition instruction in the program.

  $\psi_{class,main}$(`def` $expr$`.`m`(p1,...,pn);`u_1`;...;`u_k`;end`) =
  $\psi(expr)$`; x = result;`
  `-x/sm {;x.+ sm; aux = new; x.sm = aux;};`
  `methods = x.sm;`
  `method = new;`
  `methods.+ `m`;`
  `methods.`m` = method;`
  `method.+ label:int;`
  `method.label = `$(2i)$`;`
  `##L`$(2i+1)$`;`

[1] The current implementation of Ruby uses the *metaclass* concept to implement singleton methods. A metaclass is used to store the singleton methods of an object. The advantage of this scheme is that singleton method calls and instance method calls can be treated uniformly.

Some textbooks explain the metaclass concept as part of the Ruby language. But unlike Smalltalk, there is no such thing as a metaclass in Ruby (they can never be referenced). By using different branches for instance and singleton methods, the projection can be done without metaclasses entirely.

L($2i$);
arguments = new; arguments.+ *p1*;...;arguments.+ *pn*;
arguments.*p1* = arg1;...;arguments.*pn* = arg*n*;
stackframe.+ lv; stackframe.lv = arguments;
stackframe.+ self = self; stackframe.self = self;
$\psi_{method}(u_1);\ldots;\psi_{method}(u_k)$;
R;

L($2i+1$)

4.2 Method Calls

With the addition of singleton methods, the behavior of the method call instruction must be adapted as well. The syntax of the instruction does not change, but the way in which it searches for a matching method does change.

In RC1 the only type of method is the instance method. When a method is called, the class of the calling object is checked for a method with that name, and successively all the parent classes as well. RC2 adds singleton methods, these override instance methods, so first the program searches for a singleton method and then it searches for an instance method.

Singleton methods can also be inherited from parent classes. Of course, this only applies to class methods.

4.2.1 Example

This example is about a class `TimeStamp`. The class supplies a method `create` which makes a new `TimeStamp` object. `create` should be used instead of the instance method `new` from `Class`. The class keeps track of the last created timestamp with an instance variable `@last` of the `TimeStamp` class.

Three instance methods are defined on `TimeStamp` objects: `after`, `before` and `equals`, they compare different timestamps to each other.

`TimeStamp` objects have an instance variable `@time` that holds a `Number` object to compare their position to other timestamps. To compare different numbers, the `Number` class must be extended with a method `gt` that checks if a number is greater than another number:

```
class Number;
  def gt(x);
    if pred();
      if pred() == x;
        return true;
      else;
        return pred().gt(x);
      end;
    else;
      return false;
```

```
    end;
  end;
end;
```

The `Number` class is extended with one method `gt` without changing all the previously defined methods on `Number`.

```
class TimeStamp;

  def initialize(time);
    @time = time;
  end;

  def TimeStamp.create();
    if defined? @last;
      newtime = @last.time().succ();
      @last = TimeStamp.new(newtime);
    else;
      @last = TimeStamp.new(Number.zero());
    end;
    return @last;
  end;

  def time();
    @time;
  end;

  def after(t);
    time().gt(t.time());
  end;

  def equals(t);
    time() == t.time();
  end;

  def before(t);
    if equals(t);
      return false;
    end;
    if after(t);
      return false;
    end;
    return true;
  end;

end;

t0 = TimeStamp.create();
t1 = TimeStamp.create();
t0.before(t1) == true;
```

Note the difference in scope of the `@last` and `@time` variable. `@last` is an instance variable of the `TimeStamp` class, while `@time` is an instance variable of instances of `TimeStamp`.

When `create` is called on the `TimeStamp` class, first the program searches for a singleton method in the `TimeStamp` class (class method) with that name. It is defined there so it calls the method. Otherwise the program would first search all the superclasses of `TimeStamp` for a class method `create`, and then search the class of `TimeStamp` (`Class`) and its superclasses for a `create` method.

The same happens when calling `before` on object `t0`. First the object `t0` is inspected for a singleton method `before`, when it finds none, the class of `t0` (`TimeStamp`) is searched for a method called `before`.

4.2.2 Projection

The projection of the method call in RC1 is reused here. Now it pays off to have defined those macros previously. Such as $\varphi_{search-supers}(x, s, i)$, which searches the superclasses of x for a method `i` in field s. It can be used to search for both singleton methods as well as instance methods.

- The method call instruction is redefined here. $\psi_{search-method}$ searches for singleton methods as well.

  $\psi(exp0.m(exp1, ..., expn)) =$
  $\psi(exp0)$`;x = result;`
  $\psi(exp1)$`;arg1 = result;...;`$\psi(expn)$`;arg`n` = result;`
  $\varphi_{search-method}(m)$`;`
  `self = x;`
  `result = nil;`
  `label = method.label;`
  `R##L[label];`

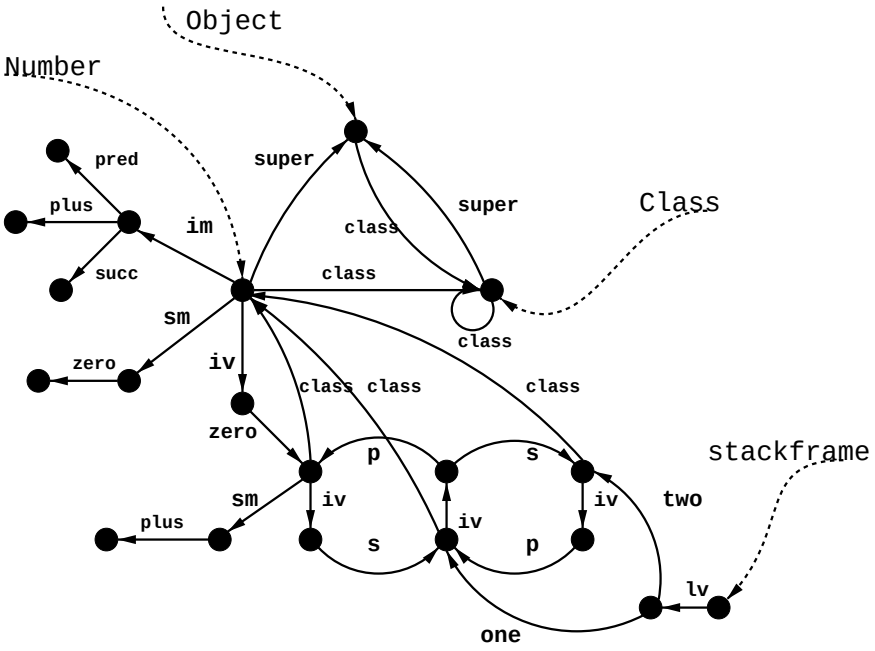
- First the singleton methods are searched in the class hierarchy of an object focused by `x` by inspecting the `sm` branches, then the instance methods are searched by inspecting the `im` branches of the class of `x`.

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

4.3 Example

As a last example, here is a recursive definition of `plus` that extends the `Number` class from the previous examples. There are two `plus` methods defined. A singleton method defined on the zero object, and an instance method defined on all instances of class `Number`. Because singleton methods are executed before instance methods, the program will pick the singleton method automatically over the instance method when the recursive method reaches the zero object.

Figure 4.1: Program state after execution of the example program in RC2




```
class Number;
  def @zero.plus(rhs);
    return rhs;
  end;

  def plus(rhs);
    return pred().plus(rhs.next());
  end;
end;

one = Number.zero().succ();
two = one.succ();
one.plus(one) ==  two;
```

The program state after execution is depicted partly in figure 4.1.

Chapter 5

Ruby Core Three (RC3)

RC1 has two types of variables: local variables and instance variables. RC3 adds another type: class variables.

5.1 Class Variables

Because classes are instance objects in Ruby, they can have instance variables. These instance variables are only accessible inside the class body or a class method. When an instance variable of a class must be updated during an instance method call, a class method must be called to access it.

Ruby has a dedicated class variable which can be manipulated inside both class methods and instance methods. It has different syntax (`@@`x) to distinguish it from instance variables. In addition, class variables are inherited by extending classes. This further facilitates access because it is not necessary to write access methods in extending classes.

5.1.1 Syntax

- Class variable expression:

  `@@`x ($expr$)

- Class variable assignment:

  `@@`x `=` $expr$

- Tests whether a class variable exists:

  `defined? @@`x ($expr$)

5.1.2 Comparison

The differences between an instance variable (`@x`) of a class `A` and a class variable (`@@x`) of `A` are the following:

- `@@x` is also accessbile in instance methods of `A`, whereas `@x` can only be accessed in class methods of `A`.
- `@@x` is accessible in subclasses of `A`, whereas `@x` is not.

Class variables make programming in Ruby easier, and programs easier to read. But when one diligently writes access methods for all instance variables of a class, the same functionality as class variables can be achieved.

5.1.3 Example

In this example the `Number` class will be extended to comprise both positive and negative numbers. The class keeps track of the minimum and maximum of the created numbers with the use of class variables. When a successor number does not exist, a new instance of `Positive` is created, and the `@@max` class variable in `Number` is updated. In a similar manner, a `Negative` number is created when the predecessor is not available, and the `@@min` variable is updated.

```
class Number;
  @@zero = Number.new();
  @@min = @@zero;
  @@max = @@zero;

  def Number.zero();
    return @@zero;
  end;

  def succ();
    if defined? @s;
      return @s;
    else;
      @s = Positive.new(self);
      return @s;
    end;
  end;

  def pred();
    if defined? @p;
      return @p;
    else;
      @p = Negative.new(self);
      return @p
    end;
  end;
end;

class Positive < Number;
  def initialize(p);
    @p = p;
    @@max = self;
```

```
  end;
end

class Negative < Number;
  def initialize(s)
    @s = s;
    @@min = self;
  end
end
```

Although positive and negative numbers have a separate class, they can access the class variables `@@max` and `@@min` because these are inherited.

5.1.4 Projection

Class variables are stored in a seperate branch of the class object: the `cv` field. It points to an object that contains all the class variables.

Because a class variable can be inherited from superclasses, the $\psi_{search-supers}$ macro is used to search the superclasses for class variables.

When a class variable is accessed but not found, the program must terminate.

- The class variable expression uses the $\psi_{search-supers}$ macro to search for a class variable x in `cl` and superclasses, if it is found the value is stored in focus `res` and `found` is set to `true`.

  $\psi($`@@`$x) =$
  $\psi_{find-class}$`;`
  $\psi_{search-supers}$`(cl,cv,`x`);`
  `+found == false;!;result = res`

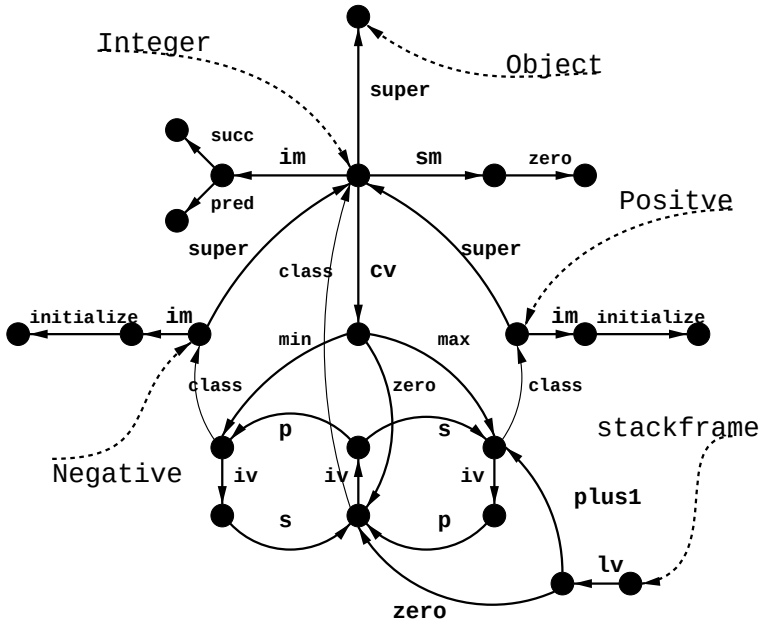
- This macro finds the class belonging to `self`. If `self` is an instance of `Class`, `cl` is focused on `self`, otherwise it is focused on the class of `self`.

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

- The assignment instruction searches for an existing class variable x. If it finds one, the variable is updated on branch `br` – that is, inside the superclass where it found the class variable. Otherwise a new class variable is added to the current class `cl`.

  $\psi($`@@`x `=` $expr) =$
  $\psi(expr)$`; p = result;`
  $\varphi_{find-class}$`;`
  $\psi_{search-supers}$`(cl,cv,`x`);`
  `+found == false{;cl.+ cv; classvars = new; cl.cv = classvars;`
  `  }{;classvars = br;};`
  `classvars.+ `x`; classvars.`x `= p`

Figure 5.1: Program state after execution of the example program in RC3



- ψ(`defined? @@`x) =
  $\varphi_{find-class}$`;`
  $\psi_{search-supers}$(`cl,cv,`x)`;`
  `result = found`

5.2 Example

Using the `Number` class defined previously, the following program creates three numbers.

```
zero = Number.zero();
plus1 = zero.pred().succ().succ();
```

The resulting program state is depicted partly in figure 5.1. The `Class` class is omitted to save space.

Chapter 6

Ruby Core Four (RC4)

Up to RC3, the only type of data that objects contain are objects themselves. In previous examples it has been shown that numbers can be represented by a chain of objects. This representation of numbers by unique objects is very inefficient. It would be better to compute with numbers in the underlying system – i.e. numbers represented by a sequence of bits. In the same way, characters and strings are represented more efficiently by bytes than by unique objects.

Ruby offers a number of basic types such as integers. These types are classes like any other in Ruby, but their instances differ from other objects because they contain data that is not accessible in instance variables. [1] In addition, instances of basic types can be created by a literal expression instead of a method call `new`.

For example, the object that represents the number thirteen is created by the expression `13`. All operations on it are instance methods defined in the number's class. The code `13 + 7` calls method `+` on number `13` with argument `7`.

In contrast to other OO languages [2] , all basic types are objects and operations on them are method calls. This makes Ruby a *pure* OO language.

RC4 extends RC3 with one basic type: *integers*.

6.1 Integers

In Ruby integers are divided in `Fixnum` and `Bignum` instances. `Fixnum` numbers have a fixed size, and `Bignum` numbers can become arbitrarily large. Conversion

[1] To be more precise, the objects have a characteristic – such as being number thirteen – which can not be retrieved from instance variables.

[2] *Java* has *primitive types* to represent numbers and characters. These types are not classes, and numbers and characters are not objects. All the functions operating on them are defined in the language, they are not methods belonging to a class. The problem with these primitive types is that uniform access to data is impossible when data is divided between objects and primitives.

between these two types happens automatically. Both classes are subclasses of the `Integer` class.

To simplify matters, all integers are instances of `Fixnum` in RC4.

Integers are created by reading in their numerical value. Integers are immutable, their values can not change during the execution of the program. See the following code:

```
x = 13;
y = x;
x = x + 1;
```

One might guess that the resulting value of variable `x` is 14 and of `y` 13. This is indeed the case. Integers are immutable, so the `+` method does not modify the value of the calling object `x`. Instead it returns a new object with value 14 which is assigned to variable `x`. The variable `y` still points to the number 13.

Another important operation on numbers is the test for equality. Lets look at an example:

```
x = 13;
y = 13;
x == y;
```

According to the rules of integer arithmetic, `x` and `y` are equal, and the test should result in `true`. This is indeed the case. But it is not obvious when regarding `x` and `y` as variables holding different objects which happen to have the same value 13. The equality test in `Object` does not care about the contents of the objects. That is why the equality test has been overridden in `Fixnum`: it tests whether the values of its arguments are the same.

In Ruby there are many methods defined on `Fixnum`, in RC4 only the `+` and `==` methods are implemented.

6.1.1 Syntax

The syntax for RC4 is extended with numbers which are translated into `Fixnum` objects.

- An *int* consists of a sequence of digits, optionally preceded by a `-` sign.

  int (*expr*)

6.1.2 Projection

`Fixnum` objects are implemented by having integer value fields which are hidden from the programmer. The only way to access the value fields is by calling pre-defined methods on the objects. [3]

[3] For what are apparently performance reasons, the current Ruby interpreter stores instances of Fixnum as integers, not as objects. This does not change the semantics, it is merely an optimization of the interpreter.

- The `Integer` and `Fixnum` classes must be defined before the user program is executed. The projection function of the program is changed accordingly.

  `rc2ipl`$(u_1; ...; u_k) =$
  $\varphi_{init-classes}$;
  $\varphi_{init-methods}$;
  $\varphi_{init-integer}$;
  $\varphi_{init-stackframe}$;
  $\psi_{main}(u_1); ...; \psi_{main}(u_k)$

- The `Fixnum` class defines two instance methods `+` and `==`.

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

  $\varphi_{im-begin}$(`Fixnum, ==`);
  ```
    +self.value == arg1.value {;result = true;}{;result = false;};
  ```
  φ_{im-end}(`Fixnum, ==`);

  $\varphi_{im-begin}$(`Fixnum, +`);
  ```
    result = new; result.+ class; result.class = Fixnum
    result.+ value:int;
    integer-add(self.value, arg1.value, result.value);
  ```
  [4]
  φ_{im-end}(`Fixnum, +`);

- *int* represents an integer; a new object is created containing the value *int*.

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

- Like the equality test, the addition operation has an infix notation which is translated to a method call.

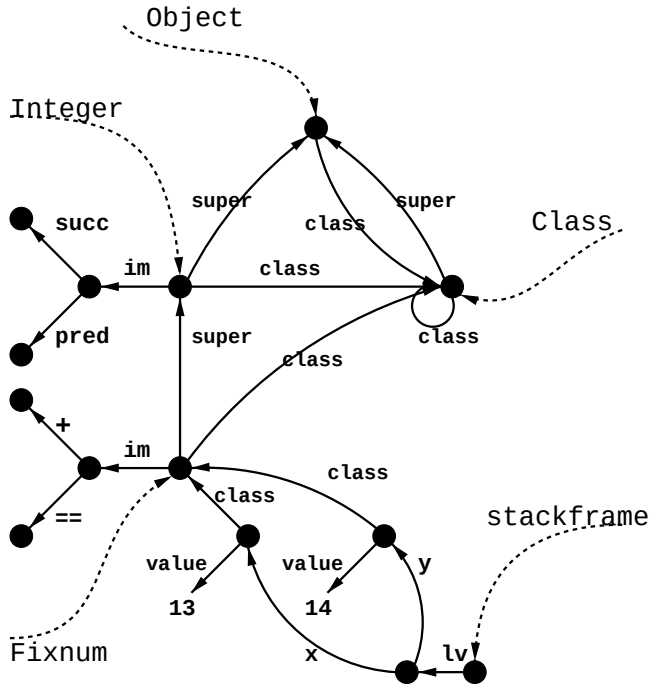
  $\psi(exp0$ `+` $exp1) =$
  $\psi(exp0$`.+(`$exp1$`)`$)$

6.2 Example

Classes can be refined after they have been defined, this is also the case with standard classes such as `Integer`. In this example the `Integer` class gains two familiar methods: `succ` and `pred`.

[4] *integer-add* is a request to the underlying system to perform integer addition on the first two arguments and store the result in the third argument.

Figure 6.1: Program state after execution of the example program in RC4




```
class Integer;
  def succ;
    return self + 1;
  end;
  def pred;
    return self + -1;
  end;
end;

x = 13;
y = x.succ();
```

The resulting program state after execution of the program is depicted in figure 6.1.

Chapter 7

Conclusion

A projection of subsets of Ruby to program algebra has been presented, which focuses on the OO constructs of the language.

An intermediate projection language (IPL) has been defined as the target language for the projection. IPL is an extension of PGLEcw and MPPV, and adds control instructions to facilitate the projection of method calls.

Four subsets have been presented starting with RC1. RC2, RC3 and RC4 extended the subset incrementally with features. RC2 changed the projection of RC1 by adapting the method call instruction to singleton methods. RC3 and RC4 added features but did not change any instructions from RC1 or RC2, so these are conservative extensions.

Below is an overview of the different subsets:

- RC1 has simple control instructions, local variables, classes, instance methods and instance variables.
- RC2 adds singleton methods and class methods.
- RC3 adds class variables.
- RC4 adds integers, as a basic type.

The presented projection is by no means the only possible one. Some arbitrariness is unavoidable in the details of the projection. There are however some mechanisms which are fundamental to the projection.

To allow polymorphism, information about the class relations and their methods must be available during execution of the program. Therefore a tree-like data structure has been constructed to represent the class hierarchy.

To allow nested method calls, a datastructure is necessary to keep track of the return positions in the program, and the values of local variables in different invocations of the same method. The *stackframe* served this purpose.

These last two mechanisms are not unique to Ruby but appear in every OO language, and are therefore interesting in their own right.

Bibliography

[1] J.A. Bergstra and M.E. Loots. Program algebra for sequential code. *Journal of Logic and Algebraic Programming*, 51(2):125-156, 2002.

[2] J.A. Bergstra and I. Bethke. Molecular dynamics. *Journal of Logic and Algebraic Programming*, 51(2):193-214, 2002.

[3] P. Wegner, *Dimensions of Object-Based Language Design*, ACM SIGPLAN Notices, Proceedings OOPSLA '87, vol. 22, no. 12, pp. 168-182, Dec 1987.

[4] David Thomas and Andrew Hunt. *Programming Ruby: The Pragmatic Programmer's Guide*, ISBN: 0-201-71089-7, Addison Wesley Professional, 2001.