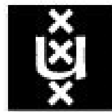
UNIVERSITEIT VAN AMSTERDAM (UvA)

Faculty of Science



A Projection and Its Implementation of a Ruby Subset to Program Algebra

Master Thesis

Ran Huo
Qiong He
July 2005

Title: A Projection and Its Implementation of A Ruby Subset to Program Algebra

Authors: Qiong He
q.he@student.uva.nl

Ran Huo
r.huo@student.uva.nl

Program: Grid Computing

Faculty: Faculty of Science, University of Amsterdam

Supervisors: mw. dr. I Bethke
Kruislaan 403, 1098 SJ, Amsterdam

dr. A Ponse
Kruislaan 403, 1098 SJ, Amsterdam

Date: July 2005

Abstract

Ruby is a powerful and open-source object-oriented (OO) programming language. It has a very clear and lightweight syntax, making it attractive to work with. The object of this thesis is to define and implement a projection of a Ruby subset to program algebra (PGA). A PGA program is a sequence of primitive instructions. In PGA, there are various ways to model program execution behavior, such as the PGA Toolset or the mathematical modeling of PGA. The projection focuses on Ruby's OO constructs and its data types. We use a Ruby program as the input for our projection and generate its I/O equivalent PGA program as the output. In order to implement the projection, several new primitive and basic instruction sets are defined in PGA. Furthermore, these extensions are implemented in the current PGA Toolset.

Keywords: PGA, MSP, Ruby, Data type

* We sincerely thank dr. A. Ponse and mw. dr. I. Bethke for their dedicative supervision on our graduation work. And we also express our very gratitude to Bob Diertens for his help and suggestions.

Contents

Abstract **iii**
Contents **iv**
Preface **ix**

1 Introduction **1**
1.1 The Ruby programming language 1
1.2 Summary of Geerlings' work 2
1.3 Structure of the thesis 5

2 Program Algebra (PGA) **7**
2.1 PGA 7
2.2 Molecular dynamics 10
2.3 The PGA Toolset 14

3 The Intermediate Projection Language (IPL) **18**
3.1 PGLEcr and PGLEcm 18
3.1.1 PGLEcr 19
3.1.2 PGLEcm 20
3.2 MSP and MSPea 23
3.2.1 MSP 23
3.2.2 MSPea 24
3.3 Extensions 28
3.3.1 PGLEcrs 28
3.3.2 PGLEcmrs 29
3.3.3 MSPeame 29

4 Implementation of the Extensions in the PGA Toolset **31**
4.1 Extending the PGA Toolset with PGLEcrs and PGLEcmrs 31
4.2 Extending the PGA Toolset with MSPeame 36

5 Implementation of Geerlings' Projections in IPL **39**
5.1 Framework 39
5.2 Keyword parsing 40
5.3 Initialization 42

5.4 Instruction parsing 45
5.5 Implementation of projections in IPL 46
5.5.1 Projections implemented in RC1 46
5.5.2 Projections implemented in RC2 53
5.5.3 Projections implemented in RC3 54
5.5.4 Projections implemented in RC4 56
5.6 Easy and difficult aspects of this implementation 57

6 Ruby Core Four Plus (RC4+) 58
6.1 Extending the integers 58
6.1.1 Example 59
6.1.2 Syntax 60
6.1.3 Projection 60
6.2 Projection example of the integers 63

7 Ruby Core Five (RC5) 66
7.1 Booleans 66
7.1.1 Syntax 66
7.1.2 Projection 67
7.2 Projection example of the Booleans 69
7.3 Strings 73
7.3.1 Example 74
7.3.2 Syntax 75
7.3.3 Projection 75
7.4 Projection example of the strings 76

8 Ruby Core Six (RC6) 78
8.1 Arrays 78
8.1.1 Example 79
8.1.2 Syntax 80
8.1.3 Projection 86
8.2 Projection example of the arrays 87
8.3 Hashes 88
8.3.1 Example 89
8.3.2 Syntax 89
8.3.3 Projection 89
8.4 Projection example of the hashes 93

9 Ruby Core Seven (RC7) 96
9.1 Blocks and iterators 96
9.1.1 Block definitions 101
9.1.1.1 Syntax 101
9.1.1.2 Projection 101
9.1.2 Invoking the blocks 102
9.1.2.1 Syntax 102
9.1.2.2 Projection 102
9.1.3 Iterators 104
9.1.3.1 Syntax 104
9.1.3.2 Projection 104
9.1.4 Examples of the blocks 105
9.2 Closures 106
9.2.1 Syntax 107
9.2.2 Projection 107
9.2.3 Example 108

10 Implementation of Our Projections in IPL 109
10.1 Framework 109
10.2 Implementation of the projections in IPL 111
10.2.1 Projections implemented in RC4+ 111
10.2.2 Projections implemented in RC5 112
10.2.3 Projections implemented in RC6 116
10.3 Easy and difficult aspects of this implementation 118

11 Other Interesting Features of Our Project 119
11.1 Compilation versus projection 119
11.2 Some interesting operations for strings 121
11.2.1 String-to-New-Atom 121
11.2.2 String-to-Existing-Atom 123
11.2.3 String-to-Field 125

12 Conclusion 128

References 130

Appendix A 132
Fluids of the Projection Examples' Execution Results

Appendix B 134
Implementation Programs for the Projection of the Ruby Subset to PGA

Appendix C 137
Programs and Modules for Implementing the PGA Toolset Extensions

Preface

This thesis was written as a follow-up of Geerlings' master thesis [9], written in November 2003 in a project done with the Programming Research Group, Faculty of Science, University of Amsterdam (UvA). This thesis is a joint work. The motivation for writing this double thesis is listed below:

First, Ruby is an interesting OO programming language. Its OO features are simple but powerful. The basic OO constructs are analyzed in [9], but only one data type – integer is dealt with. In order to make our work on Ruby more practical than in [9], we consider several other data types as well as the block, a particular strength of Ruby.

Second, we are both interested in the field of PGA. PGA is an algebraic framework that is intended to contribute to a better understanding of sequential programming. In PGA, a program is nothing more than a sequence of primitive instructions. This was a brand new field to us when we began our master program at UvA. It showed us a new aspect of doing computer science research. What's more, we have some successful experience of working together. We were in the same group when doing our profile project, which was also under the supervision of dr. Ponse. Due to above reasons, we decided to do our graduation project in the same domain.

Last but not least, three high-level basic instruction sets of PGA have been defined and implemented in the PGA Toolset, described in [7]. This means some new powerful instructions are available now, such as the instructions for objects and methods. It is feasible to implement the projection of the Ruby subset to PGA with several other extensions described in this thesis.

Although doing the same project, we had a clear distribution of the work. One part of the thesis, which defines the projection functions of a Ruby subset to PGA, is a corporative work and this was done at the first phase of the whole project. This part of our work helped us to get a deep understanding of the Ruby language and the projections defined in [9]. As for the other parts we had our own tracks: Qiong He focused on the PGA extensions and how to implement them in the PGA Toolset while Ran Huo focused on the implementation of the projection of a Ruby subset to PGA. In other words, Ran Huo implemented the projection functions defined by Geerlings and ourselves with the PGA primitive and basic instruction sets provided by Qiong

He’s work. In Chapter 1 we return to this point and explain more precisely how the work was distributed. All the programs for implementing our work on Ruby and for implementing the extensions of the PGA Toolset are available in the appendices.

Amsterdam,
Qiong and Ran,
July 2005

Chapter1

Introduction

1.1 The Ruby programming language

Ruby is a powerful OO, open source programming language. It has adopted various features from other programming languages, such as SmallTalk's pure object orientation, Perl and Python's full regular expression support, convenient shortcuts, and dynamic evaluation. Ruby combines strong theoretical roots with a very clear and lightweight syntax. "It is simple, straight-forward, extensible and portable."[14]

Ruby is dynamic and pure OO. It is dynamic because almost everything in Ruby is done at runtime. There is no static type information and troublesome type declaration to keep in Ruby. In particular, a program distributed over different files is not problematic. Ruby uses simple naming conventions to denote the scope of variables, for examples, `var` is a local variable, `@var` is an instance variable and `$var` is a global variable. Types of variables and expressions are determined at runtime. At the same time, Ruby is designed from the ground up to support the OO paradigm. Almost everything in Ruby is an object and classified to the classes that are objects themselves. Furthermore, all operations in Ruby are method calls. For example, the integer `1` is an object of the class `Fixnum` and `1+1` is a call of the instance method integer addition. Ruby's OO is carefully designed so that it is complete and easy to improve. For example, Ruby's classes are open and new methods can always be added to a class or even to an instance of the class during runtime, thus if needed, an instance of a class can be behaviorally different from this class' other instances.

Ruby is an interpreted language with high performance. No compilation is needed. This makes Ruby portable (platform independent, including Unix, Dos, Windows, OS/2, etc.), and allows for dynamic scoping (classes and methods are built up dynamically during runtime), easy debugging and revising (it is easy to get source code information), and at the same time small program size (it has the convenience and flexibility to choose instruction code).

Blocks and iterators are significant features of Ruby. A block can be attached to a method and the method can call back this block from within its execution. A block is used for loops and

various other purposes. This will be discussed in detail in Chapter 9.

As well as other modern OO languages, Ruby provides support for exception handling to enhance the reliability of the programs. Ruby also has a garbage collection mechanism to recycle all the unused objects automatically. What's more, Ruby has a set of class libraries covering a large range of features, from basic data types to network programming.

All the features of Ruby, mentioned or not mentioned above, make Ruby an interesting and enjoyable language to work with. In this thesis, only a very limited subset of Ruby will be discussed, including Ruby's OO core, some data types and blocks, but this subset can be treated as a programming language on itself.

1.2 Summary of Geerlings' work

Ruben M.Geerlngs is the author of the master thesis "A Projection of the Object Oriented Constructs of Ruby to Program Algebra" [9]. In his thesis a projection of a subset of Ruby to program algebra (PGA) is described. The motivation for such a projection is twofold. Firstly, Program algebra, which uses the molecular programming primitives (MPP)[1] in combination with PGLEc[2], offers a language with objects and methods. But if it is viewed as an Object Oriented (OO) programming language, several essential features must be counted in, such as classes and inheritance. Secondly, Ruby is a good choice as an OO language. It is relatively easy to learn, since its underlying principles are simple.

The programming language Ruby can be defined as a pair (L, φ). L is some collection of textual objects and φ is a program algebra projection. The program algebra projection is a mapping from the set L to PGA. As for the projection language, Geerlings uses an intermediate projection language (IPL), which extends the combination of PGLEc and MPP with some advanced control instructions.

The molecular programming environment Geerlings used in this projection is a model for the structure and meaning of object based programming systems. The memory state of a system is modeled as a fluid consisting of a collection of molecules. Each molecule is a collection of atoms with bindings between them. So a change to the structure will result in a chain of actions,

[1] MPP is a collection of instructions to create and manipulate objects and their connections, suitable for modeling the memory state of OO languages [1].
[2] PGLE programs are sequences of instructions, consisting of control instructions and a collection of basic instructions. PGLEc extends PGLE with conditional compositions.

which help to show the changes of the state in a system clearly. The collection of basic instructions used in [9], is defined in MPPV, which adds values to MPP, for storing integers and Booleans.

From Chapter 3 onwards, Geerlings describes the projection of a small subset of Ruby to PGA. He focuses on the OO constructs. He splits the whole projection into four subsets and explains these in detail in four chapters. It begins with the most basic features of OO languages and the extensions are added gradually in the following three chapters.

The first subset is the most basic one: RC1 (Ruby Core One). RC1 introduces all the basic constructs of OO languages: classes, methods and objects. In RC1 Geerlings only considers one type of methods: the instance method. The overall idea of RC1 is that classes are created dynamically during the execution of a program. They are instances of the class `Class`. Traditionally, classes define methods and variables. Often the instance variables are created when they are assigned a value for the first time inside a method. Based on these concepts, Geerlings writes three basic projections: class definition, method definition and method call. One important mechanism used in the projection is the *stackframe*, which is a data structure created by the instructions in MPPV[3]. It has two important operations: *stackframe-up*, which creates a new layer to store data on and *stackframe-down*, which removes the top layer.

In the class definition part, a class consists of a name and a collection of instance methods, which are stored in the field named `im`. An essential property of classes is inheritance. It is implemented in the projection by using the subclass and superclass focuses. In the method definition part, a method consists of a name and a sequence of instructions, which are executed as a unit. The input objects are called *arguments* and the result object is the *return* object. In the method call part, the method with that name is searched in the object's class. When the method is not defined in the object's class, the object's superclasses are searched up the class hierarchy until it reaches the `Object` class, which has no superclass. If the method cannot be found there, the program terminates, otherwise messages are sent to objects to perform some kind of computation with the data belonging to that object.

In the end of this chapter a whole projection of RC1 is given. It contains all the basic elements needed when initializing a projection of a Ruby program. It shows that the projection begins with a superclass named `Object`, and it has four subclasses: `Class`, `TrueClass`,

[3] MPPV is a set of basic instructions that extends MPP with some instructions for values. It will be described in the next chapter.

FalseClass and NilClass. The Object class is the superclass of all other classes. And 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. There are three build-in instance methods initialize, class and equality test ==. And each subclass has its own build-in instance methods as well.

In Chapter 4, RC1 is extended with two new types of methods to the subset RC2: class methods and singleton methods. A singleton method is defined on a single instance object. No other objects belonging to that object's class can call that method. In contrast, a class method is not defined on instances of the class but the class itself. The difference is that a singleton method is defined on a single object and a class method is defined on a single Class object.

In the projection Geerlings uses a sm field to store the singleton methods. And the projection is separated into two parts. One is the singleton method definition and the other is the method call. 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 then if necessary all the superclasses will be checked. RC2 adds singleton methods and these override instance methods, so the program will first search for a singleton method and then for an instance method.

In Chapter 5, the projection is extended to RC3 with another type of variables: class variables. They are instance variables for the classes, because in Ruby classes are instance objects of class Class. They can be manipulated inside both class methods and instance methods. They can only be accessed inside a class body or a class method. In the projection a cv field is used to store the class variables, which is a separate branch of the class object.

Up to RC3, the only type of data that objects contain is object itself. In Chapter 6, Geerlings introduces a new type of data in RC4: integers. In Ruby, numbers are represented as integer objects. In the projection a superclass Integer and a subclass Fixnum are brought in. All the integers are instances of Fixnum. Two instance methods of integers are projected here: add + and equality test ==. Integers are immutable, which means the operating result of two integers will not modify the value of the calling objects, instead it returns a new object with the result value. In addition, instances of basic types can be created by a literal expression instead of a method call.

Geerlings ends the paper with Chapter 6. And his work accomplishes four Ruby Core subsets: RC1 to RC4. RC1 has simple control instructions, local variables, classes, instance

methods and instance variables. RC2 adds singleton methods and class methods. RC3 adds class variables. And in the end, RC4 adds integers.

1.3 Structure of the thesis

This thesis covers two aspects of contents: a projection and its implementation. One part is a projection of a Ruby subset to PGA. It is a follow-up of Geerlings' projection we described above. In Geerlings' work, the projection of the basic OO constructors of Ruby and the data type integer is given. In order to make this projection more interesting and practical, we consider some other data types and their instance methods, as well as blocks, iterators and objectified blocks.

The other part is about the implementation of the projection, including Geerlings' and ours. It works like a "compiler" of Ruby programs. As described above, Ruby is an interpreted language; its interpreter parses and executes the commands in a Ruby program. In this thesis, it is assumed that Ruby programs can be theoretically compiled into I/O equivalent PGA programs. As to PGA, there are some high-level primitive and basic instruction sets defined and implemented in the PGA Toolset. Combined with several new extensions we defined in this thesis, it is possible to implement both the projection functions defined by Geerlings and ourselves.

After the introductory chapter, this thesis continues with a description of PGA, including the primitive instruction sets, basic instruction sets as well as the PGA Toolset (Chapter 2). Then in Chapter 3 and Chapter 4, the projection language will be discussed; two new primitive instruction sets and a new basic instruction set are defined and implemented in the Toolset. These three chapters were written by Qiong He.

In the following chapter, the implementation of Geerlings' projection is described, covering RC1 to RC4. The implementation is based on the projection functions given by Geerlings although some changes are made as to make the implementation realizable and readable. Chapter 5 was Ran Huo's work.

Then the thesis continues with the projection of several extensions of the Ruby subset to PGA, which are RC4+, defining some new instance methods on the integers; RC5, comprising Booleans and strings; RC6, containing arrays and hashes and RC7, which contains blocks, closures and continuations. These 4 chapters (Chapter 6 to Chapter 9) were combined work of both authors.

The next chapter, Chapter 10 is about the implementation of the extensions that are made in the previous chapters. The style of implementation in this section is more or less similar to that of Chapter 5, which focuses on Geerlings' projection. In this thesis, the implementation of RC7 is not discussed although the projection functions are given because it is not finished yet. Ran Huo wrote this Chapter.

There are some interesting features in our project, such as the compilation versus projection question and some usage of the new primitive and basic instructions of PGA. We describe these in Chapter 11.

In the end, we come to our conclusions in Chapter 12.

The programs for implementing our work on Ruby and for implementing the extensions of the PGA Toolset are available in Appendix B and C. The complete fluids of the execution results of all the projection examples in this thesis are available in Appendix A.

Chapter 2

Program Algebra (PGA)

In this chapter, we discuss some details of PGA, including its syntax and a small hierarchy of program notations; molecular dynamics, containing several sets of basic instructions; as well as the PGA Toolset, a so-called "running environment" for the PGA languages [8].

2.1 PGA

PGA is an algebraic framework that is intended to contribute to a better understanding of sequential programming [2]. The syntax of program expressions in PGA is generated from five kinds of constants, which are called primitive instructions, and two composition mechanisms. The primitive instructions have a set of basic instructions as parameters. After execution of a basic instruction, a Boolean value is returned.
The primitive instructions are:

- *Basic instruction* `a`
  After having performed a basic instruction `a`, a program continues with its next instruction. If the following instruction does not exist, inaction occurs. Inaction is a lack of activity without proper termination. When executing a basic instruction, the state may be modified by the instruction and a Boolean value is returned. The basic instruction does not make use of the returned Boolean value.
- *Termination instruction* `!`
  Termination of the program. It will neither lead to any further effects on the state nor return any value.
- *Positive test instruction* `+a`
  If `+a` is performed by a program, a Boolean value is produced as the result of executing `a` and returned to the program. In case the basic instruction `a` returns `true`, execution continues with the next instruction. If there is no remaining instruction after the positive test, inaction occurs. In case the basic instruction `a` returns `false`, the next instruction is skipped and execution proceeds with the instruction following the skipped one. In this case, there must be at least two instructions following `+a`, otherwise inaction occurs.

- *Negative test instruction* `-a`
  If `-a` is performed by a program, a Boolean value is produced as the result of executing `a` and returned to the program. In case the basic instruction `a` returns `false`, execution continues with the next instruction. If there is no remaining instruction after the negative test, inaction occurs. In case the basic instruction `a` returns `true`, the next instruction is skipped and execution proceeds with the instruction following the skipped one. In this case, there must be at least two instructions following `-a`, otherwise inaction occurs.
- *Forward jump instruction* #*k*
  For any natural number *k*, this instruction denotes a jump of length *k*. If *k*=0, it jumps to the instruction itself and inaction occurs. In the case *k*=1, the instruction skips itself and the subsequent instruction will be executed. If there is no further instruction, inaction occurs. If *k*>1, this instruction skips itself and the next *k*-1 instructions. If there is not that many instructions left in the remaining part of the program, inaction occurs.

There are two compositions operations in PGA:

- Concatenation of X and Y, written X;Y
- Repetition of X, written X^{ω}

PGLA is a program notation that is specifically designed for representing PGA expressions in a linear fashion. An additional primitive instruction is introduced in PGLA: the repeat instruction \\#*n*, for any natural number *n*>0. This additional instruction takes the place of repetition in PGA and repetition is not present in PGLA. A program text ending with this instruction will repeat its last *n* instructions, excluding the repeat instruction itself. The instructions after a repetition instruction are irrelevant and can be deleted. In the case that the program does not have enough instructions, it will be padded with an initial sequence of instructions #0 as to make sufficient instructions. For example, `a;#2;\\#3` is a very short PGLA program and it behaves the same as the PGA program `a;#2;(#0;a;#2)`$^{\omega}$.

Other program notations are designed on top of PGLA and can be mapped to PGLA. A mapping from a higher-level language to a lower one is called a projection. In the direction of projections, the program notations become less and less flexible. A projection can be seen as a theoretical compiler optimized for human understanding. A mapping in the other direction, from a lower-level language to a higher one is called an embedding. An embedding explains the meaning of a program in terms of a more flexible program notation

The hierarchy of the program notations described in relevant publications is shown in Figure 2.1. The real lines arrows indicate available projections and the broken lines arrows indicate the embeddings.

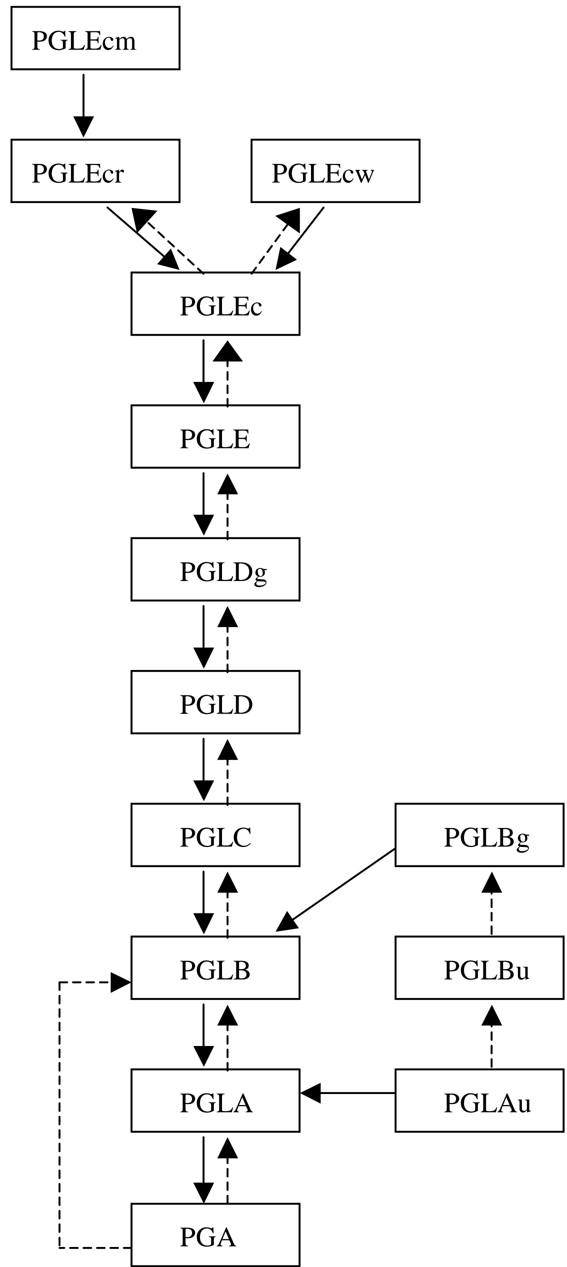


Figure 2.1 Hierarchy of the program notations

2.2 Molecular dynamics

In [5], molecular dynamics is presented as a model for the structure and meaning of object based programming systems. In molecular dynamics the memory state of a system is modeled as a fluid consisting of a collection of molecules. A molecule consists of a collection of atoms, which may have bindings between them and all the atoms are reachable from one of them, the root, by sequences of links. A link exists if the former atom has a so-called *field* containing the latter. Fields can be added to or withdrawn from an atom and the contents of fields can be modified by means of actions causing a state change. Selected atoms can be brought into focus so as to make particular behavioral observations. It can be thought of that, there are a number of proto-atoms available in the memory, which is a measure of the available memory space. A proto-atom is turned into an atom by creating an atom and an atom can also be turned into a proto-atom by garbage collection. In the initial state of a fluid, there is nothing more than proto-atoms. Then the actions in a computation start with creating atoms and end with garbage collections.

The basic instruction set is called MPP (molecular programming primitives). Four mutations, two assignments and two tests instructions are introduced. Where not mentioned, an instruction returns `true` unless a failure occurs, which returns `false`.

Four mutations:

- *Atom creation* `x=new`
  Creates a new atom and assigns it to `x`.
- *Field introduction* `x.+f`
  Adds a reflexive field named `f` to the atom in focus `x`, if this field exists already, `false` is returned and the instruction is ignored.
- *Field withdrawal* `x.-f`
  Removes the field named `f` from the atom in focus `x`, if this field doesn't exist, `false` is returned and the instruction is ignored.
- *Field mutation* `x.f=y`
  Places the atom in focus `y` in the field `f` of the atom in focus `x`. Returns `false` if the atom in focus `x` doesn't own a field `f` and the instruction is ignored.

Two assignments:

- *Field selection* `x=y.f`
  Brings the atom in field `f` of the atom in focus `y` into the focus `x`. Returns `false` if the atom in focus `y` doesn't have a field `f` and the instruction is ignored.
- *Assignment* `x=y`
  Places the atom in focus `y` also in focus `x`. If `y` does not exist, `false` is returned.

Two tests:

- *Atom identify test* `x==y`
  If foci `x` and `y` share an atom, `true` is returned; otherwise `false` is returned.
- *Field membership test* `x/f`
  If the atom in focus `x` has a field `f`, `true` is returned, otherwise `false` is returned.

Garbage collection is an extension of MPP, which removes the garbage atoms to save cost of memory space. Garbage atoms are those who are not reachable by any focus through successive field selection and their existence is potentially a problem for computation. We assume there exists a unique atom *null* that collects all the foci that don't focus atoms different from *null* explicitly. We also assume that all the atoms are stored with a so-called *reference counter*, which indicates how many references to the atom exist.

The garbage collection instructions are:

- *Removing a focus* `rma x`
  This instruction can only be successfully executed when the atom in focus `x` has *reference count* 1. This instruction turns the atom in focus `x` into a proto-atom and puts the *null* in focus `x` instead of its original one. Each atom that is directly reachable from the degraded atom has its *reference count* decreased by 1.
- *Restricted garbage collection* `rgc`
  This instruction removes all the atoms that have *reference count* 0 cumulatively. And each atom that is directly reachable from the degraded atom has its *reference count* decreased by 1. Atoms thereafter having *reference count* 0 will be removed by turning them into proto-atoms and so on. Cycles in the reference structure will be left intact when executing this instruction, which means not all garbage need to be removed.
- *Full garbage collection* `fgc`
  This instruction removes cyclic garbage as well as other garbage that can be identified by the *reference count*.

There is also a version of MPP with values, which extends MPP with some instructions that deal with values (integers and Booleans) so called MPPV. Here integers indicate all the non-negative ones.

Mutations:

- *Value field introduction* `x.+f:t`
  Adds a value field named `f` of type `t` for an atom in focus `x`. If the atom in focus `x` does not own a field `f` of type `t`, `true` is returned; otherwise `false` is returned and the instruction is ignored. The value field `f` gets a default value: in case of Boolean, it is `false`; in case of integer it is `0`.

- *Constant value field mutation* x.f=u
  Assigns the value u to the field f of the atom in focus x. If the atom focused by x owns a field f of the appropriate type, true is returned, otherwise false is returned and the instruction is ignored.
- *Value field mutation* x.f=y
  Assigns the value in focus y to the field f of the atom in focus x. If the atom focused by x owns a field f of the appropriate type, true is returned, otherwise false is returned and the instruction is ignored. If the atom in focus y does not have a value, false is returned and this instruction is ignored.

Assignments:

- *Constant value assignment* x=u
  Assigns the value u to the atom in focus x. This instruction returns true by default.
- *Value assignment* x=y
  Puts a copy of the value in focus y in focus x. This instruction returns true by default.
- *Value field selection* x=y.f
  Puts a copy of the value in field f of the atom in focus y in focus x. If the atom in focus y owns a field f, this instruction returns true, otherwise false is returned and the instruction is ignored.

Tests:

- *Constant value identify test* x==u
  This instruction returns true if the value in focus x carries the label u, otherwise returns false.
- *Value identify test* x==y
  This instruction returns true if the value in focus x and y carries the same labels, otherwise returns false.

Here is an example MPPV program:

```
x=new;y=new;x.+f:int;x.f=5;y.+f:bool;y.f=true;x.+link;
x.link=y
```

The following fluids show the states of running the above program. The initial state is depicted in Figure 2.2: there is only a null atom. After executing the first three instructions, the atoms in focuses x and y are established and the atom in focus x has an integer field named f, initialized with 0 (Figure 2.3). After executing the following two instructions, the value 5 is assigned to the f field of the atom in focus x and the atom in focus y has a Boolean field also named f, initialized with false (Figure 2.4). The next two instructions change the value of the field f of the atom in focus y to true and add a new reflexive field link to the atom in focus x (Figure 2.5). The last instruction places the atom in focus y in the field link of the atom in

(Figure 2.6).

Figure 2.2 The initial state

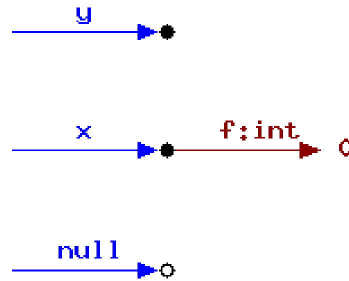


Figure 2.3 The state after executing the first three instructions

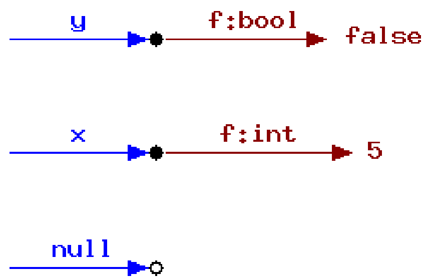


Figure 2.4 The state after executing the next two instructions

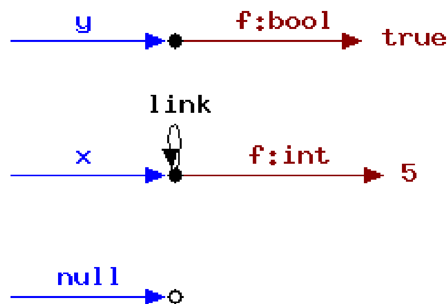


Figure 2.5 The state after executing the seventh instruction

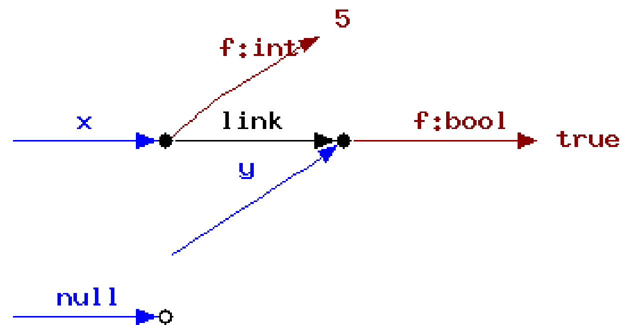


Figure 2.6 The final state focus x

2.3 The PGA Toolset

The PGA Toolset [9] is intended to be used in education and further research. It is a so-called "running environment" of the PGA languages, which contain the primitive instruction sets, their projections and embeddings and the basic instruction sets. These instruction sets are described in the previous sections. In the PGA Toolset, it is possible to test the PGA programs by simulation. The implementation language of the PGA Toolset is Perl [12] because of its regular expressions and ease of manipulation. For graphical views, the language Tcl/Tk[4] is chosen. Both Perl and Tcl/Tk are available for a variety of platforms, which makes the PGA Toolset portable. In the PGA Toolset, a new primitive or basic instruction set can easily be added. Also testing of the programs in the new instruction sets can be done easily.

The PGA Toolset consists of a generic parser, a generic simulator, a parallel simulator and a bisimulator. We can load the appropriate primitive and basic instruction set by the command options to the generic parser. There is an input module that decomposes the input program into primitive instructions. There is a module that parses the primitive instructions from the input module and passes the basic instructions on to another module and this module will parse these basic instructions. There is also a module used to route all the output. The parsing result is stored in the form of a list of instructions in memory. These instructions consist of opcodes and their arguments. The parser also checks and resolves the language constructs that contain more than one instruction, for example, matching "`if`" and "`endif`". The PGA Toolset also implements projections and embeddings, which transform programs in the input language to the

---

[4] In [13], Tcl, Tool Command Language, is an interpreted language with programming features, available for platforms running Unix, Windows and the Apple Macintosh operating system. Tk, the associated toolkit is an easy and efficient way of developing window based applications. The wider availability, usage and ease of teaching and learning of Tcl/Tk makes it the most appropriate tool for teaching the principles of Graphical User Interface design and development.

target language. If there is no one step mapping of the two instruction sets or there is need for information from other instructions, the projections can be used in a pipeline. To invoke a command consisting of a bunch of projections, a program that holds the knowledge of the projection hierarchy and can invoke the proper commands is available from the PGA Toolset.

In the generic simulator, the Toolset executes the instructions from the generic parser. Execution of a primitive instruction may pass a basic instruction to another module, which parses the basic instruction. Execution of a basic instruction returns a Boolean value upon which the primitive instruction can act. We can load several programs into the simulator and there is a module that takes care of the storage and switching of these programs. The generic simulator has both a textual and a graphical user interface, which can be invoked by command options. There is a module that gets the user command from the user interface and acts upon it. The simulation result of running a program in a specific primitive instruction set with a basic instruction set can be seen in a fluid with molecules. To invoke the generic simulator, the command `gensim` with several options such as load the program name or set a breakpoint is used, as default PGLA is used as the primitive instruction set with MPP as the basic instruction set. Figure 2.7 shows the initial state of the generic simulator's graphical user interface after loading the last example program in Section 2.2 and Figure 2.8 shows the state after executing this program.

The PGA Toolset can also control several simulators that all work on one core of the basic instruction set and this controller is so-called parallel simulator. It is implemented by using the ToolBus[5]. The basic instruction set is specified at the start of the parallel simulator. This controller can start new simulators for a specified primitive instruction set and program with that basic instruction set. We can quit the whole parallel simulator or quit each simulator, but in the latter case, the operations working on the core (dump, initialize and update) are blocked.

The PGA Toolset provides a bisimulator that can test the equality of the behavior of two programs. The two test programs are converted to a labeled transition system (LTS) by a generic converter. The programs in PGLA or PGLEc are now available to be converted in the current PGA Toolset. Due to the need of speed in comparing two large programs, the bisimulator is implemented in the language C. Research on the algorithm used in the current bisimulator is

[5] The ToolBus is a software application architecture developed at the University of Amsterdam. It uses a scripting language based on process algebra to describe the communication between software tools. A ToolBus script describes a number of processes that can communicate with each other and with tools residing outside the ToolBus.

going on.

This is a short description of the PGA Toolset. In this Toolset, new primitive or basic instruction sets can be added easily. Furthermore, it is also very easy to test the programs in the new instruction sets due to the generic nature of the parser and the simulator. In our project, we mainly use the first two functions provided by the PGA Toolset. These functions parse and simulate the PGA programs.

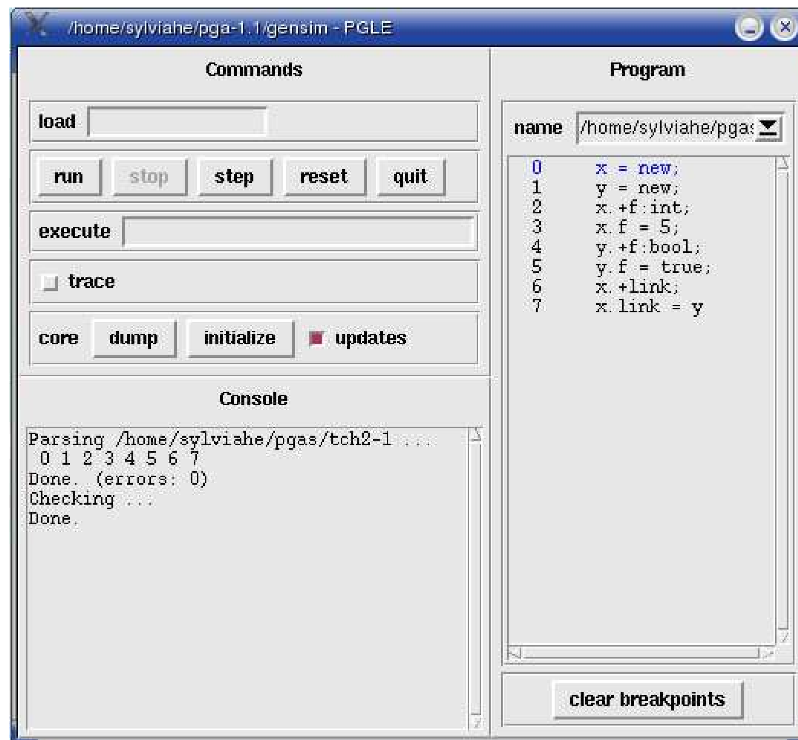


Figure 2.7 The initial state of the graphical user interface after loading a sample program

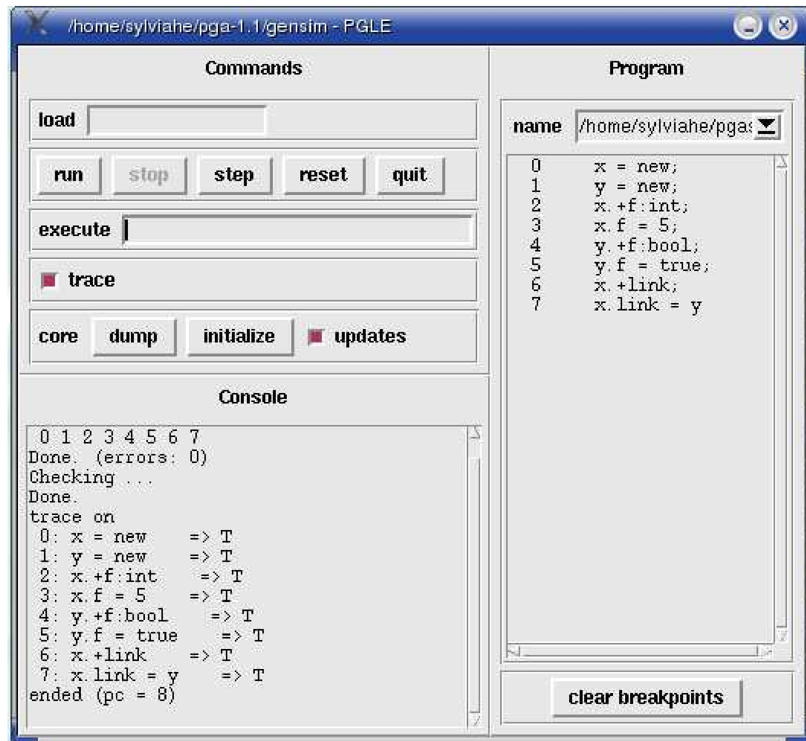


Figure 2.8 The state of the graphical user interface after executing a sample program

Chapter 3

The Intermediate Projection Language

As explained in Chapter 1, the program language is defined in a tuple (RC, φ), where RC is a subset of Ruby and φ a projection from RC to PGA. The projection is a mapping of Ruby programs to program algebra and it is done in stages. Before the input program is projected into an output molecule, it should be first mapped onto an intermediate projection language (IPL). This chapter describes the IPL used in the projection of a Ruby subset to PGA.

In this chapter, we will first introduce some high-level instruction sets of PGA. They are the basis of the IPL in this thesis. The IPL extends PGA with some new instruction sets. These extensions are defined by ourselves and will be introduces in the latter part of this chapter.

3.1 PGLEcr and PGLEcm

PGLE is a primitive instruction set that allows to use labels and corresponding gotos. Labels are natural numbers. `L`*k* denotes a label with name *k* and the corresponding goto instruction `##L`*k* denotes a jump to the first occurrence of label *k*. If no such a label is found, the program terminates. PGLE has a restriction that each test instruction must be followed immediately by a goto or termination instruction. An example program in the primitive instruction set PGLE with MPPV as the basic instruction set is:

`a=2;+a==1;##L1;##L2;L1;b=5;!;L2;b=0`.

In this program, the value of `a` is 2 and because the following test fails, the value of `b` is 0. If we change the program to:

`a=1;+a==1;##L1;##L2;L1;b=5;!;L2;b=0`

Then the test `+a==1` succeeds and the value of `b` is `5`.

PGLEc extends PGLE with conditional constructs.

- *Conditional instructions* `+a{` or `−a{`
  For a basic instruction `a`, `+a{` indicates a positive test of a conditional construct while `−a{` a negative one.
- *Then/else separator* `}{`

Connects two sections of program that are enclosed in braces.

- *End brace* }

  A closing brace in connection with its complementary opening brace.

As an example, the following is a part of a program in PGLEc with MPPV,

```
+x==1{;b=new;b.+v:int;b.v=5;}{;c=new;c.+v:int;c.v=5;}.
```

Here, depending on the value of x, a new focus will be created with value 5.

3.1.1 PGLEcr

PGLEcr extends PGLEc with recursion: here we have in addition the returning goto instruction `R##L`*k* and the return instruction `R`. `R##L`*k* performs a goto and a jump back to the instruction immediately after itself whenever it encounters an `R` instruction. Here *k* is a natural number. Let's look at this example,

```
a=5;b=1;L1;+a==b;R;incr b;R##L1;c=new
```

Here the value of `b` keeps increasing until it is equal to the value of `a`, then the program returns to execute the next instruction `c=new`.

The projection of PGLEcr to PGLEc uses a data structure `stackframe`, which keeps track of the program's control points. We assume that there exists a fieldless atom in focus `stackframe`. There is also another stackframe that we used in the projection of the Ruby subset to PGA. In order to avoid confusion, we used `StackFrame` to take the place of `stackframe` that was used in [9] in the projection of the Ruby subset to PGA. Suppose the maximal numerical label in the program is *max*, to avoid label clashes, we make fresh label numbers with the increase of *max*+1. The projection `pglecr2pglec` is defined using the operation ψ_i (for $1 \le i \le n$). Given a PGLEcr program $u_1 ; \ldots ; u_n$ we write

$$\texttt{pglecr2pglec}(u_1 ; \ldots ; u_n) = \psi_1(u_1) ; \ldots ; \psi_n(u_n)$$

- ψ_i(`R##L`k)=

```
aux=new;aux.+baxk;aux.back=stackframe;
stackframe.+next;stackframe.next=aux;
aux.+label:int;aux.label=max+1+k;
stackframe=aux;aux=null;##Lk;L(max+1+k)
```

- ψ_i(`R`)=

```
-stackframe/label;!;label=stackframe.label;
stackframe=stackframe.back;stackframe.-next;
##L[label]
```

- $\psi_i(u)=u$ otherwise.

Here the focus depending goto instruction `##L[label]` abbreviates

`+label==`*max*`+1+1;##L(`*max*`+1+1);+label==`*max*`+1+2;##L(`*max*`+1+2);...;`

`+label==`*max*`+1+`*m*`;##L(`*max*`+1+`*m*`)`

where *m* is the maximal label value of the returning goto instructions in the PGLEcr program and *max*+1+*m* is the maximal return label on the stack.

The projection result of the above PGLEcr programs is as below:

```
stackframe=new;a=5;b=1;L1;+a==b;-stackframe/label;!;
label=stackframe.label;stackframe=stackframe.back;
stackframe.-next;+label==3;##L3;incr b;aux=new;aux.+back;
aux.back=stackframe;stackframe.+next;stackframe.next=aux;
aux.+label:int;aux.label=3;stackframe=aux;aux=null;
##L1;L3;c=new
```

3.1.2 PGLEcm

PGLEcm extends PGLEc with method calls. For example,

```
node1=new;node2=new;method(node1,node2);
method(a,b){;a.+next;a.next=b;}
```

This method has two parameters `a` and `b`. In the method call, the corresponding arguments are `node1` and `node2`. After execution, `node1` will have a `next` field that links up with `node2`. PGLEcm programs are not executable in the current PGA Toolset, but they can be projected to PGLEcr programs first and then to PGLEc programs. The returning goto and return instructions that are introduced in Section 3.1.1 serve as the basis in the projection of method calls. We introduce the syntax and projection for method calls in stages.

1) Void unparameterized static method calls

We first introduce the simplest form: the void unparameterized static method call, which performs a computation without returning a value.

- *Method definition* `m`*k*`(){;`u_{i+1}`;…;`u_{i+m}`;}`
- *Method call* `m`*k*`()`

Here k is a natural number and instructions u_{i+1} ;…; u_{i+m} ; make up the body of the method. In the projection, in order to avoid label clashes, fresh labels are introduced with the increase of max+1, where max is the maximal label number in the program. The projection of PGLEcm to PGLEcr of this kind of method definition and method call is the following:

- ψ_i (mk(){; u_{i+1} ;…; u_{i+m} ;})=L(max+1+k); ψ_{i+1} (u_{i+1});…; ψ_{i+m} (u_{i+m});R
- ψ_i (mk())=R##L(max+1+k)

2) Non-void unparameterized static method calls

Here we introduce the non-void unparameterized static method calls. A focus `that` is used to return the value and the method body must contain at least one assignment of the form `that=y`. The syntax of the method definition and method call:

- *Method definition* mk(){; u_{i+1} ;…;that=y;…; u_{i+m} ;}
- *Method call* x=mk()

The projection of the method definition is similar to that of the void case. The projection of a method call is:

- ψ_i (x=mk())=mk();x=that

3) Non-void parameterized static method calls

We assume there exist some fixed foci `arg1,…,argj`. They may occur in the method body and focus the method's parameters during the method call to incorporate the parameters. The syntax of non-void parameterized static method definition and method call:

- *Method definition* mk(arg1,…,argj){; u_{i+1} ;…; u_{i+m} ;}
- *Method call* x=mk(v1,…,vj)

The projection of the method definition is similar to the unparameterized one. The projection of the method call:

- ψ_i (x=mk(v1,…,vj))=

  stackframe.+arg1=arg1;…;stackframe.+argj=argj;
  arg1=v1;…;argj=vj;
  x=mk();arg1=stackframe.arg1;…;argj=stackframe.argj;
  stackframe.-arg1;…;stackframe.-argj

Now we can deal with the methods with arbitrary parameters `p1,…,pj` easily. The syntax and projection of arbitrary parameterized method definition is:

- *Method definition* mk`(p1,…,pj){;` u_{i+1} `;…;` u_{i+m} `;}`

- ψ_i (mk`(p1,…,pj){;` u_{i+1} `;…;` u_{i+m} `;})=`

  `mk(arg1,…,argj){` ψ_{i+1} (u_{i+1} `[p:=arg]);…;` ψ_{i+m} (u_{i+m} `[p:=arg]);}`

Here u_l `[p:=arg]` denotes the instruction that substitutes `p1,…,pj` by `arg1,…,argj`.

4) Non-void parameterized instance method calls

The last form to be introduced is non-void parameterized instance method calls. The syntax and the projection of the method definition remain unchanged. The syntax of the non-void parameterized instance method call:

- *Method call* `y=x.m`k`(v1,…,vj)`

The projection of the method call is with the help of an additional field `this` of the stackframe. The calling object is put into a focus `this` when the method is called. After the call, the current object is returned into focus `this`:

- ψ_i `(y=x.m`k`(v1,…,vj))=`

```
stackframe.+this=this;this=x;y=mk(v1,…,vj);
this=stackframe.this;stackframe.-this
```

The projection result in the PGA Toolset of the example at the beginning of this section is:

```
node1=new;node2=new;stackframe.+a;stackframe.a=a;a=node1;
stackframe.+b;stackframe.b=b;b=node2;R##L1;a=stackframe.a;
stackframe.-a;b=stackframe.b;stackframe.-b;L1;a.+next;
a.next=b;R
```

The execution result of this PGLEcr program is depicted in Figure 3.1.

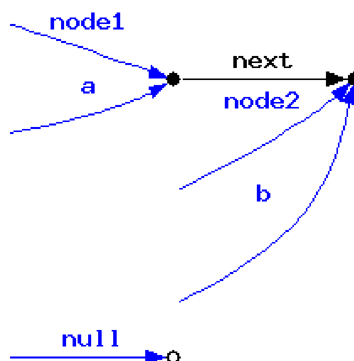


Figure 3.1 The execution result of the PGLEcr program

3.2 MSP and MSPea

Before introducing MSP and MSPea, we first introduce HMPPV (High-level MPPV). It adds some instructions that are the shorthand for commonly used combinations of MPPV instructions. H-MPPV also extends the types of MPPV[6] with strings. Let's have a look at one of the new instructions as an example:

```
extfocus1.+f=extfocus2
```

An `extfocus` (extended focus) denotes either a focus or a field selection that may be compound. This instruction adds a new field `f` to the atom selected by `extfocus1`, and assigns it the atom selected by `extfocus2`. It is the shorthand of these two instructions:

```
extfocus1.+f;extfocus1.f=extfocus2.
```

3.2.1 MSP

MSP (Molecular Scripting Primitives) extends HMPPV with some new instructions. If the focus or the field selection does not exist, `false` is returned as the result. If the data type of the selected value is not correct, also `false` is returned.

Operation on non-negative integers:

- `incr extfocus`

  Increases the integer value selected by `extfocus` with `1`.
- `incr extfocus n`

  Increases the integer value selected by `extfocus` with `n`.

[6] MPPV has two value types: Boolean and non-negative integer.

- `incr extfocus1 extfocus2`
  Increases the integer value selected by `extfocus1` with the integer value selected by `extfocus2`.
- `decr extfocus`
  Decreases the integer value selected by `extfocus` with `1`. Returns `false` if the value was already `0`.
- `decr extfocus n`
  Decreases the integer value selected by `extfocus` with `n` if the value is larger than or equal to `n`. Returns `false` otherwise.
- `decr extfocus1 extfocus2`
  Decreases the integer value selected by `extfocus1` with the integer value selected by `extfocus2` if it is larger than or equal to the value selected by `extfocus2`. Returns `false` otherwise.

Operations on strings:

- `first extfocus1 extfocus2`
  Takes the first character of the string selected by `extfocus1` and assigns it as a string of length 1 to `extfocus2`. If the selected string is an empty string, `false` is returned.
- `delfirst extfocus`
  Removes the first character from the string selected by `extfocus`. If the string is empty, returns `false`.
- `append extfocus1 extfocus2`
  Appends the string selected by `extfocus2` to the end of the string selected by `extfocus1`.

Type conversions:

- `int extfocus1 extfocus2`
  Converts the string selected by `extfocus1` to an integer value and assigns it to `extfocus2`. If no integer value can be obtained from the string, returns `false`.
- `str extfocus1 extfocus2`
  Converts the integer value selected by `extfocus1` to a string and assigns it to `extfocus2`.

3.2.2 MSPea

MSPea (MSP with eval/apply) extends MSP with following instructions for evaluation, application and compilation.

- `compile extfocus`
  This instruction compiles the string selected by `extfocus` into a molecule and assigns it

to the atom in focus `extfocus`. It can only compile the string that represents a program in the primitive instruction set PGLA with the basic instruction set MSP. If there is any error, `false` is returned.

- `apply extfocus`
  If `extfocus` is a string whose content is a basic instruction, then the basic instruction is executed and its result is returned as the result of the `apply` instruction. Otherwise the instruction fails and returns `false`.
- `eval extfocus`
  If `extfocus` is a string, this instruction compiles the string selected by `extfocus` into a molecule and evaluates the molecule. If `extfocus` is an atom, it is evaluated. Otherwise the evaluation fails and `false` is returned. It does not do the assignment as `compile` does. The result of the last evaluated basic instruction is returned as the result of the `eval` instruction.

If the string in `extfocus` is a program in the primitive instruction set PGLA with the basic instruction set MSP, then the compilation result is a molecule that represents a list of instructions by a list of atoms connected by `next` fields. In this list each atom has additional fields that represent the content of the corresponding instruction. Let's have a look at this example:

```
a=11;b=1;decr a;incr b;+a==b;!;\\#4
```

Figure 3.2 shows the execution result of this PGLA program. Then we put this PGLA program

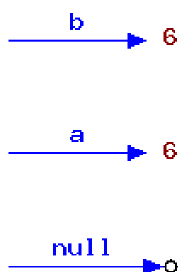


Figure 3.2 The execution result of the PGLA program

in a string `x` and compile this string.

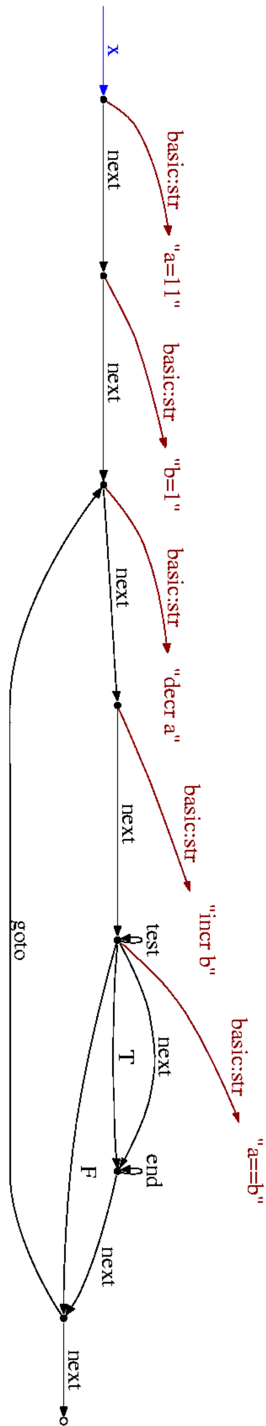
```
x="a=11;b=1;decr a;incr b;+a==b;!;\\\\#4";compile x
```

The result is depicted in Figure 3.3. In this picture, we can see that there are several kinds of additional field of each atom in the list. For example, there is an atom with a field named `end`, this indicates that this is a termination instruction and the evaluation terminates. The atom before the "`end`" atom has four fields besides the `next` field: `test`, `basic`, `T` and `F`. `test`

means this is a test instruction and the test content is saved in the basic field. The next instruction to be evaluated depends on the return value of the basic instruction in the basic field and it is the instruction pointed either by T or F field. The atom with a goto field indicates a repeat instruction and the first instruction to be repeated is selected by the goto field. After compilation, the list of atoms is assigned to x.

If we evaluate the string x instead of compiling it, the result fluid will be slightly different: there is no assignment to x after evaluation and the result of executing the program in x is given in the fluid also.

In our project, we only make use of the compile instruction to separate the input Ruby program by semicolons. A Ruby program obviously is not in the proper program notation that can be compiled, so there is no compilation result in the fluid. This will be explained in detail in Section 12.1.



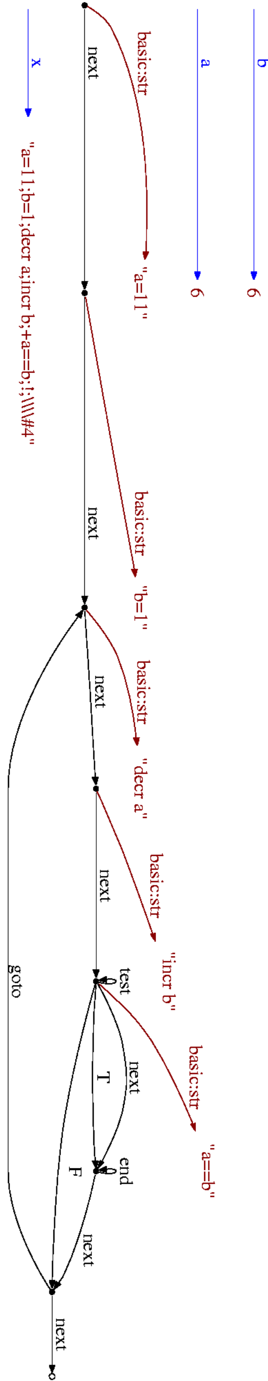


Figure 3.3 The compilation result of the previous PGLA program

Figure 3.4 The evaluation result of the previous PGLA program

3.3 Extensions

In order to implement the projection of a Ruby subset to PGA as well as to make the implementation easier, some extensions to the PGA language hierarchy were made: these are string labels instructions and the instructions of multiplication and exponent. The primitive instruction sets that will be discussed in this thesis are on the top of the hierarchy of the program notations in Chapter 2 and they can be projected to lower level ones.

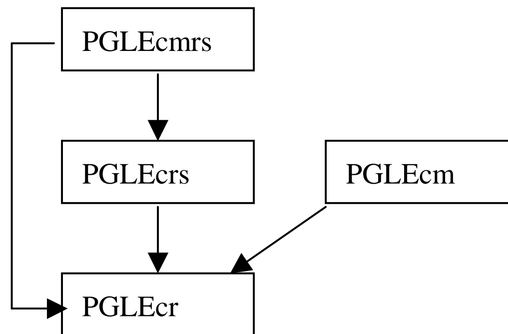


Figure 3.5 The hierarchy of the extended program notations

3.3.1 PGLEcrs

PGLEcrs extends PGLEcr with the string labels, gotos and returning gotos, where the names of labels are strings in double quotes, such as `L"class"`, `##L"class"` and `R##L"class"`.

The projection of the string labels, gotos and returning gotos to the numerical ones works in the same way as the description in [9]: there is a dictionary D^x of all the unique strings appearing in the string labels, gotos and returning gotos in the program *X*. The dictionary has indexes of all the strings from 1 to *n*, where *n* is the size of D^x.

- ψ(Ls)=L($i+m$)
- ψ(##Ls)=##L($i+m$)
- ψ(R##Ls)=R##L($i+m$)

Here *s* is a string in double quotes, *m* is the maximal numerical label in the program and *i* is the index of the string *s* in D^x.

Consider this part of a PGLEcrs program,

```
L100;method=new;methods.+initialize;
methods.initialize=method;class="shape";
method.+label:str="shape:initialize";
```

```
##L"shape:initialize:skip";
L"shape:initialize";arguments=new;
arguments.+p1=arg1;R;
L"shape:initialize:skip";label=method.label;
R##L"shape:initialize"
```

After projection, all the string labels are changed to corresponding numerical ones:

```
L100;method=new;methods.+initialize;
methods.initialize=method;class="shape";
method.+label:str="shape:initialize";
##L101;L102;arguments=new;arguments.+p1=arg1;
R;L101;label=method.label;R##L102
```

3.3.2 PGLEcmrs

PGLEcmrs extends the combination of PGLEcr and PGLEcm with string labels, gotos and returning gotos as described in the last section. Our projection of Ruby to PGA is in this primitive instruction set. Because PGLEcm is not executable but projectable in the current PGA Toolset, PGLEcmrs is also only projectable but not executable. We will first project a PGLEcmrs program to a PGLEcrs one. The projection is based on that of PGLEcm to PGLEcr, which was explained in Section 3.1.2, leaving the string labels unchanged. Then we project the PGLEcrs program to a PGLEcr one. In this step, all the string labels are projected to numerical ones. At last, we execute the PGLEcr programs. It is also possible to project a PGLEcmrs program directly to a PGLEcr one and execute this program in the PGA Toolset.

3.3.3 MSPeame

In order to make our projection of a Ruby subset to PGA easier to implement, we have extended the basic instruction set MSPea with the instructions described below and call this MSPeame.

An `extfocus` is used in the instructions described below to denote either an existing focus or a field selection that may also be compound. If the focus or the field selection does not exist, `false` is returned as the result. If the data type of the selected value is not correct, also `false` is returned.

Operations on integers are:

- `mult extfocus` *n*

  Multiplies the integer value selected by `extfocus1` with *n* and *n* is an integer (here an

integer means a non-negative integer).

- `mult extfocus1 extfocus2`

  Multiplies the integer value selected by `extfocus1` with the integer value selected by `extfocus2` and save the result in `extfocus1`. For example, `x=2;y=3;mult x y` Here 6 is the result of the multiplication and it is assigned to `x`.
- `expo extfocus` *n*

  Exponentiation the integer value selected by `extfocus1` with *n* and *n* is an integer.

  `expo extfocus1 extfocus2`

  Exponentiation the integer value selected by `extfocus1` with the integer value selected by `extfocus2` and save the result in `extfocus1`. For example, `x=2;y=3;expo x y` Here 8 is the result of the exponentiation and it is assigned to `x`.

Consider the following program:

```
x=5;y=2;+x==5{;expo x y;}{;mult x y;}
```

The execution result of this program is that the value of `x` is `25` and that of `y` is unchanged.

We will continue to explain how to implement these extensions in the PGA Toolset in Chapter 4.

Chapter 4

Implementation of the Extensions in the PGA Toolset

We introduced some details of PGA and the PGA Toolset in Chapter 2. In Chapter 3, we described the IPL we used in this project: the combination of PGLEcmrs and MSPeame. Both the primitive and the basic instruction sets in the IPL are extensions of PGA and the PGA Toolset. In this chapter, we discuss how to implement the extensions in the PGA Toolset according to the syntax we gave in the Chapter 3.

We can use the `-L` option of the PGA Toolset's generic simulator to load the new instruction set easily. The modules and programs for implementing the extensions in the PGA Toolset can be found in Appendix C.

4.1 Extending the PGA Toolset with PGLEcrs and PGLEcmrs

First of all, we explain why we chose to extend PGA and the PGA Toolset with PGLEcmrs. According to the projection functions in [9], numerical labels and gotos are not enough for the projection. We need string labels and gotos as well as the variable returning gotos. String labels make the label instructions more descriptive and we explained the syntax and projection of string labels Section 3.3.1. Variable returning goto, `R##L[a]` denotes a returning goto instruction whose destination depends on the value of the atom in focus `a`. This kind of instruction is mainly used in the projection function of a method call: whenever a method is called, we should jump to the body of the method definition and transfer the parameters, this jump is implemented by a variable returning goto instruction. The corresponding part of code is:

```
label=method.label;R##L[label]
```

Here the atom in focus `label` is assigned to the `label` field of the atom in focus `method`. Each method has its own `label` field. The `label` field denotes the position of the method body. Executing the instruction `R##L[label]`, the control point of the program jumps to the method body as we supposed. After executing the method body, an `R` instruction turns the

control back to the instruction immediately after `R##L[label]`.

The variable returning goto is a very convenient instruction for the projection of the method. We first tried to extend PGA and the PGA Toolset with variable label, goto and returning goto instructions and named the new primitive instruction set PGLEcmrv (v denotes the variable labels and r denotes the returning gotos). In order to determine the value of a focus, we used a series of test instructions. This method worked well with the variable goto and returning goto instructions. For example, the variable goto instruction `##L[a]`, a part of its projection looked like this:

+a==1;##L1;+a==2;##L2;…;+a==*max*;##L*max*

Here *max* is the maximal numerical label value in the whole program.

But this method leaded to label clashes when dealing with variable labels. We projected a variable label `L[a]` to a numerical one like this:

+a==1;L1;+a==2;L2;…;+a==*max*;L*max*

Whenever there is a variable label, `L1`, `L2`, … and L*max* are created. If there is more than one variable label in a program, then in the projection result there will have label clashes, i.e. repeated groups of `L1`, `L2`, … and L*max* will appear in the projection result. It was too complex and difficult to solve this problem. To avoid sticking on this point, we decided to extend PGA and the PGA Toolset with string labels and gotos only, which is the primitive instruction set PGLEcmrs we described in Chapter 3. We changed the projection function of Ruby's method call to avoid using the variable returning gotos. This will be further discussed in Chapter 5.

To implement the new primitive instruction set PGLEcmrs, we first extended the PGA Toolset to recognize the string labels, gotos and returning gotos by using regular expressions. We first look through the whole PGA program for instructions that coincide with the given regular expressions of string label, goto and returning goto instruction. Then we tag these instructions with "string label", "string goto" or "string returning goto". At the same time, the operands of these instructions, in these cases strings, are saved in an array. Then we treat the string labels, gotos and returning gotos in the same way as we do with the numerical ones: print the instructions, check the labels for goto and returning goto instructions (i.e. check whether the corresponding labels of these two kinds of instructions are available in the program) and execute these instructions by making use of their operands that are stored in the array.

We use the following Perl-script to "recognize" the string labels, gotos and returning gotos.

```
if ($i =~ /^L($string)$/) {
```

```
        $prog[$ic ++] = [( 'LABELSTR', $1)];
    } elsif ($i =~ /^##L($string)$/) {
        $prog[$ic ++] = [( 'GOTOSTR', $1, '?')];
    } elsif ($i =~ /^R##L($string)$/) {
        $prog[$ic ++] = [( 'RETGOTOSTR', $1, '?')];
```

Here the Perl-variable `$string` is a regular expression that denotes an arbitrary string in double quotes. The Perl-variable `$i` holds the input instruction. `$proc[]` is an element of the Perl array variable `@proc`. We push the name of the input instruction, such as `'LABELSTR'` (string label) and `'GOTOSTR'`(string goto), and its operands to the array variable `@proc`. If the input instruction is goto or returning goto that should have a matching label instruction available in the program, we tag it with a question mark `'?'`. Later on, when we check the whole program, for each goto or returning goto instruction, if its corresponding label is found, we use a variable that denotes its corresponding label to take the place of the `'?'`. Otherwise, an error message is given.

The projection of PGLEcmrs to PGLEcrs is almost the same as the projection of PGLEcm to PGLEcr. We only project the method relevant instructions and leave all the other instructions unchanged. The projection functions are described in section 3.1.2.

As for the projection of PGLEcrs to PGLEcr, we first use a loop to find out the maximal numerical label in the whole program. Then we make a dictionary for all the string names of the labels. We use a hash to mimic the dictionary. The string names are keys and each of them has a unique corresponding number, which progressively increases by 1, as the values of the hash. Figure 4.1 shows the structure of the hash. Thus all the unique string names of the labels have their corresponding numbers. The projection directly from PGLEcmrs to PGLEcr is the combination of the projections PGLEcmrs to PGLEcrs and PGLEcrs to PGLEcr. The Perl-module for implementing this projection is now available in the PGA Toolset.

In the following Perl-program, we check the whole program to project the string labels, gotos and returning gotos to the numerical ones. Whenever we encounter a string label, goto or returning goto instruction, we use the name of the label to find out its corresponding number in the hash we described above, and increase it with the maximal numerical label to get this string label's corresponding numerical label.

Key	Value
“initialize”	1
“add”	2
“equaltest”	3
……	…
“newmethod”	n

Figure 4.1 The structure of the hash

The following Perl-script is used to build up the “dictionary”.

```
if ($opc eq 'LABELSTR' || $opc eq 'GOTOSTR' ||
$opc eq 'RETGOTOSTR') {
        $l = shift @i;
        if (!(exists $str{$l})){
        $str{$l} = $va;
        $va ++;
        }
}
```

The Perl hash variable `%str` is used to mimic the dictionary and `$str{}` denotes its elements. The instruction `!(exists $str{$l})` is used to make sure that this string label, goto or returning goto instruction appears for the first time in the whole program. The Perl-variable `$va` begins with 1, it denotes the corresponding number of the label’s string name.

Here is an example of executing a PGLEcrs program in the PGA Toolset.

```
a=11;b=1;L"repeat";+a==b;R;decr a;incr b;R##L"repeat";c=a
```

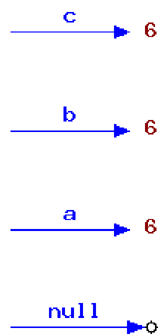


Figure 4.2 The running result of the sample PGLEcrs program

As explained above, we first project the programs in PGLEcmrs to PGLEcrs. Then we project the PGLEcrs programs to PGLEcr ones. At last we execute the PGLEcr programs with

MSPeame as the basic instruction set. Here is an example of the PGLEcmrs program,

```
L3;##L"able:skip";L"able";arguments=new;
arguments.+v:int=3;R;L"able:skip";
a=new;b=new;a.+next=b;b.+back=a;
R##L"able";a.+v:str="able";
n1=new;n2=new;me(n1,n2);
me(c,d){;c.+next=d;d.+back=c;}
```

Its projected result in PGLEcrs is:

```
L3;##L"able:skip";L"able";arguments=new;
arguments.+v:int=3;R;L"able:skip";
a=new;b=new;a.+next=b;b.+back=a;
R##L"able";a.+v:str="able";
n1=new;n2=new;stackframe.+c;stackframe.c=c;
c=n1;stackframe.+d;stackframe.d=d;
d=n2;R##L1003;c=stackframe.c;stackframe.-c;
d=stackframe.d;stackframe.-d;L1003;
c.+next=d;d.+back=c;R
```

Then we project this PGLEcrs program to PGLEcr and execute the PGLEcr program in the PGA Toolset. Figure 4.3 shows the execution result in our extension of the PGA Toolset:

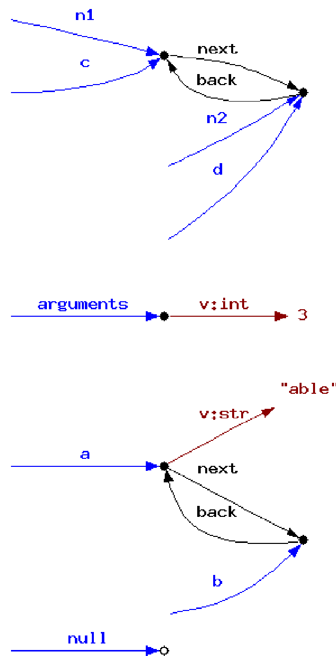


Figure 4.3 The execution result of a PGLEcr program

4.2 Extending the PGA Toolset with MSPeame

We described the syntax of MSPeame in Section 3.3.3. This new basic instruction set extends MSPea with several operations for integers.

The implementation of MSPeame is done in two steps: first, make the PGA Toolset recognize the new instructions by making use of corresponding regular expressions and tag the instructions with their names, and save the operands of these instructions in an array, like what we do with the string label instructions; second, we deal with the instructions separately, do their corresponding operations, and return the results. For example, to deal with the instruction `mult x 5`, we first recognize this instruction: multiply a variable `x` with 5 and tag it with "multiplication". We save the operands `x` and `5` in an array. Then our extension of the PGA Toolset "reads" the program again. When encountering the tag "multiplication", our extension of the PGA Toolset executes the corresponding operation and returns the result to variable `x`. At last `x` has its new value.

The Perl-script used to "recognize" the basic instructions is in the same style as that we use to deal with the primitive instruction set PGLEcmrs in the last section.

```
if ($i =~ /^mult\s+($object)\s+($value{int})$/) {
  @prim = ( 'MULT', $1, $2);
} elsif ($i =~ /^mult\s+($object)\s+($object)$/) {
  @prim = ( 'MULTVAR', $1, $2 );
} elsif ($i =~ /^expo\s+($object)\s+($value{int})$/) {
  @prim = ( 'EXPO', $1, $2);
} elsif ($i =~ /^expo\s+($object)\s+($object)$/) {
  @prim = ( 'EXPOVAR', $1, $2 );
}
```

Here the Perl-variable `$object` is a regular expression that denotes a PGA variable (an atom) and `$value{int}` is an element of a hash that denotes an integer. In the first case of this "`if` control structure", the input instruction is a multiplication of a variable and an integer, while in the second case, both operands of the multiplication are variables. We give them different names: `'MULT'` and `'MULTVAR'`. This is because when implementing the operation, if the operand is a variable, we should call a Perl-subroutine to get its value while if it is an integer, we don't need to do so. We push the name of the instruction and its operands into an array `@prim`. We treat the exponentiation in the same style.

Part of the Perl-script of implementing a multiplication is this:

```
if ($opc eq 'MULT'){
   $x = shift @i;
   $v = shift @i;
   if (IsReserved($x)) {
         return 0;
   }
   if ($a = GetObjectTypedValue($x, 'int')) {
         return AssignValue($x, GetValue($a) * $v, 'int');
   } else {
         return 0;
   }
}
```

In the PGA Toolset, a PGA variable is saved in such a form: `$atom[$a]`. All the variables form a Perl-array `@atom`. `$atom[$a]` is an element of this array and `$a` is the name of the PGA variable. This `$atom[$a]` itself is a hash, storing the data type and the value of this variable.

We first pop the name of the instruction from the Perl-array `@i` (we assign `@prim` to `@i`) and determine the next block of code we have to execute. In this case, it is `'MULT'`. Then we pop the two operands from the array `@i` again. In the following code, we call four Perl-subroutines. The first one, `IsReserved()` is to determine whether the name of a Perl-variable is a reserved word in PGA. If it is, `0` is returned and the program terminates. The second Perl-subroutines, `GetObjectTypedValue()` first determines whether a PGA variable is of a certain data type: if it is, return its value, otherwise `undefined` is returned and the program terminates. `AssignValue()` is used to assign a value of a certain data type to a PGA variable. `GetValue($a)` is used to get the value of a PGA variable without determining its data type. In this block of Perl-script, the second operand of the multiplication is an integer, so we don't need to determine whether its name is a reserved word in PGA and whether it is a certain data type. If both operands are PGA variables, then the code is a little bit complex because both operands need to be preprocessed (check for reserved words in PGA and the variable's data type). The core block of the Perl-script to implement a multiplication of two PGA variables is:

```
if ($a = GetObjectTypedValue($x, 'int')) {
      if ($ay = GetObjectTypedValue($y, 'int')) {
        return AssignValue($x, GetValue($a) * GetValue($ay),
        'int');
      }
```

```
} else {
  return 0
}
```

Figure 4.4 shows the execution result of the example in section 3.3.3 in our extension of the PGA Toolset.

```
x=5;y=2;+x==5{;expo x y;}{;mult x y;}
```

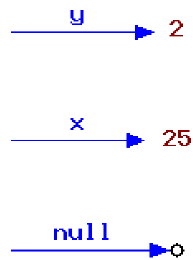


Figure 4.4 The execution result of the sample program

Chapter 5

Implementation of Geerlings' Projections in IPL

In [9] four subsets of Ruby have been presented starting with RC1, RC2, RC3 and RC4. RC1 has all the basic constructs of OO languages: class, methods, and objects. Besides OO constructs, it also contains simple control instructions, local variables and instance variables. And the projection of an entire RC1 program is given as an example in RC1. RC2 changed the projection of RC1 by adapting the method call instruction to singleton methods. And RC2 extends RC1 with a new type of method: class method. RC3 and RC4 added features but did not change instructions of RC1 and RC2. RC3 adds class variables. RC4 adds integers as a basic type.

In this chapter we will introduce the constructs of the parser that is the core of our implementation and describe how the projection functions in [9] are used in the parser. Some parts of our programs (in Appendix B) will be mentioned and explained in this chapter.

5.1 Framework

The core of our implementation is the design of the parser (ψ). Before the input program is projected into an output molecule, it should be first mapped onto the IPL defined in Chapter 3. The parser is based on IPL and used for projecting programs from a subset of Ruby into PGA programs. The scheme below depicts the framework of the parser. The commands in between are used to implement the corresponding projections.

- `pglecmrs2pglecr` *<result of the parser> output*
  This command is used to project PGLEcmrs programs to PGLEcr programs. They read the result of the parser and write to standard output.
- `gensim -P PGLEcr -B MSPeame -l` *output*
  This is a typical invocation of the generic simulator, which uses `PGLEcr` as set of

primitives and MSPeame as basic instruction set. It loads program from the standard output and makes it the current one.

- `r`

  This is a command that used in the generic simulator. It runs the current program.
- `output=17`

  This is an example running result of a Ruby program, which outputs an integer `17`.
- `output → 17`

  This is a part of the running result of the relevant PGA molecule. In this output molecule, there must be a focus holding an integer value `17`.

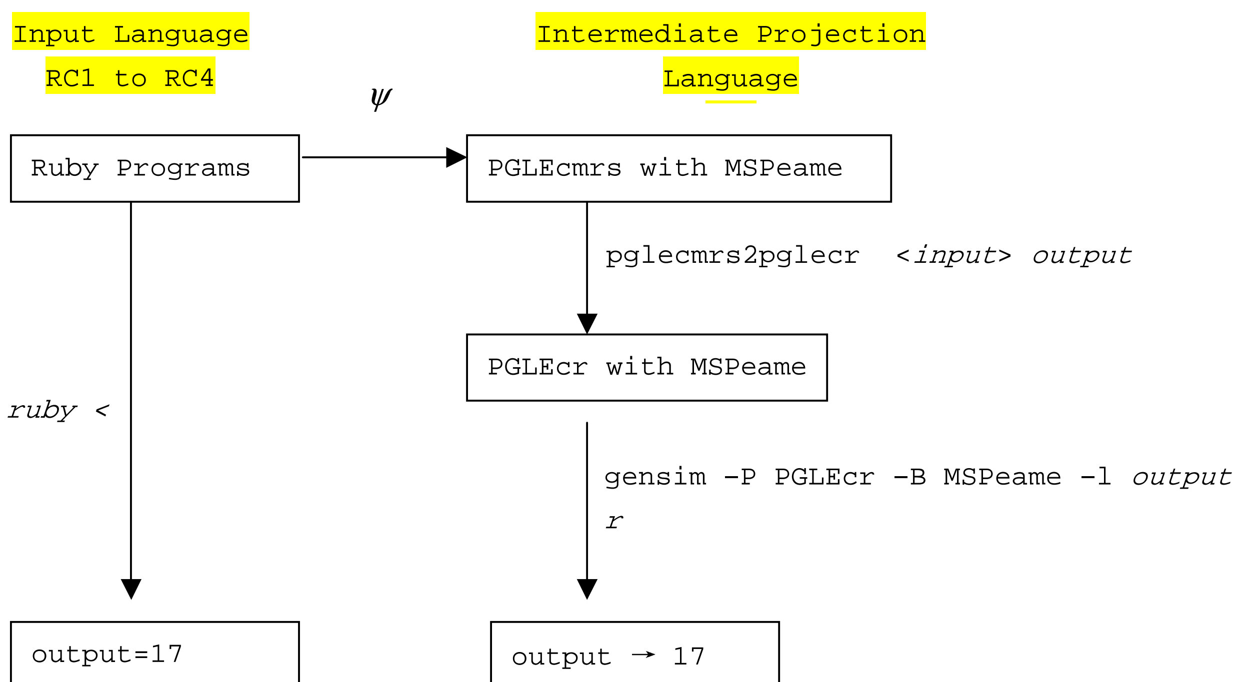


Figure 5.1 Scheme

The results can be seen after running a Ruby example program. The result executed in program algebra should be similar to the result run on a common Ruby platform.

5.2 Keyword parsing

Before reading a Ruby program, a keyword table should be built. The table is composed of a list of keyword and projection function pairs. The keywords are used as index, and each projection function defined in RC1 to RC4 is mapped to a keyword. Every projection function is written

between braces “{}” along with a unique method name. Table 5.1 lists all keywords and their corresponding methods that contain projection functions used in RC1 to RC4.

Keywords	**Method**
`class`	`ClassDefinition`
`def`	`MethodDefinition`
`return`	`Return`
`=`	`Assignment`
`end`	`End`
`()`	`MethodCall`
`if`	`If`
`==`	`Equalitytest`
`+`	`Add`

Table 5.1 Keywords and their responding methods

For example, the projection function of a method definition in RC1 is written between braces with a method name “`MethodDefinition`”. It is mapped to the keyword “`def`”. The parameters `cl` and `m` are the current class name and method name in Ruby programs.

```
MethodDefinition(cl,m){;
self=StackFrame.self;
+self==main{;cl=Object;}{;cl=self;};
-cl/im{;cl.+im=new;};
methods=cl.im;
method=new;
methods.+m;
method.+label:str="cl:m";
##L"cl:m:skip";

L"cl:m";
arguments=new;arguments.+p1=agr1;...;arguments.+pn=argn;
StackFrame.+lv=arguments;StackFrame.+self=self;
ψm(u1);...;ψm(uk);
R;

L"cl:m:skip";};
```

Method `m` is stored in `cl.im`, where `cl` is the class name and `im` is the field that points to the object that holds all the instance methods. There can be many methods with the same name in a class hierarchy. Every method is assigned with a unique string label `"cl:m"` to avoid a

label clash and the alias problem. This label is the combination of its class name and method name.

The jump after it: `##L"cl:m:skip"` is necessary to prevent the program from executing the method when it encounters the definition. Programs will directly jump to the label instruction `L"cl:m:skip"` and continue to execute the latter instructions. In the label instruction `L"cl:m:skip"`, the string `"skip"` following the class name and method name is used to distinguish from the label instruction `L"cl:m"`. Just as its name implies, the instructions after it will be skipped.

Whenever the method is called the program jumps to the label instruction `L"cl:m"` in the definition. And before it executes the instructions of the method block, the local variables are initiated with the method arguments and the self reference is updated.

5.3 Initialization

The second step is the initialization of PGA molecules. It is the initial state of the output and the entry to the entire projection. Before any statements in the program are projected, seven classes must be created: the `Object`, `Class`, `TrueClass`, `FalseClass`, `NilClass`, `Integer` and `Fixnum` classes. The Object class is the superclass of all other classes. And every class is an instance of the class `Class` (including itself). The following three classes are made for the keyword objects `true`, false and nil. The last two classes are made for the integer type defined in RC4.

- This macro creates predefined classes and constants.

$\varphi_{init-classes}$ =

```
Object=new;
Class=new;Class.+super=Object;
TrueClass=new;TrueClass.+super=Object;
FalseClass=new;FalseClass.+super=Object;
NilClass=new;NilClass.+super=Object;
Integer=new;Integer.+super=Object;
Fixnum=new;Fixnum.+super=Integer;
Object.+class=Class;
Class.+class=Class;
TrueClass.+class=Class;
```

```
FalseClass.+class=Class;
NilClass.+class=Class;
Integer.+class=Class;
Fixnum.+class=Class;

True[2]=new;True.+class=TrueClass;
False=new;False.+class=FalseClass;
nil=new;nil.+class=NilClass;
main=new;main.+class=Object;
```

Part of the initial program state is depicted in Figure 5.2. The whole fluid of the running result can be found in Appendix A, Initialization.gif.

[2] As `true` and `false` are keywords in PGA they can not be used as names of field or foci. Therefore the `true` and `false` objects defined in the first three paragraphs of $\varphi_{init-classes}$ in RC1 are replaced by `True` and `False`.

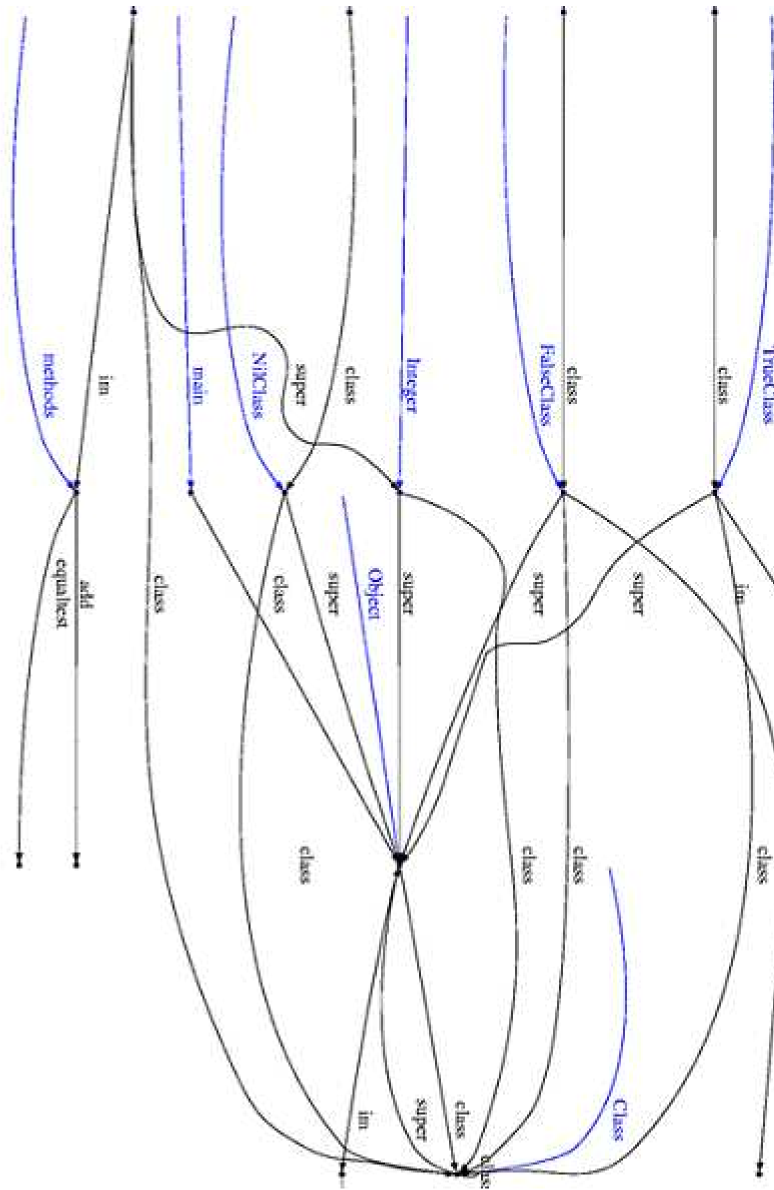


Figure 5.2 Part of the initial program state

5.4 Instruction Parsing

The Ruby programs are read instruction by instruction.[3] In order to make Ruby instructions executable in program algebra, a delimiter set and a keyword table are designed.

The delimiter set contains the delimiters used to separate the keywords from instructions. All allowed delimiters are listed below between braces.

```
{ " " "=" "." "[" "]" "+" "-" ">" "<" "&" "|" "("
")" "@" "\"" }
```

As mentioned before, a keyword table containing projection functions and keywords is built. According to the keyword received during execution, the keyword table can dispatch the corresponding projection functions.

Pseudo code:

$\varphi_{keyword-fetch}$ =

```
L1;
first strA h;
-h==delimiter{;
                  append instr h;
                  delfirst strA;
                  ##L1;
              }{;
                  return keyword;
              }
```

$\varphi_{keyword-fetch}$ is an algorithm used to return the keyword. We can see from the pseudo code that the Ruby instructions are saved as a string in `strA`. We keep on removing the first character from `strA` and appending it to the end of the string `instr` until a key word is found.

[3] How to read Ruby programs is explained in Section 11.1: Compilation versus projection

5.5 Implementation of the projections in IPL

5.5.1 Projections implemented in RC1

As mentioned in [9], RC1 has simple control instructions, local variables, classes, instance methods and instance variables. In this section, we will explain how to use the projection functions defined in RC1 in our parser program.

- **Class definition:**
  `class C [<H]`
  `C` is an identifier of a class, starting with a capital letter. `H` is its superclass. If there is no `H`, the default super class is `Object`.

Using the $\varphi_{keyword-fetch}$ method, the Ruby instruction: `class MutablePair<Pair` is read character by character until we find a keyword "`class`".

```
keyword

class     MutablePair   <   Pair
```

Then we can retrieve in the keyword table the corresponding method containing the proper projection functions. According to the keyword table, the method "`ClassDefinition`" will be called. The projection functions inside the braces will be applied.

```
{C.+class=Class;
C.+super=H;
StackFrame.+next=new;
StackFrame.next.+back;
StackFrame.next.back=StackFrame;
StackFrame=StackFrame.next;
StackFrame.+lv=new;
StackFrame.+self=C;}
```

We continue to read the Ruby instruction until a delimiter "<" is returned. This delimiter is used to separate the subclass from the superclass. In this example, the subclass is "`MutablePair`" and the superclass is "`Pair`". As a result, the parameters `(C,H)` will be replaced by `(MutablePair,Pair)`.

C H

MutablePair < Pair

The final method call in our implementation will be `ClassDefinition(MutablePair,Pair)`.

- **Method definition:**
  `def m(p1,p2,...,pn)`
  `m` is an identifier of a method, starting with a small letter. And `p1` to `pn` are arguments.

  Using the $\varphi_{keyword-fetch}$ method, the Ruby instruction: `def initialize(a)` will be

analyzed as below.

keyword

def initialize (a)

According to the keyword received, the keyword table can dispatch the corresponding projection functions. In this example the keyword is "`def`" and the method "`MethodDefinition`" will be called.

Keywords	Projection
def	MethodDefinition

m p1

initialize(a)

Then we continue to read the Ruby instruction until a delimiter "`(`" is returned. This delimiter is used to separate the method names from the parameters. In this example, the method name is "`initialize`" and the parameter is "a". Then projection functions inside the braces will be applied.

```
MethodDefinition(initialize){
self=StackFrame.self;
+self==main{;cl=Object;}{;cl=self;};
-cl/im{;cl.+im=new;};
methods=cl.im;
```

```
method=new;
methods.+initialize;
methods.initialize=method;
method.+label:str="MutablePair:initialize";
##L"MutablePair:initialize:skip";

L"MutablePair:initialize";
arguments=new;arguments.+a=agr1;
StackFrame.+lv=arguments;StackFrame.+self=self;
R;

L"MutablePair:initialize:skip";}
```

- **Return instruction:**

  `return expr`[4]

  A special statement only occuring in method definitions.

  Using the $\varphi_{keyword-fetch}$ method, the Ruby instruction: `return @e` will be analyzed as below.

```
keyword   expr

return    @e
```

In this example, the keyword is "`return`" and according to the keyword table, the method `Return(@e)` will be called. Then the projection functions inside the braces will be applied.

```
Return(@e){
self=StackFrame.lv;
-self/iv{;result=nil;}{;instancevars=self.iv;
-instancevars/e{;result=nil;}{;result=instancevars.e;};}
```

- **Method call:**
  - `m(exp1,exp2,...,expn)` (*expr*)[5]

    The method call is sent to the object referred to by `self`. Expressions `exp1` to

---

[4] `expr` represents an expression. Some expressions are instructions which- when executed – yield an object. After an expression is evaluated, the result is put in a dedicated focus `result`. Expressions can be used as arguments for the `return` statements.

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

`expn` are the actual arguments.

- `exp0.m(exp1,exp2,...,expn)` `(expr)`
  The prefix `exp0` supplies the object to which the method call is sent.

- `expr=exp0.m(exp1,exp2,...,expn)` `(expr)`
  A *value-returning* method contains at least one assignment of the form `that=y`[6]. It will at last be assigned to the `expr`.

Using the $\varphi_{keyword-fetch}$ method, the Ruby instruction: `a=Object.new()` will be analyzed as below.

```
expr      expr0        m

a    =    Object   .   new ()
```

According to the keyword table, the method `MethodCall(initialize)`[7] will be called.

- **Assignment:**
  - Local variables
    `x=expr`
    The assignment of an arbitrary expression to the local variable `x`.

  - Instance variables
    `@x=expr`
    The assignment of an arbitrary expression to instance variable `@x`.

Using the $\varphi_{keyword-fetch}$ method, the Ruby instruction: `@e0=a` will be analyzed as below.

```
@x   keyword    expr

@e0      =         a
```

According to the keyword table, the method `Assignment(@e0,a)` will be called.

[6] On a *value-returning* method we suppose that the value to be returned is saved in the focus `that`.
[7] `initialize` is a special method in Ruby programs. When you call Object.new to create a new object, Ruby creates an uninitialized object and then calls that object's `initialize` method.

Furthermore, the projection functions inside the braces will be applied.

```
Assignment(@e0,a){
localvars=StackFrame.lv;
-localvars/a;!;result=localvars.a;
self=StackFrame.self;
-self/iv{;self.+iv;aux=new;self.iv=aux;}
instancevars=self.iv;
instancevars.+e0;
instancevars.e0=result;}
```

- **`If statement:`**

  `if expr; u1;...;uk;[else; uk+1;...;ul;]end`

  The expression will be evaluated first. If it returns true, then the first section after an `if` statement is executed. If there is an `else`-section, that section is executed when the `if`-section is not.

Using the $\varphi_{keyword-fetch}$ method, the Ruby instruction:

```
if fir.first()==a;fir.switch();recursiveFirst(fir);end
```

will be analyzed as below.

```
keyword    expr

if          fir.first()   ==    a
```

In this example, the keyword is "`if`" and the expression for evaluation is "`fir.first()==a`". According to the keyword table, the method `If(fir.first()==a)` will be called. Then the projection functions inside the braces will be applied.

```
{ψ(fir.first()==a);
-result==false{;ψ(fir.switch());ψ(recursiveFirst(fir));}}
```

A complete RC1 example is shown below.

```
RubyProgram="
def recursiveFirst(pair);
    fir=pair;
    if fir.first()==a;
        fir.switch();
        recursiveFirst(fir);
    end;
 end;
 class Pair;
    def initialize(a,b);
        @e0=a;
        @e1=b;
    end;
    def first();
        return @e0;
    end;
    def second();
        return @e1;
    end;
    def switch();
        temp=@e0;
        @e0=@e1;
        @e1=temp;
    end;
 end;
 a=Object.new();
 b=Object.new();
 pair1=Pair.new(a,b);
 recursiveFirst(pair1)";
```

Figure 5.3 depicts part of the final state of executing the RC1 example. The complete fluid is available in Appendix A, RC1.gif.

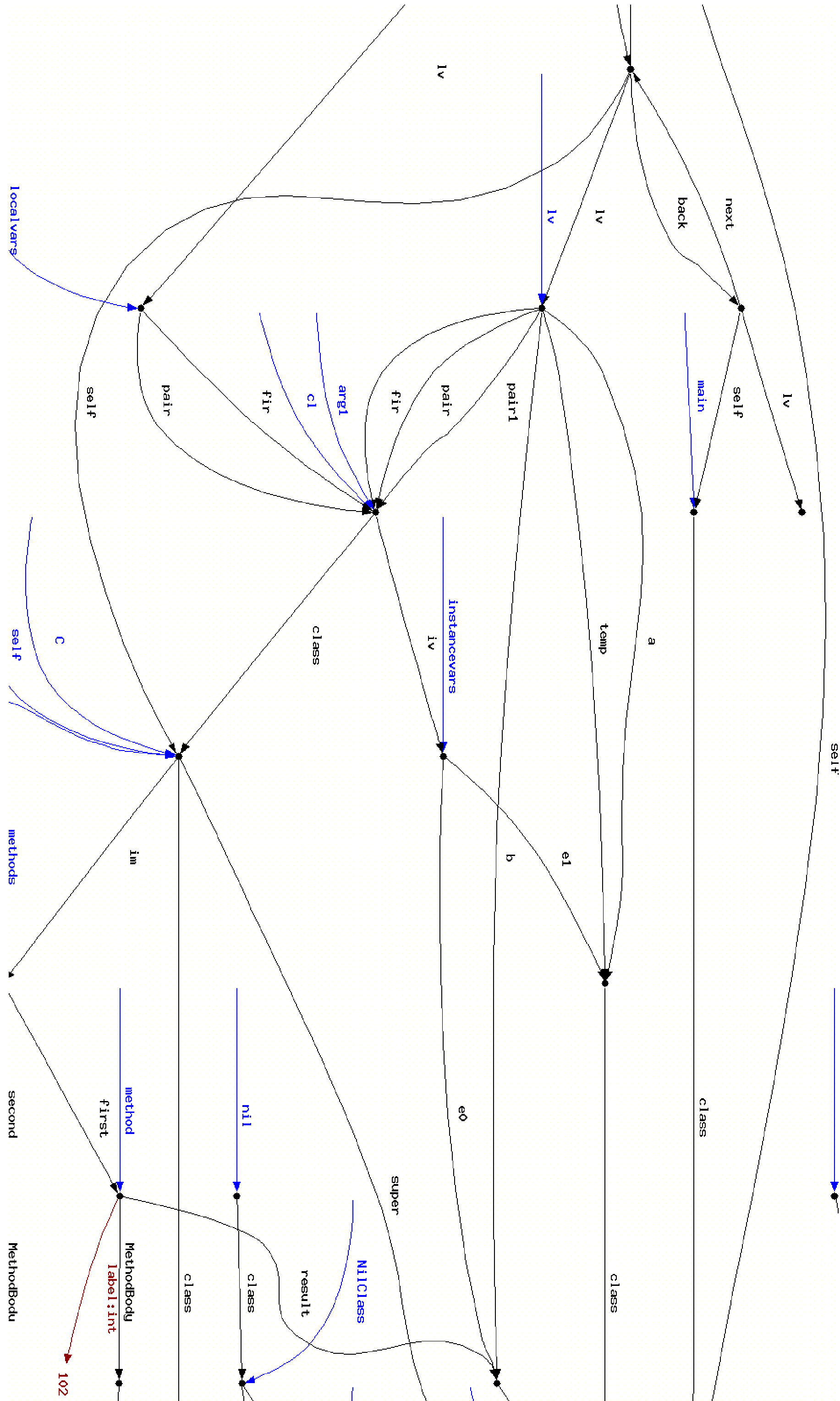


Figure 5.3 Part of the final state of executing the RC1 example

5.5.2 Projections implemented in RC2

RC2 changed the projection of RC1 by adapting the method call instruction to singleton methods. 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. Singleton methods are added to the field `sm`, while instance methods are added to the field `im`.

- **Singleton method:**
  `def expr.m(p1,...,pn);u1;...;uk;end`
  The expression, when evaluated, supplies the object on which the method `m` will be defined.

Using the $\varphi_{keyword-fetch}$ method, the Ruby instruction: `def Number.zero( )` will be analyzed as below.

```
keyword   expr             m
def         Number    .    zero ()
```

According to the keyword table, the method `MethodDefinition(Number.zero)` will be called. Then the projection functions inside the braces will be applied.

```
MethodDefinition(Number.zero){
φ(Number);x=result;
-x/sm{;x.+sm=new;};
methods=x.sm;
method=new;
methods.+zero;
methods.zero=method;
method.+label:str="Number:zero";
##L"Number:zero:skip";

L"Number:zero";
arguments=new; arguments.+a=agr1;
StackFrame.+lv=arguments; StackFrame.+self=self;
R;

L"Number:zero:skip";}
```

First, the expression `Number` should be evaluated. It supplies the object on which the method `zero` will be added.

Here φ `(Number)=`

```
localvars=localvars.lv;
-localvars/Number;!;result=localvars.Number;
```

5.5.3 Projections implemented in RC3

RC2 has two types of variables: local variables and instance variables. RC3 adds class variables. A class variable is a dedicated variable that can be manipulated inside both class methods and instance methods. In addition, it is inherited by extending classes. As a result, there is no need to write access methods in extending classes.

- **Class variables assignment**

  `@@x=expr`

  The assignment of an arbitrary expression `expr` to a class variable `@@x`.

Using the $\varphi_{keyword-fetch}$ method, the Ruby instruction: `@@zero=self` will be analyzed as below.

@@x	keyword	expr
@@zero	=	self

According to the keyword table, the method `Assignment(@@zero,self)` will be called. Furthermore, the projection functions inside the braces will be applied.

```
Assignment(@@zero,self){
φ(self);p=result;

φ_find-class ;

φ_search-supers (cl,cv,zero);

+found==false{;cl.+cv;classvars=new;cl.cv=classvars;
}{;classvars=br;};
classvars.+zero;
classvars.zero=p;}
```

Class variables are stored in a separate branch of the class object: the `cv` field. It points to an object that contains all the class variables. Class variables have a different syntax (`@@x`) to distinguish them from instance variables (`@x`).

Firstly, the expression `self` should be evaluated.

```
φ(self)=
localvars=localvars.lv;
-localvars/self;!;result=localvars.self;
```

Then a macro $\varphi_{find-class}$ helps to find the class belonging to `self`.

```
φ_find-class =
self=StackFrame.self;
cl=self.class;
+cl==Class{;cl=self;};
```

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

```
φ_search-supers(cl,cv,zero)=
loop=true;
found=false;
sp=cl;
L"search:supers";
+loop==true{;
    +sp/cv{;br=sp.cv;
      +br/zero{;loop=false; found=true; res=br.zero;};};
    +sp/super{;sp=sp.super;}{;loop=false;};
    ##L"search:supers";};
```

5.5.4 Projections implemented in RC4

RC4 extends RC3 with 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. Integers are created by reading in their numerical value. To simplify matters, all integers are instances of Fixnum, which is subclass of the Integer class in RC4. The Fixnum class defines two instance methods + and ==.

- **Equality test expression:**
  `exp0==exp1    (expr)`

  Using the $\varphi_{keyword-fetch}$ method, the Ruby instruction: `x==y` will be analyzed as below.

```
exp0   keyword   exp1

x          = =          y
```

According to the keyword table, the method `Equalitytest(exp0,exp1)` will be called. Then the projection functions inside the braces will be applied.

```
Equalitytest(exp0,exp1){
self=exp0;arg1=exp1;
+self.v==arg1.v{;
                    result.+boolvalue=True;
                    result.+class=TrueClass;
                }{;
                    result.+boolvalue=False;
                    result.+class=FalseClass;};
                }
```

- **Addition expression:**
  `exp0+exp1    (expr)`

  Using the $\varphi_{keyword-fetch}$ method, the Ruby instruction: x+y will be analyzed as below.

```
exp0   keyword   exp1

x            +          y
```

According to the keyword table, the method `Add(exp0,exp1)` will be called. Then the projection functions inside the braces will be applied.

```
Add(exp0,exp1){
self=exp0;arg1=exp1;
result=new;result.+class=Fixnum;result.+v:int;
integer-add(self.v,arg1.v,result.v);
}
```

5.6 Easy and difficult aspects of this implementation

All the projection functions in RC1 to RC4 are given in [9]. So it is easy for us to use those functions in our programs. The only work left is to program them together in order to make them executable in program algebra.

Next to the easy aspect, a lot of difficulties arose in our implementation. The most difficult one is program automation. For example, the parameters used inside the projection functions are fixed in the definition. But in our real programming work, they depend on the variables received. So the following problem arises, how can we accept different parameters in the same projection? One way to solve this problem is to use methods. Just as we mentioned in the keyword parsing section, each projection function is mapped to a unique method name. This method can accept variables as parameters.

In addition, an extension of the PGA toolset is necessary. Because 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 to be called is only known at run time when the class of the calling object is determined. In order to avoid confusion, a string label is assigned to each method. This string is the combination of a class name and a method name. This part is described in Chapter 3.

Finally, the subset of Ruby is extended gradually. Our implementation is based on those projections, so that we should add new features step by step. Some subsets will not change any instructions of the old version. Some subsets will override existing methods. Being dependent on the previous subset, we cannot project from RC4 to RC1. As a result, we have to change the parser programs again and again to adapt the new features.

Chapter 6

Ruby Core Four Plus (RC4+)

RC4 has basic data type: integer, which is represented as a class in Ruby. Instance methods *equality test* (`==`) and *addition* (`+`) have been projected in [9]. In this chapter, some other instance methods on integers are extended: *monus*[7] (`-`), *greater than test* (`>`) and *less than test* (`<`). An entire example using these three instance methods is given at the end of this chapter.

6.1 Extending the integers

In RC4+, all integers are instances of `Fixnum` as in RC4. The `Fixnum` class is a subclass of the `Integer` class. `Fixnum` numbers have fixed sizes, in contrast to `Bignum` numbers, which can be arbitrary large. Several instance methods are introduced here as supplements to the projection of RC4. The first one: *monus* (`-`), for example:

```
x=13;
x=x-1;
```

This example is straightforward and the result of this example is `x=12`. Another example,

```
x=13;
y=14;
x=x-y;
```

Here `x` is less than `y`, and according to the arithmetic rules of *monus*, `0` is returned to `x` as the result.

Another two important operations on integers are the comparison instance methods `>` and `<`. For example:

```
x=13;
y=14;
x>y;
x<y;
```

[7] Because integers in PGA are non-negative numbers, our projection is about monus instead of minus, which means if the minuend is less than the subtrahend, 0 is returned.

In this example, x is less than y, and according to the integer arithmetic rules, the result of the *greater than test* (>) is false, while that of the *less than test* (<) is true.

6.1.1 Example

This example is about a class Range, denoting an integer range, such as [1,5]. The method initialize is called to create a new instance of Range. This method initializes the new Range instance with two integer elements in instance variables @e0 and @e1. After an instance of Range is created, the methods max and min retrieve the bigger and the smaller elements, which are the maximal and the minimal values of the range. If the values of the two elements in a Range instance are equal, then the return values of max and min methods are the same.

```
class Range;
  def initialize(a, b);
        @e0=a;
        @e1=b;
  end;
  def max();
        if @e0>@e1;
              return @e0;
        else;
              return @e1;
        end;
  end;
  def min();
        if @e1<@e0;
              return @e1;
        else;
              return @e0;
        end;
  end;
end;
```

6.1.2 Syntax

The syntax for RC4+ is almost the same as for RC4[8].

- An *int* expression consists of a sequence of digits.
  int (*expr*)[9]

6.1.3 Projection

The projection for RC4+ extends that of RC4 with three instance methods `-`, `>` and `<`.

- $\varphi_{im-begin}$ is a macro that was defined in [9]. It begins a definition of an instance method *m* for class *C*. After this macro the method body is given and the definition of this method *m* is ended with another macro φ_{im-end}.

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

```
-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 close the definition of an instance method *m* for class *C* that begins with $\varphi_{im-begin}$.

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

```
R;
L"C:m:skip"
```

[8] Because there isn't any negative number in PGA, the syntax for RC4+ *int* is not preceded by a − sign, this is different from the syntax for RC4 in [9].
[9] An instruction, which is tagged with (*expr*) can be used in other instructions as an expression.

- rc2ipl denotes the projection function of the Ruby subset to PGA. It is defined in [9]. u1;…;uk denote the instructions of the Ruby program. The Integer class and the Fixnum class are defined before the projection of the Ruby program.
  rc2ipl(u1;…;uk)=

  $\varphi_{init-classes}$; $\varphi_{init-methods}$; $\varphi_{init-StackFrame}$; $\varphi_{init\text{-}integer}$; ψ_{main} (u1);…; ψ_{main} (uk)

- We extend the projection function of the integer initialization $\varphi_{init\text{-}integer}$ with three instance methods: - (*monus*), > (*greater than test*) and < (*less than test*). The projection code for setting up the Integer and Fixnum classes and the projection code of the instance methods == and + were given in [9]. Because true, false and value are reserved words in PGA, we use True, False and v instead.

In [9], integer-add is a request to the underlying system to perform the integer addition, it abbreviates:

```
result.v=self.v;incr result.v arg1.v;
```

Here integer-monus is a request to the underlying system to perform integer subtraction. The projection of *monus* is in the same style and integer-monus abbreviates:

```
result.v=self.v;decr result.v arg1.v;.
```

A decr instruction on integers, which is defined in MSP is used in this projection function. With the help of the decr instruction we can compare the values of the two operands. If the minuend is greater than or equal to the subtrahend, then the subtraction is implemented. Otherwise returns false. Thus by judging the Boolean value generated by the decr instruction, we can compare two integers. As to the *greater than test* (>), if the Boolean value generated by the decr instruction, which uses the first operand of the *greater than test* (>) as the minuend and the second one as the subtrahend, is false, then the result of the *greater than test* (>) is false also; otherwise its result is true. We project the *less than test* (<) in the same style.

$\varphi_{init\text{-}integer}$ =

```
Integer=new;Integer.+class=Class;
Integer.+super=Object;
Fixnum=new;Fixnum.+class=Class;
Fixnum.+super=Integer;
```

$\varphi_{im-begin}$ (Fixnum,==);

```
result=new;
+self.value==arg1.value{;
                              result.+class=TrueClass;
                              result.+boolvalue=True;
                         }{;
                              result.+class=FalseClass;
                              result.+boolvalue=false;
                         };
```

φ_{im-end} (Fixnum,==);

$\varphi_{im-begin}$ (Fixnum,+);

```
result=new;result.+class=Fixnum;result.+v:int;
integer-add(self.v,arg1.v,result.v);
```

φ_{im-end} (Fixnum,==);

$\varphi_{im-begin}$ (Fixnum,-);

```
result=new;result.+class=Fixnum;result.+v:int;
integer-monus(self.v,arg1.v,result.v);
```

φ_{im-end} (Fixnum,-);

$\varphi_{im-begin}$ (Fixnum,>);

```
i=self.v;j=arg1.v;
+decr i j{;
            result=new;
            result.+boolvalue=True;
            result.+class=TrueClass;
         }{;
            result=new;
            result.+boolvalue=False;
```

```
            result.+class=FalseClass;
        };
```

φ_{im-end} (Fixnum,>);

$\varphi_{im-begin}$ (Fixnum,<);

```
i=self.v;j=arg1.v; result=new;
+decr j i{;
            result.+boolvalue=True;
            result.+class=TrueClass;
        }{;
            result.+boolvalue=False;
            result.+class=FalseClass;
        };
```

φ_{im-end} (Fixnum,<)

- Like the *addition* operation and the *equality test* in [9], these three instance methods we defined here have infix notation. In the projection they are translated to method calls.
  ψ (*exp0*−*exp1*) = ψ (*exp0*.−(*exp1*))
  ψ (*exp0*>*exp1*) = ψ (*exp0*.>(*exp1*))
  ψ (*exp0*<*exp1*) = ψ (*exp0*.<(*exp1*))

6.2 Projection example of the integers

In the next example, four new `Fixnum` objects `a1`, `b1`, `a2` and `b2` are created in the first four instructions and the following two instructions put them in the instances of the `Range` class, `range1` and `range2` respectively. Then the next four instructions call the methods `max` and `min` to get the maximal and the minimal elements in `range1` and `range2`.

```
a1=12;a2=15;b1=2;b2=4;
range1=Range.new(a1,b1);
range2=Range.new(a2,b2);
max1=range1.max();
max2=range2.max();
min1=range1.min();
min2=ranger2.min();
```

Part of the state after executing this program is depicted in Figure 6.1. Because “+”, “==”and “-” are all reserved words in PGA, so they can’ t be used as focous names or field names. In Figure 6.1, the names of the integer instance methods “+”, “-” and “==” in RC4 and RC4+ are replaced with “add”, “sub” and “equal”. The whole fluid can be found in Appendix A, Integerplus.gif.

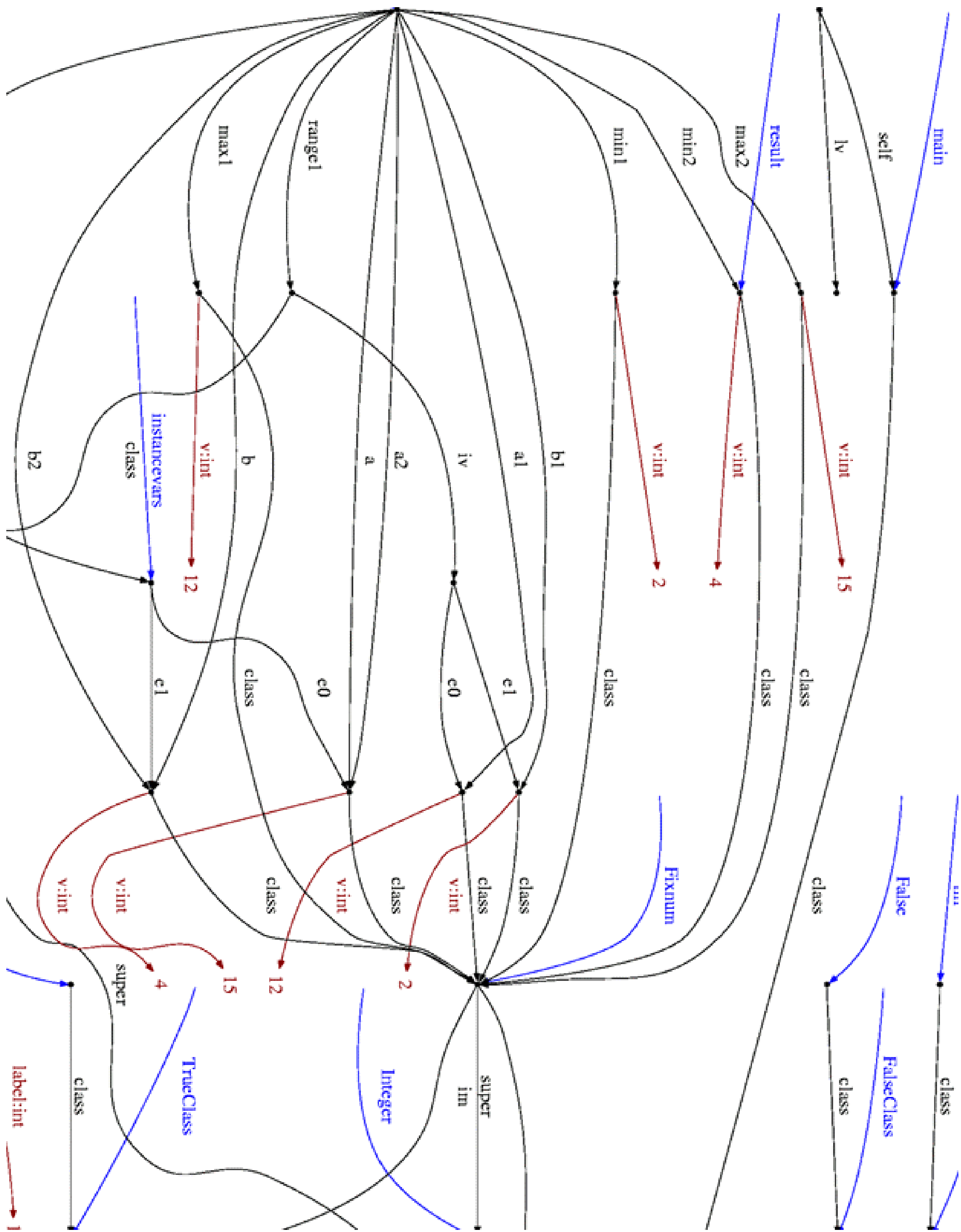


Figure 6.1 Program state after executing the example program of RC4+

Chapter 7

Ruby Core Five (RC5)

Up to RC4+, the basic data type integer has been projected. RC5 adds another two types: Boolean and string. Several important instance methods on strings and Booleans are also projected in this chapter.

Boolean expressions are mainly used as the conditions in conditional statements. Strings have over 75 standard methods. In this chapter, we'll look at some common instance methods on Booleans and strings.

7.1 Booleans

In Ruby, any value that is not `nil` or the constant `false` is `true`, including the number `0` and empty strings, arrays and hashes. Ruby supports all the standard Boolean operators: `&&`, `||` and `!`. There is also another syntactic form of these Boolean operators: `and`, `or` and `not`. The only difference between these two syntactic forms is the precedence. In this thesis, we ignore this difference, i.e. both `and` and `&&` are the same Boolean operators in our projection, the same to the other two pairs. For example, this is a part of a program,

```
if(x2>x1 && x2<x3) p=x2
```

In this example, if `x2` is the medium, then the program will print `x2`. If we use `and` operator here, the result is the same.

7.1.1 Syntax

The syntax for RC5 is extended with Boolean expressions, which are translated into objects of `TrueClass` or `FalseClass`[10].

- A *bool* consists of the value of a Boolean expression.
  bool (*expr*)

[10] Because we don't project any instance method for `NilClass` in RC5, all the Boolean expressions we have in this thesis are instances of `TrueClass` and `FalseClass`.

7.1.2 Projection

As there already exist `TrueClass` and `FalseClass` in the initialization phase in RC1 [9], all the Boolean expressions are objects of these classes, and a new `Boolean` class will not be created here in order to keep the original class hierarchy. The instance methods `and`, `or` and not are defined both for `TrueClass` and `FalseClass`. They are added to the class initialization phase as supplements.

As to `TrueClass`, only the test of the second operand for the operation `&&(and)` is needed because the first operand is always `true`; the result of the operation `||(or)` is always `true` no matter what the second operand is; and the operation `!(not)` always returns `false` as the opposite of `true`. As to `FalseClass`, the result of the operation `&&(and)` is always `false`; only the test of the second operand of the operation `||(or)` is needed because the first is always `false`; and the operation `!(not)` always returns `true` as the opposite of `false`. Because `&&`, `||` and `!` are reserved words in PGA, we use `and`, `or` and `not` in the definition of these methods only.

- This macro creates the pre-defined classes and constants. In this thesis this macro is slightly different from that in [9] to avoid using reserved words and to add the definitions of the instance methods on Booleans.

  $\varphi_{init-classes}$ =

```
Object=new;Class=new;Class.+super=Object;
TrueClass=new;TrueClass.+super=Object;
FalseClass=new;FalseClass.+super=Object;
NilClass=new;NilClass.+super=Object;
Object.+class=Class;Class.+class=Class;
TrueClass.+class=Class;FalseClass.+class=Class;
NilClass.+class=Class;

True=new;True.+class=TrueClass;
False=new;False.+class=FalseClass;
nil=new;nil.+class=NilClass;
main=new;main.+class=Object;
```

  $\varphi_{im-begin}$ `(TrueClass,and);`

```
result=new;
+arg1==true{;
                        result.+boolvalue=True;
                        result.+class=TrueClass;
                    }{;
                        result.+boolvalue=False;
                        result.+class=FalseClass;
                    };
```

φ_{im-end} (TrueClass,and);

$\varphi_{im-begin}$ (TrueClass,or);

```
result=new;result.+class=TrueClass;result.+boolvalue=True;
```

φ_{im-end} (TrueClass,or);

$\varphi_{im-begin}$ (TrueClass,not);

```
result=new;result.+class=FalseClass;
result.+boolvalue=False;
```

φ_{im-end} (TrueClass,not);

$\varphi_{im-begin}$ (FalseClass,and);

```
result=new;result.+class=FalseClass;
result.+boolvalue=False;
```

φ_{im-end} (FalseClass,and);

$\varphi_{im-begin}$ (FalseClass,or);

```
result=new;
+arg1==true{;
                        result.+boolvalue=True;
                        result.+class=TrueClass;
```

```
                        }{;
                            result.+boolvalue=False;
                            result.+class=FalseClass;
                        };
```

φ_{im-end} (FalseClass,or);

$\varphi_{im-begin}$ (FalseClass,not);

```
result=new;result.+class=TrueClass;result.+boolvalue=True;
```

φ_{im-end} (FalseClass,not)

- *bool* represents the value of a Boolean expression, a new object is created containing the Boolean value *bool*.
  ψ (*bool*) =
  result=*bool*;
  ```
  +result==true{;
                    result.+class=TrueClass;
                    result.+boolvalue=True;
              }{;
                    result.+class=FalseClass;
                    result.+boolvalue=False;
              }
  ```
- The and and or operations on Boolean expressions both have infix notations, which are translated into method calls.
  ψ (*exp0* and *exp1*) = ψ (*exp0*.and(*exp1*))
  ψ (*exp0* or *exp1*) = ψ (*exp0*.or(*exp1*))

7.2 Projection example of the Booleans

The Range class defined in RC4+ is used again here. The first seven instructions are the same as the example in Chapter 6. The last instruction returns a Boolean value as the result of the and operation.

```
a1=12;a2=15;b1=2;b2=4;
```

```
range1=Range.new(a1,b1);
range2=Range.new(a2,b2);
max1=range1.max();
max2=range2.max();
min1=range1.min();
min2=range2.min();
include=(max1>max2)and(min1<min2);
```

Figure 7.1 shows the class hierarchy of this example, including the classes `Class`, `Object`, `Fixnum`, `Integer`, `TrueClass`, `FalseClass` and `NilClass`. It is a basic part of the fluid that shows the result of executing this example in the PGA Toolset. This whole fluid can be found in Appendix A, Boolean.gif. Figure 7.2 is also a part of this fluid and it shows the state of the class `Range` after execution. Figure 7.3 shows the final state of `include` as well as class `Range`'s two instances, `range1` and `range2`.

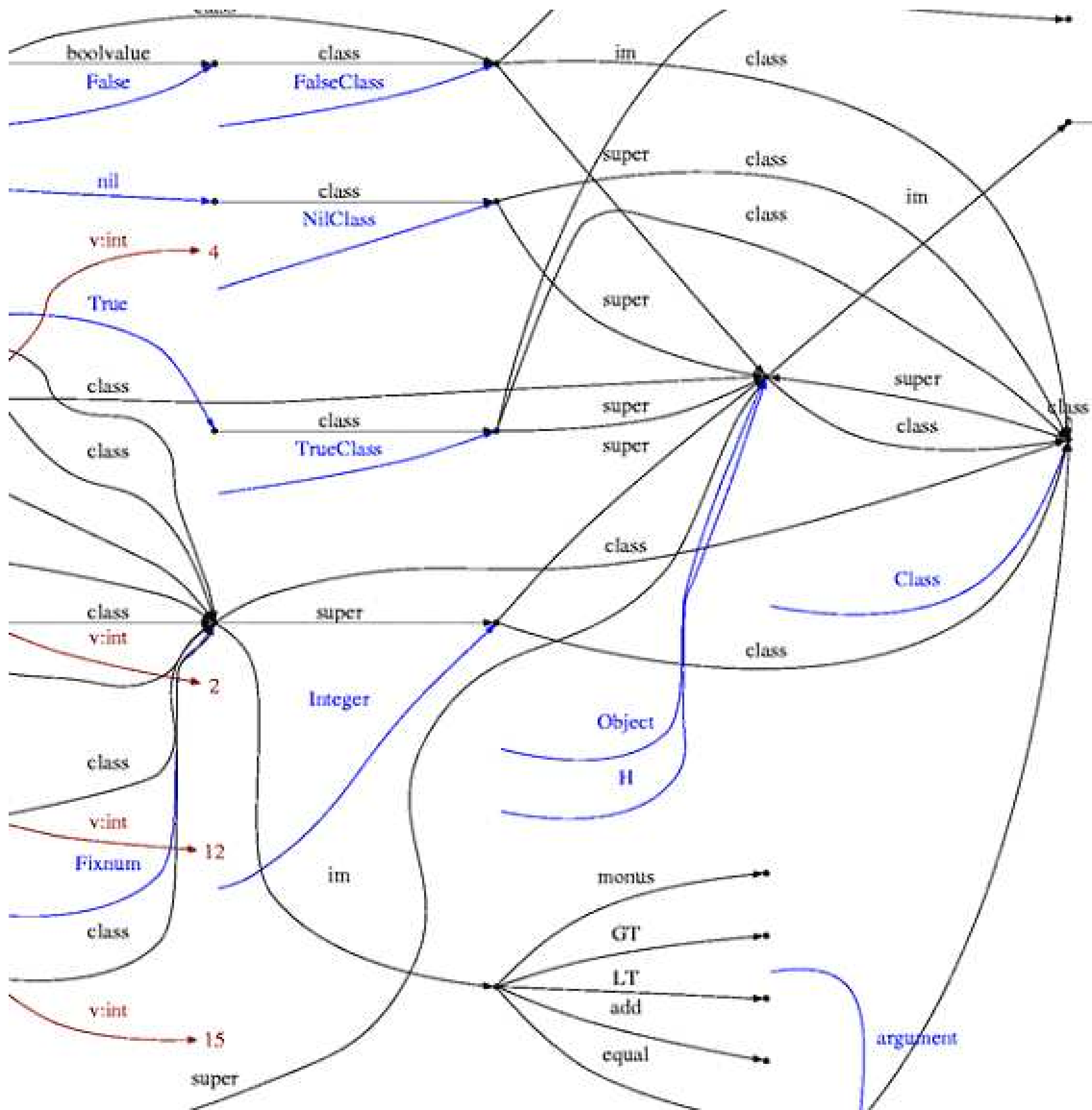


Figure 7.1 Classes hierarchy

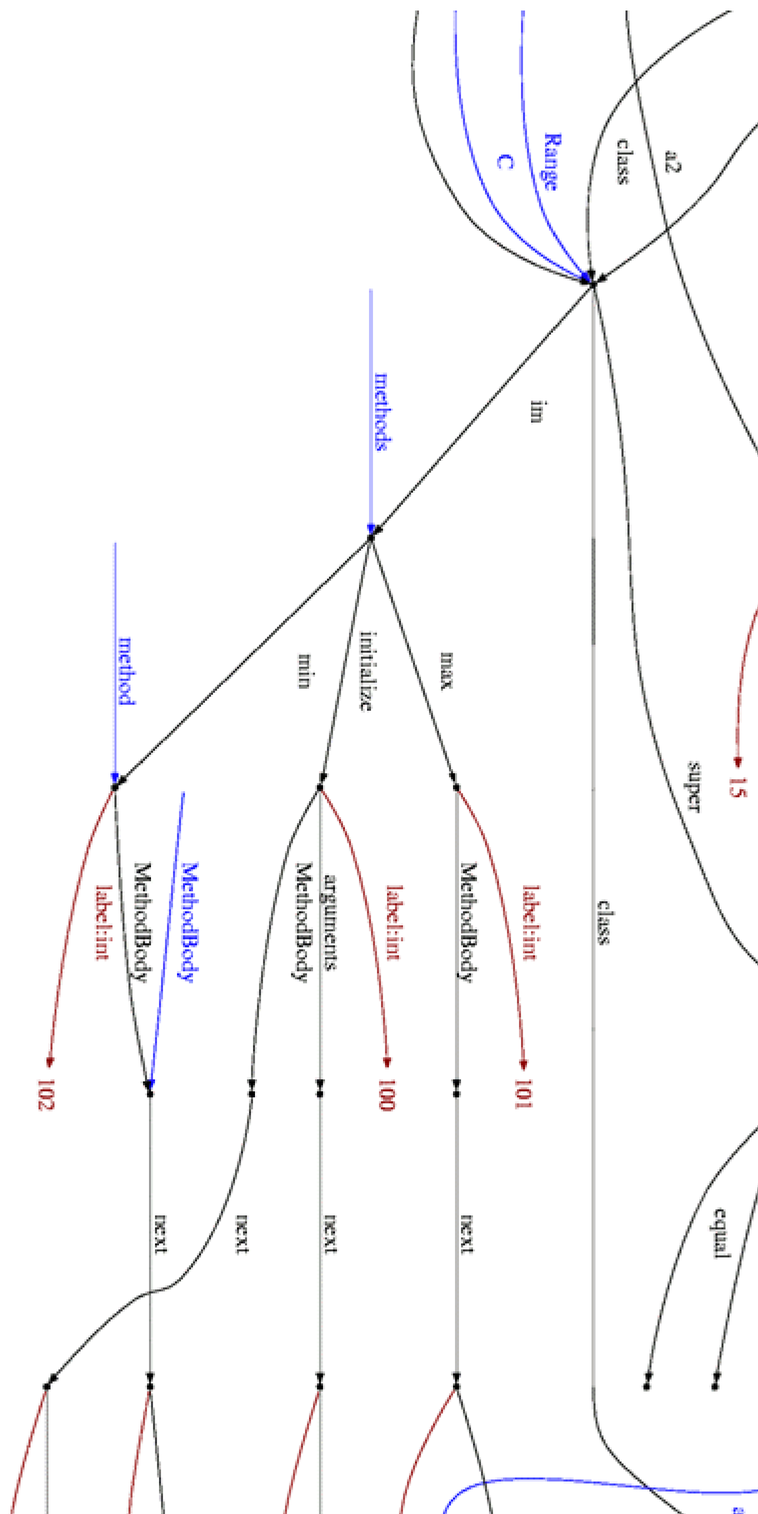


Figure 7.2 The final state of the class `Range`

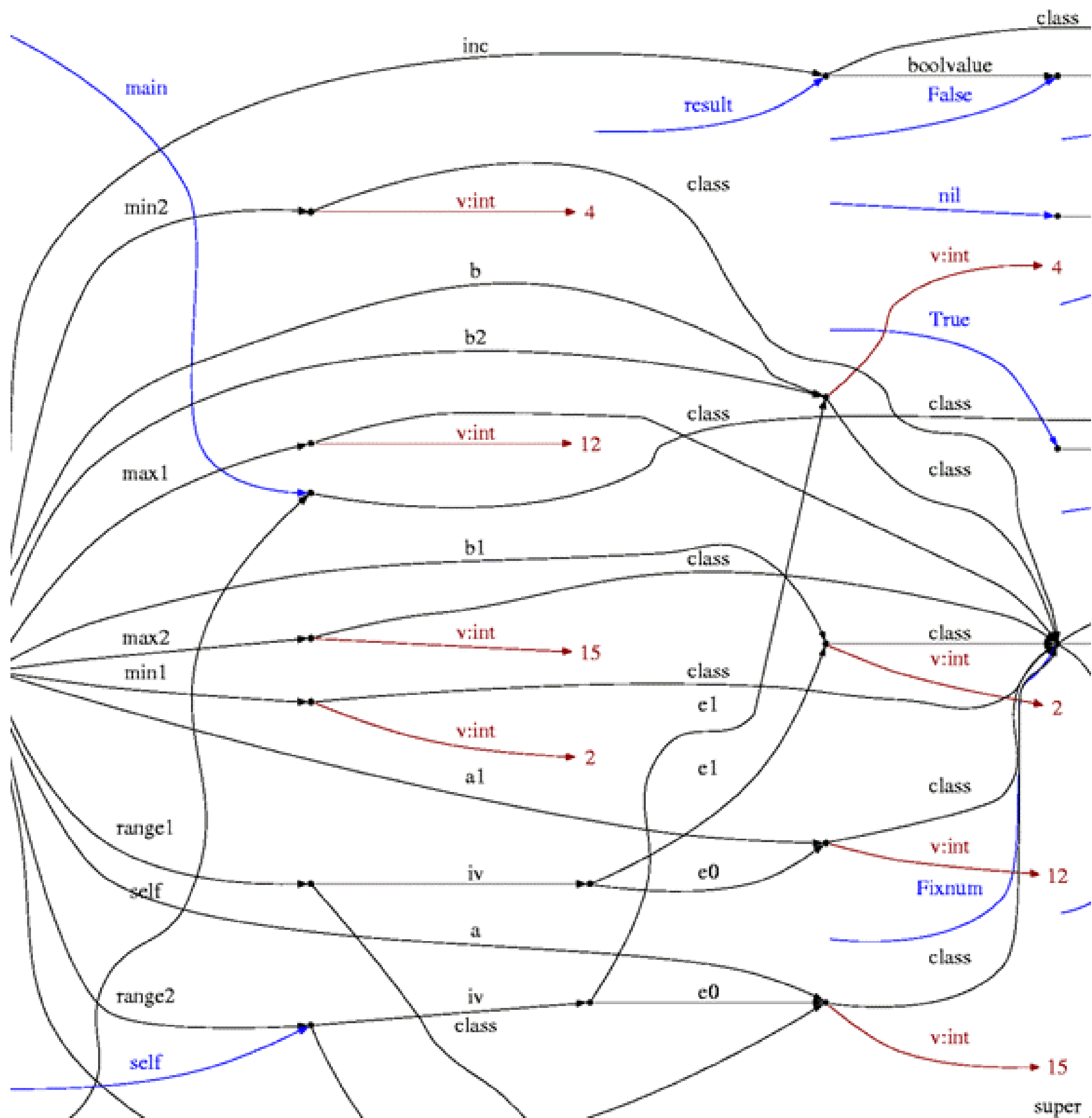


Figure 7.3 The states of `include`, `range1` and `range2`

7.3 Strings

In Ruby, strings are stored as objects of the class `String`. Strings are often created using string literals: sequences of characters between delimiters. Ruby supports four kinds of delimiters: single and double quotes, `%q` and `%Q`. We restrict to the strings in double quotes in this thesis. Although in Ruby strings in double quotes allow the substitutions and backslash notations, we

won't project this feature due to its complexity.

In Ruby, many operations are defined on strings. The first operation introduced here is concatenation. Let's have a look at this example:

```
str1="This is";
str2=" RC5";
str1=str1+str2;
```

The result of this operation is that, the content of `str1` is `This is RC5` and that of `str2` is `RC5` (begins with a space). For another example,

```
str1="There are two spaces ";
str2=" between them";
str1=str1+str2;
```

The result of `str1` is `There are two spaces  between them` because the last character of `str1` and the first character of `str2` both are spaces. Because strings are immutable in Ruby, the concatenation operation does not modify the content of the calling object `str1`. Instead it returns a new string object with content `There are two spaces between them`, and assigns it to `str1`. The concatenation can also be done in another way:

```
str1="This is" + " RC5";
```

Another important operation is the equality test. For example, in `"RC5" == "RC5"`, the result of this test is `true`. If the strings of each side don't have the same content, `false` will be returned.

Although there are also some other operations like repetition, extracting characters and extracting substrings, in RC5 only `concatenation` (+) and `equality test` (==) methods are implemented.

7.3.1 Example

Here is an example of a class `Books`. A `Books` instance is initialized with two string elements: `name` and `author` in the instance variables `@name` and `@author`. The methods `getname` and `getauthor` are used to access the elements. The method `displaybook` concatenates two elements of a `Books` instance with another string `" from "`, both the first and the last character of this string are spaces.

```
class Books;
        def initialize(n,a);
              @name=n;
              @author=a;
        end;
        def getname();
              return @name;
        end;
        def getauthor();
              return @author;
        end;
        def displaybook();
              return @name+"  from "+@author;
        end;
end;
```

7.3.2 Syntax

The syntax for RC5 is also extended with strings, which are translated into `String` objects.

- A *string* consists of a sequence of characters.
  string (*expr*)

7.3.3 Projection

- The `String` class initialization should be done before the user program is executed. The projection function of the program is changed accordingly.
  `rc2ipl(u1;……;uk)=`

  $\varphi_{init-classes}$; $\varphi_{init-methods}$; $\varphi_{init\text{-}integer}$; $\varphi_{init-string}$; $\varphi_{init-StackFrame}$;

  ψ_{main} `(u1);……;` ψ_{main} `(uk)`

- The `String` class defines two instance methods: `concatenation` and `equaltest` (equality test).

  $\varphi_{init-string}$ =

  `String=new;String.+class=Class;String.+super=Object;`

$\varphi_{im-begin}$ (String,equaltest);

```
result=new;result.+class;result.+boolvalue;
+self.content==arg1.content{;
                                result.boolvalue=True;
                                result.class=TrueClass;
                           }{;
                                result.boolvalue=False;
                                result.class=FalseClass;
                           };
```

φ_{im-end} (String,equaltest);

$\varphi_{im-begin}$ (String,concatenation);

```
result=new;result.+class=String;result.+content:str;
temp=new;temp.+content:str=self.content;
append temp.content arg1.content;
result.content=temp.content;
```

φ_{im-end} (String,concatenation)

- *string* represents a string, a new object of the class `String` is created containing the content of *string*.
  ψ (*string*)=
  `result=new;result.+class=String;result.+content:str=`*string*
- The concatenation operation has an infix notation, which is translated into a method call.
  ψ (*exp0* + *exp1*) = ψ (*exp0* .+(*exp1*))

7.4 Projection example of the strings

An instance of the class `Books`, which was defined in Section 7.3.1 is created in the following program.

```
s1="Making using of Ruby";
s2="Suresh Mahadevan";
book1=Books.new(s1,s2);
displaybook1=book1.displaybook()";
```

Part of the final state of this example is depicted in Figure 7.4. The whole fluid can be

found in Appendix A, String.gif.

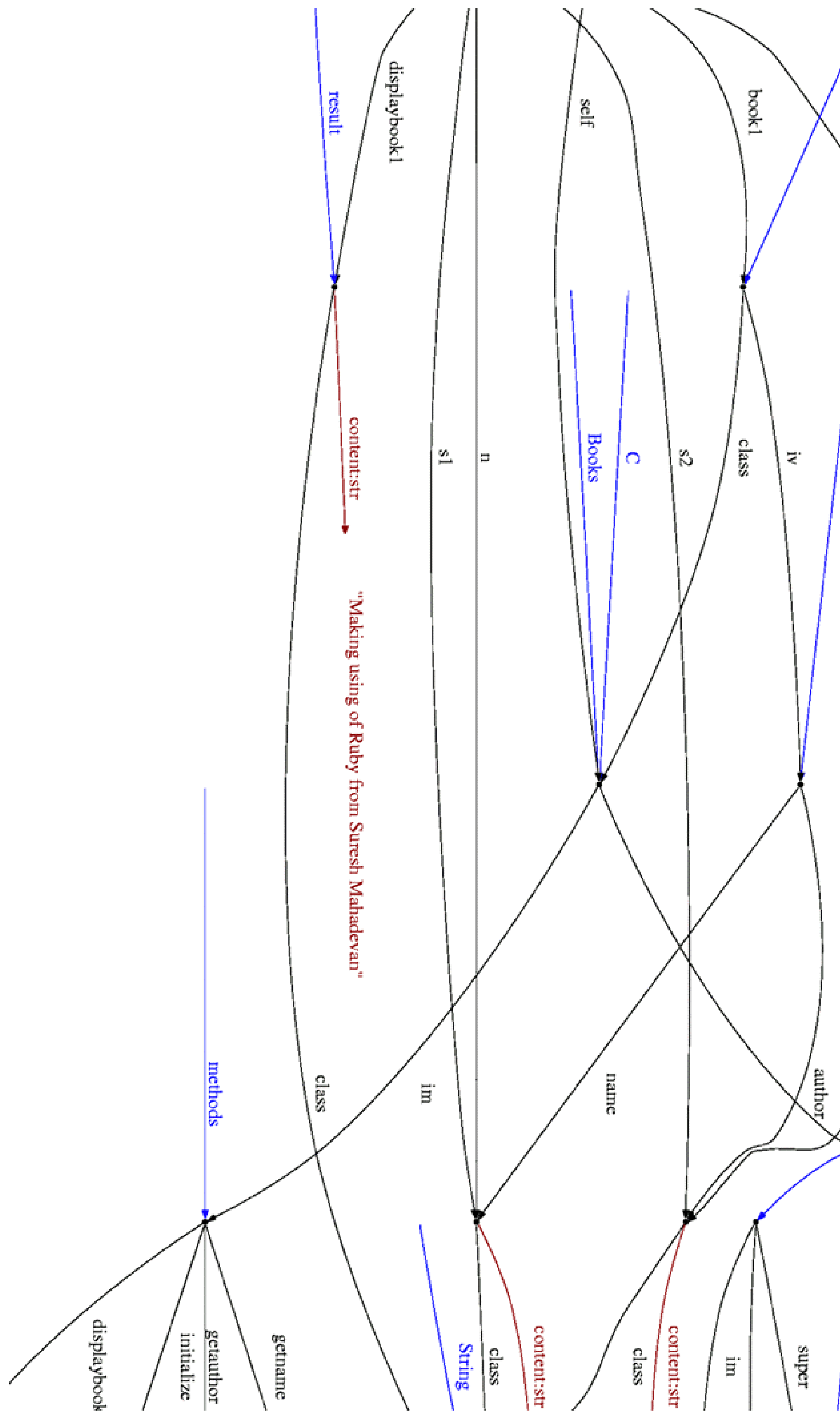


Figure 7.4 Part of the execution result

Chapter 8

Ruby Core Six (RC6)

Up to RC5, the data types that objects contain are integers, Booleans and strings. RC6 extends the previous projection with another two important type constructors: arrays and hashes. Both arrays and hashes in Ruby are indexed collections of objects. They are the commonly used data structures in Ruby. Both arrays and hashes in Ruby are indexed collections of objects. With arrays, the indices are non-negative integers while with hashes they can be any objects. Arrays and hashes can hold different types of objects: integers, strings and floating numbers, etc.

8.1 Arrays

Arrays are instances of the class `Array`, whose superclass is the `Object` class. An array is formed by comma-separated list of objects in square brackets “`[]`” and the index begins with zero. The elements of an array are arbitrary Ruby objects.

Arrays have several index methods: if an array is indexed with a single integer “`a[n]`”, it returns the object at that position or returns `nil` if nothing's there; if an array is indexed with a negative integer[11], it counts backward from the end. The index of the last element is -1. For example:

```
array1=[1,2,3,4,5];
array1[-2];
array1[3];
```

The results of these two expressions both are `4`. Arrays can also be indexed by a pair of numbers “`a[start,length]`”, which return a new array object holding `length` number of objects starting from the `start` position, such as

```
array1=[1,2,3,4,5];
array1[2,2];
```

This expression produces a new array `[3,4]`. If the `length` is negative, `nil` is returned.[12]

[11] The negative index will not be implemented here because in PGA integers are non-negative.
[12] We omit this kind of condition because there are no negative integers in PGA.

Arrays can be indexed by ranges also: "a[start..end]" or "a[start…end]". In ranges, start and end positions are separated by two or three periods. The difference between these is that the two-period form includes the end position while the three-period form does not. If the end is larger than the length of the array, in both cases it will be rounded to the length. If the start is out of the range of an array, both return nil. And if the start is larger than the end, both return an empty array. For example:

```
array1=[1,2,3,4,5];
array1[1..3];
array1[1…3];
array1[2..5];
array1[3..1];
```

These four expressions produce [2,3,4], [2,3], nil and an empty array.

Besides these index methods, one of the important instance methods of Array is the concatenation (+), which returns a new array with the result of concatenating the two arguments together. For example, [1,2,3]+[4,5,6] produces a new array [1,2,3,4,5,6].

Another important instance method is the equality test (==), which returns true only if the contents of two arrays are the same, which means both arrays have the same number of elements, while each element of one array is the same as the corresponding element of another array. [1,2,3]==[4,5,6] returns false while [1,2,3]==[1,2,3] returns true.

Calling the set intersection (&) method can return the common elements of both arrays. For example, [1,2,3]&[3,4,5] returns 3 and [1,2,3]&[1,3,5] returns [1,3].

Both size and length methods return the number of the elements in an array, [1,2,3,4,5].length returns 5 and [1,2,3,4,5,6].size returns 6.

There are also many other useful instance methods of arrays. In RC6, only the above methods are included.

8.1.1 Example

Here is a very simple example with arrays:

```
array1=[1,2,3];
array2=[1,2];
array1.size;
```

```
array1[2];
array1[0..1];
array1[0,2];
array1+array2;
array1==array2;
```

In this example, six instance methods of an `Array` object are used, which are `a[n]`, `a[start,length]`, `a[start..end]`, concatenation, equality test and `size`.[13]

8.1.2 Syntax

The syntax for RC6 is extended with arrays, which are translated into `Array` objects.

- An array consists of a list of elements within square brackets `[]` and these elements are separated with commas.
  `[`*exp1*`, . . . . . . ,` *expn*`]` (*expr*)

8.1.3 Projection

`Array` objects are implemented by a list of atoms connected with the `next` fields. Each atom in the list contains both a `v(alue)` and a `key` field as a pair. The `v` field is used to store the element of the array and the `key` field is to store the index, which starts from zero. Another atom represents the end of the array, which has only one field `Tail`, that is to say, if an atom in the list has only a `Tail` field, then it is the end of the array. If the array is empty, then there are two atoms in the list, the first is in the focus that is named after the name of the array, and it has a `next` field pointing to the second atom, which has only a `Tail` field, representing the end of the array.

- The initialization of class `Array` $\varphi_{init-array}$ should be defined before the Ruby program is projected.
  `rc2ipl(u1;……;uk)=`

  $\varphi_{init-classes}$; $\varphi_{init-methods}$; $\varphi_{init\text{-}integer}$; $\varphi_{init-string}$; $\varphi_{init-array}$; $\varphi_{init-StackFrame}$;

  ψ_{main} `(u1)`;……; ψ_{main} `(uk)`

[13] The running results of this example can be found in Appendix A.

- The `Array` class defines four instance methods for indexing: `return-nth` (`a[n]`), `returnlength` (`a[start,length]`), `returnrange2` (`a[start..end]`) and `returnrange3` (`a[start…end]`). It also defines other four instance methods: `concatenation`, `equaltest` (equality test), `intersection` (set intersection) and `size` (both for `size` and `length`). In order to avoid label clashes, we use string labels here.

  $\varphi_{init-array}$ =

  ```
  Array=new;Array.+class=Class;Array.+super=Object;
  ```

  $\varphi_{im-begin}$ `(Array,return-nth);`

  ```
  tail=new;tail.+Tail;result=new;result.+class=Array;
  result.+next=tail;current=result;

  L"Areturn-nth";
  -self/Tail{;
    +self.key==n{;
                  aux=new;aux.+v:int;
                  aux.+next;aux.v=self.v;
                  aux.next=tail;current.next=aux;
              }{;
                  result=nil;self=self.next;
                  ##L"Areturn-nth";
              };
  };
  ```

  φ_{im-end} `(Array,return-nth);`

  $\varphi_{im-begin}$ `(Array,returnlength(`*start*`,`*length*`));`

  ```
  tail=new;tail.+Tail;result=new;result.+class=Array;
  result.+next=tail;current=result;
  end=length;incr end start;decr end;

  L"Areturnlength:1";
  -self.key==start{;
  ```

```
 -self/Tail{;
                    self=self.next;##L”Areturnlength:1”;
          }{;
                    result=nil;##L”Areturnlength:3”;
          };
};
L”Areturnlength:2”;
-self.key==end{;
                    aux=new;aux.+v:int;aux.+next;aux.v=self.v;
                    aux.next=tail;current.next=aux;
                    current=current.next;self=self.next;
                    ##L”Areturnlength:2”;
               }{;

                    aux=new;aux.+v:int;aux.+next;
                    aux.v=self.v;aux.next=tail;
                    current.next=aux;current=current.next;
                    self=self.next;
               };
L”Areturnlength:3”;
```

φ_{im-end}(Array,returnlength($start$, $length$));

$\varphi_{im-begin}$(Array,returnrange($start$, end));

```
tail=new;tail.+Tail;result=new;result.+class=Array;
result.+next=tail;current=result;

L”Areturnrange:1”;
-self.key==start{;
   -self/Tail{;
                    self=self.next;##L”Areturnrange:1”;
            }{;
                    result=nil;##L”Areturnrange:2”;
            };
};
L”Areturnrange:3”;
```

```
-self.key==end{;
                  aux=new;aux.+v:int;aux.+next;
                  aux.v=self.v;aux.next=tail;
                  current.next=aux;current=current.next;
                  self=self.next;##L"Areturnrange:3";
              }{;
                  aux=new;aux.+v:int;aux.+next;
                  aux.v=self.v;aux.next=tail;
                  current.next=aux;current=current.next;
                  self=self.next;
              };
L"Areturnrange:2";
```

φ_{im-end}(Array,returnrange($start$, end));

$\varphi_{im-begin}$(Array,concatenation);

```
tail=new;tail.+Tail;result=new;result.+next=tail;
result.+class=Array;current=result;num=0;

L"Aconcatenation:1";
-self/Tail{;
                aux=new;aux.+key:int;aux.+v:int;aux.+next;
                aux.key=self.key;aux.v=self.v;aux.next=tail;
                result.next=aux;result=result.next;
                self=self.next;
                incr num;##L"Aconcatenation:1";
};

L"Aconcatenation:2";
-arg1/Tail{;
                aux=new;aux.+key:int;aux.+v:str;aux.+next;
                aux.key=arg1.key;incr
                aux.key num;aux.v=arg1.v;
                aux.next=tail;result.next=aux;
                result=result.next;
                arg1=arg1.next;##L"Aconcatenation:2";
```

```
};
result=current;
```

φ_{im-end}(Array,concatenation);

$\varphi_{im-begin}$(Array,equaltest);

```
result=new;result.+class=TrueClass;result.+boolvalue=True;
num1=self.size;num2=arg1.size;
-num1==num2{;result.boolvalue=False;##L"Aequaltest:1";};

L"Aequaltest:2";
-self/Tail{;
    L"Aequaltest:3";
        -arg1/Tail{;
             +arg1/invalid{;
                              result.boolvalue=False;
                              result.class=FalseClass;
                              arg1=arg1.next;
                              ##L"Aequaltest:3";
             };
             +self.v==arg1.v{;
                                   result.boolvalue=True;
                                   result.class=TrueClass;
                                   arg1.+invalid;
                                   self=self.next;
                                   arg1=current;
                                   ##L"Aequaltest:2";
                              }{;
                                   result.boolvalue=False;
                                   result.class=FalseClass;
                                   arg1=arg1.next;
                                   ##L"Aequaltest:3"; };
        };
};
L"Aequaltest:1";
```

φ_{im-end} (Array,equaltest);

$\varphi_{im-begin}$ (Array,intersection);

```
tail=new;tail.+Tail;result=new;result.+class=Array;
result.+next=tail;current=result;

L"Aintersection:1";
-self/Tail{;
   L"Aintersection:2";
      -arg1/Tail{;
             +self.v==arg1.v{;
                           aux=new;aux.+key:int;
                           aux.+v:int;aux.+next;
                           aux.key=self.key;
                           aux.v=self.v;aux.next=tail;
                           result.next=aux;
                           result=result.next;
                           self=self.next;arg1=current;
                           ##L"Aintersection:1";
                            }{;
                           arg1=arg1.next;
                           ##L"Aintersection:2"; };
             };
       self=self.next;##L"Aintersection:1";
};
result=current;
```

φ_{im-end} (Array,intersection);

$\varphi_{im-begin}$ (Array,size);

```
result=new;result.+class=Fixnum;result.+v:int;result.v=0;
L"size";
-self/Tail{;incr result.v;self=self.next;##L"size";};
```

φ_{im-end} (Array,size)

- [*exp1* , ... , *expn*] represents an array. This is a new object of class `Array`.
  ψ ([*exp1* , ... , *expn*]) =[3]

```
tail=new;tail.+Tail;result=new;result.+class=Array;
result.+next=tail;current=result;
aux=new;aux.+key:int;aux.+v:int;aux.+next;
aux.key=0;aux.v=exp1;aux.next=tail;result.next=aux;
result=result.next;

aux=new;aux.+key:int;aux.+v:int;aux.+next;
aux.key=1;aux.v=exp2;aux.next=tail;
result.next=aux;result=result.next;
... ...
aux=new;aux.+key:int;aux.+v:int;aux.+next;
aux.key=n-1;aux.v=expn;aux.next=tail;
result.next=aux;result=result.next;result=current
```

- Both the `concatenation` and `intersection` operations have infix notation, which are translated into method calls.
  ψ (*exp0* `concatenation` *exp1*) = ψ (*exp0*.`concatenation`(*exp1*))
  ψ (*exp0* `intersection` *exp1*) = ψ (*exp0*.`intersection`(*exp1*))

8.2 Projection example of the arrays

Part of the execution result of the example in section 8.1.1 is depicted in Figure 8.1. The whole fluid can be found in Appendix A, Array.gif.

[3] The elements of an array can be arbitrary Ruby objects. Here we use integers as an example to show how to build an array instance.

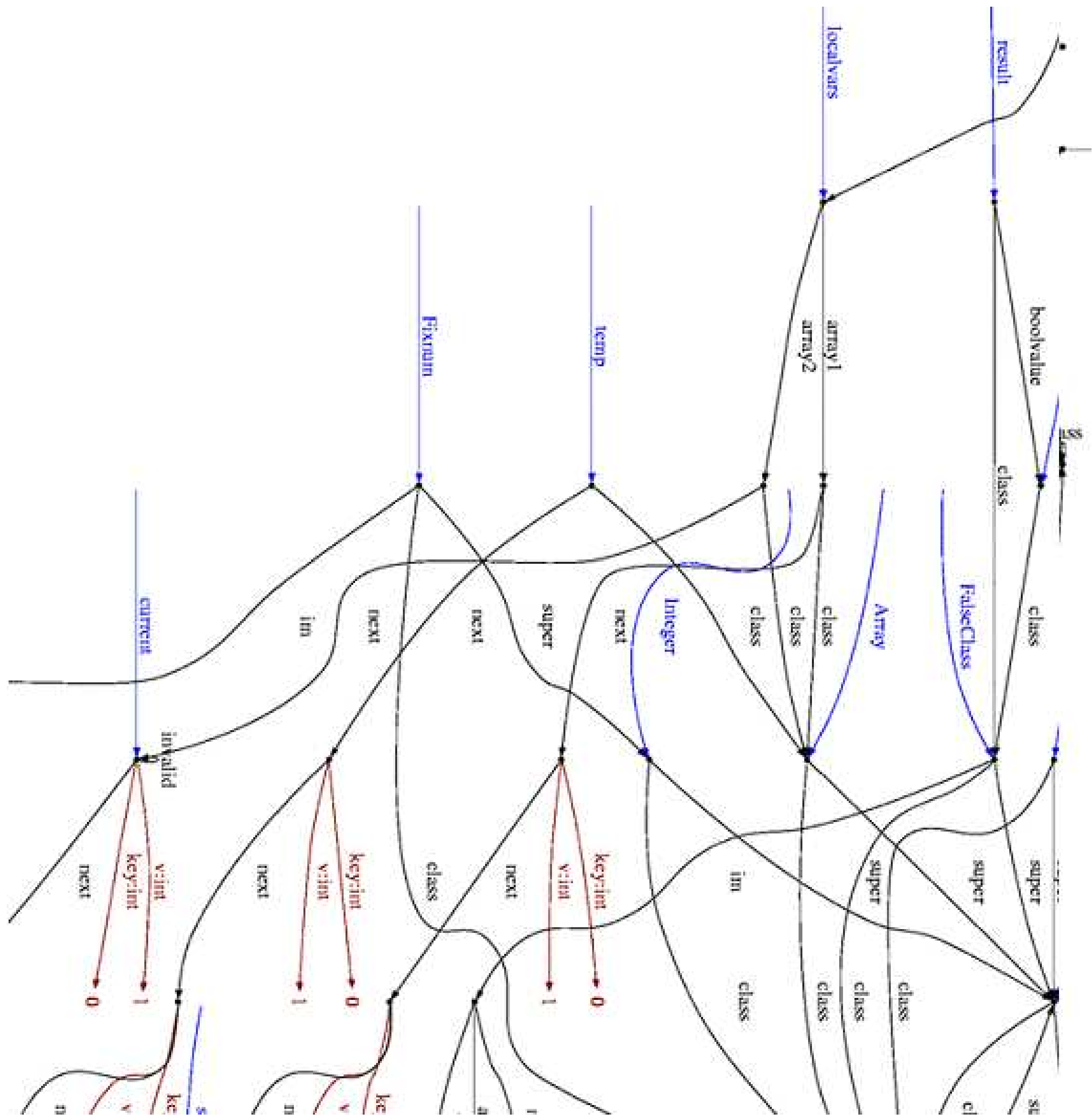


Figure 8.1 Part of the execution result

8.3 Hashes

Hashes are similar to arrays except that they can be indexed with arbitrary object types: strings, regular expressions and so on. Compared with arrays, this is a significant advantage of hashes. The values of a hash can be any object of any type. The list of key-value pairs is placed between braces "{}", with either a comma or the sequence => between the key and the value. For example:

```
students=
{"id"=>"000001","name"=>"Hesper Biggs","add"=>"61 Val R.D.",
"zipcode"=>"1111AA"}
```

However, hashes also have a disadvantage: their elements are not ordered as arrays and this makes that hashes cannot be used as a queue or a stack. A hash can be created in such a way: `Hash[key1,value1,…,keyn,valuen]`. For example,

`Hash[1,1,2,2]` produces a hash `{1=>1,2=>2}`.

The first important instance method of hashes in RC6 is the element reference `h[key]`, which returns the value associated with the `key`, if this `key` is not found in the hash, it returns `nil`. In the following example `h1={1=>2,2=>4,3=>6}`, `h1[2]` returns `4` and `h1[4]` returns `nil` because there isn't such an index.

Hashes also have the equality test (==). It tests whether two hashes have the same number of key-value pairs and if each key-value pair is the same as the corresponding one in the other hash. Let's look at an example:

```
h1={1=>2,2=>4,3=>6};
h2={2=>4,1=>2,3=>6};
h3={1=>2,2=>4,4=>8};
```

Then `h1==h2` returns `true` while `h2==h3` returns `false`.

Like arrays, both `size` and `length` methods return the number of key-value pairs in a hash. In the last example, `h1.size` returns `3` and `h2.length` also returns `3`.

Hashes also have methods `keys` and `values` to return a new array. The returning array of the `keys` method is combined with all the keys of the hash while of the `values` method is combined with all the values. For example, `h1.keys` returns `[1,2,3]` and `h1.values` returns `[2,4,6]`.

8.3.1 Example

Here is a simple example with hashes:

```
hash1={1=>2,2=>4,3=>6};
hash2={1=>2,2=>4,4=>8};
hash1.keys;
hash2.values;
hash1==hash2;
```

Two important instance methods: `keys` and `values` are used here. Both of them are not included in `Array` class.

8.3.2 Syntax

The syntax for RC6 is extended with hashes also, which are translated into `Hash` objects.

- A hash consists of a list of key-value pairs within brackets { }, separated by commas.
  { *key1=>value1 , ... , keyn=>valuen* } (*expr*)

8.3.3 Projection

Similar to the `Array` objects, each atom in a `Hash` instance has three fields: the `next` field is used to connect with the next atom, the `key` field stores the index and the `v(alue)` stores the value, except the first and the last ones. The first atom in focus that is named after the name of the hash has a next `field` pointing to the next atom, which represents the first element of the hash. The last atom of the list has only one field `Tail`.

- The `Hash` classes must be defined before the user program is executed. The projection function of the program is changed accordingly.
  `rc2ipl(u1;......;uk)=`

  $\varphi_{init-classes}$; $\varphi_{init-methods}$; $\varphi_{init\text{-}integer}$; $\varphi_{init-string}$; $\varphi_{init-array}$; $\varphi_{init-hash}$; $\varphi_{init-stackframe}$;

  ψ_{main} (u1);......; ψ_{main} (uk)

- The `Hash` class defines five instance methods `elementref` (element reference), `equaltest` (equality test), `size` (both for `size` and `length`), `keys` and `values`.

$\varphi_{init-hash}$ =

```
Hash=new;Hash.+class=Class;Hash.+super=Object;
```

$\varphi_{im-begin}$ (Hash,elementref(*key*));

```
tail=new;tail.+Tail;result=new;result.+class=Array;
result.+next=tail;current=result;

L"elementref";
-self/Tail{;
  +self.key==key{;
                    aux=new;aux.+v:int;aux.+next;aux.v=self.v;
                    aux.next=tail;current.next=aux;
                }{;
                    result=nil;self=self.next;##"elementref";
                };
};
```

φ_{im-end} (Hash,elementref(*key*));

$\varphi_{im-begin}$ (Hash,equaltest);

```
result=new;result.+class=TrueClass;result.+boolvalue=True;
num1=self.size;num2=arg1.size;

-num1==num2{;
                result.boolvalue=False;
                result.class=FalseClass;
                ##L"Hequaltest:3";
};

L"Hequaltest:4";
-self/Tail{;
        L"Hequaltest:5";
        -arg1/Tail{;
        +arg1/invalid{;
```

```
                              result.boolvalue=False;
                              result.class=FalseClass;
                              arg1=arg1.next;##L"Hequaltest:5";
                          };
          +self.v==arg1.v{;
                              result.boolvalue=True;
                              result.class=TrueClass;
                              arg1.+invalid;
                              self=self.next;
                              arg1=current;
                              ##L"Hequaltest:4";
                             }{;
                              result.boolvalue=False;
                              result.class=FalseClass;
                              arg1=arg1.next;
                              ##L"Hequaltest:5";
                             };
      };
};
L"Hequaltest:3";
```

φ_{im-end} (Hash,equaltest);

$\varphi_{im-begin}$ (Hash,size);

```
result=new;result.+class=Fixnum;result.+v:int;result.v=0;
L"Hsize";
-self/Tail{;incr result.v;self=self.next;##L"Hsize";};
```

φ_{im-end} (Hash,size)

$\varphi_{im-begin}$ (Hash,keys);

```
tail=new;tail.+Tail;keys=new;keys.+class=Array;
keys.+next=tail;i=0;result=keys;

L"Hkeys";
```

```
-self/Tail{;
            aux=new;aux.+key:int=i;aux.+v:int=self.key;
            aux.+next=tail;result.+next=aux;
            result=result.next;self=self.next;
            incr i;##L"Hkeys";
};
```

φ_{im-end} (Hash,keys);

$\varphi_{im-begin}$ (Hash,values);

```
tail=new;tail.+Tail;values=new;values.+class=Array;
keys.+next=tail;i=0;result=values;

L"Hvalues";
-self/Tail{;
            aux=new;aux.+key:int=i;aux.+v:int=self.v;
            aux.+next=tail;result.+next=aux;
            result=result.next;self=self.next;
            incr i;##L"Hvalues";
};
```

φ_{im-end} (Hash,values)

- {*key1*=>*value1*, ..., *keyn*=>*valuen* } represents a hash. This is a new object of the class Hash.
  ψ ({ *key1*=>*value1*, ..., *keyn*=>*valuen*})=[4]
  tail=new;tail.+Tail;result=new;result.+class=Hash;
  result.+next=tail;current=result;
  aux=new;aux.+key:int;aux.+v:int;aux.+next;
  aux.key= *key1*;aux.v= *value1*;aux.next=tail;result.next=aux;
  result=result.next;
  aux=new;aux.+key:int;aux.+v:int;aux.+next;
  aux.key= *key2*;aux.v= *value2*;aux.next=tail;result.next=aux;
  result=result.next;

[4] The indexes and elements of a Hash can be any Ruby objects. But here we restrict them to integers as an example for simplicity.

```
... ...
aux=new;aux.+key:int;aux.+v:int;aux.+next;
aux.key= keyn;aux.v= valuen;aux.next=tail;result.next=aux;
result=result.next;result=current
```

8.4 Projection example of the hashes

Part of the execution result of the example in section 8.3.1 is depicted in Figure 8.2 and Figure 8.3. The whole fluid can be found in Appendix A, Hash.gif.

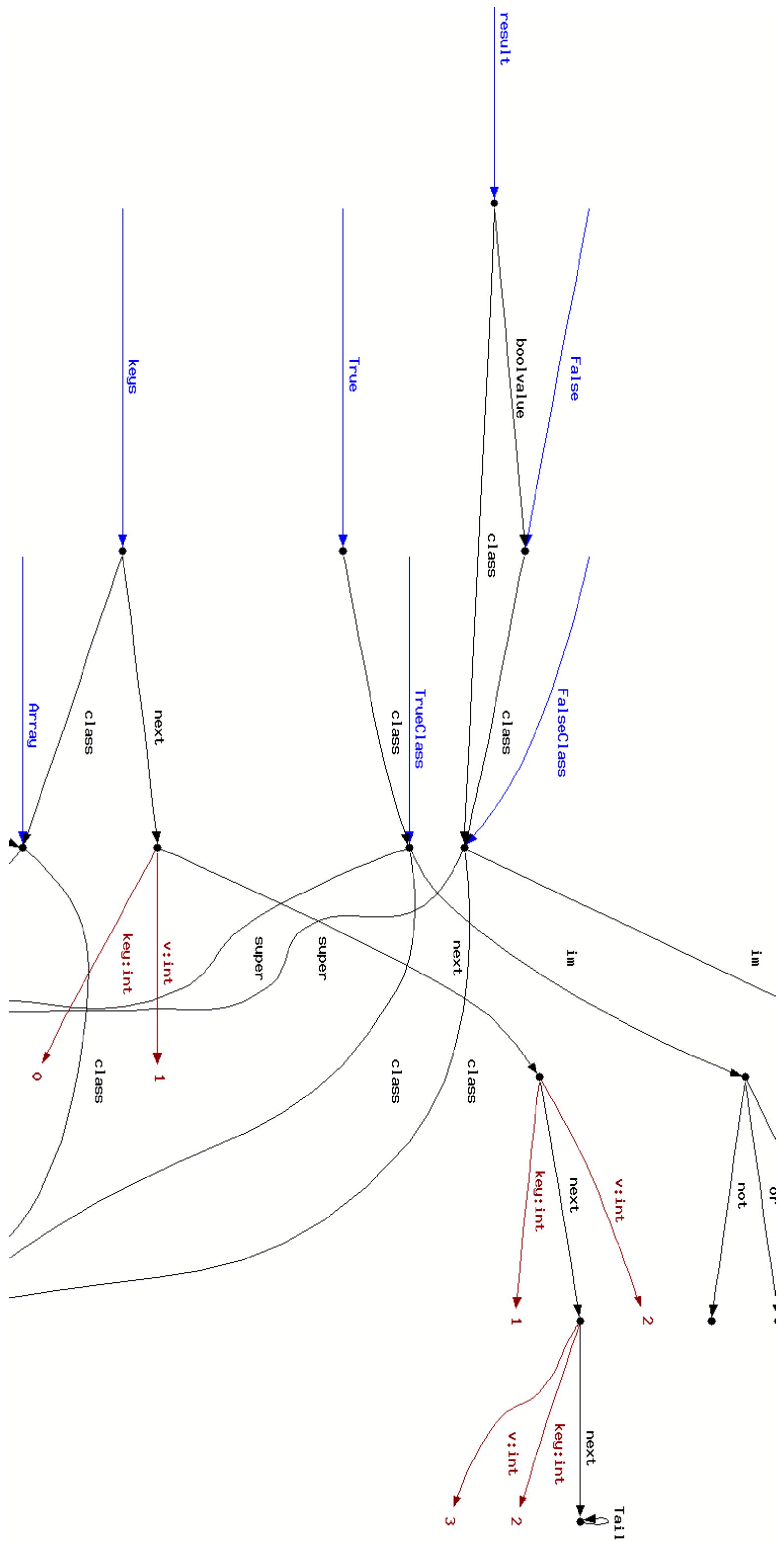


Figure 8.2 Part of the execution result of command: hash1.keys

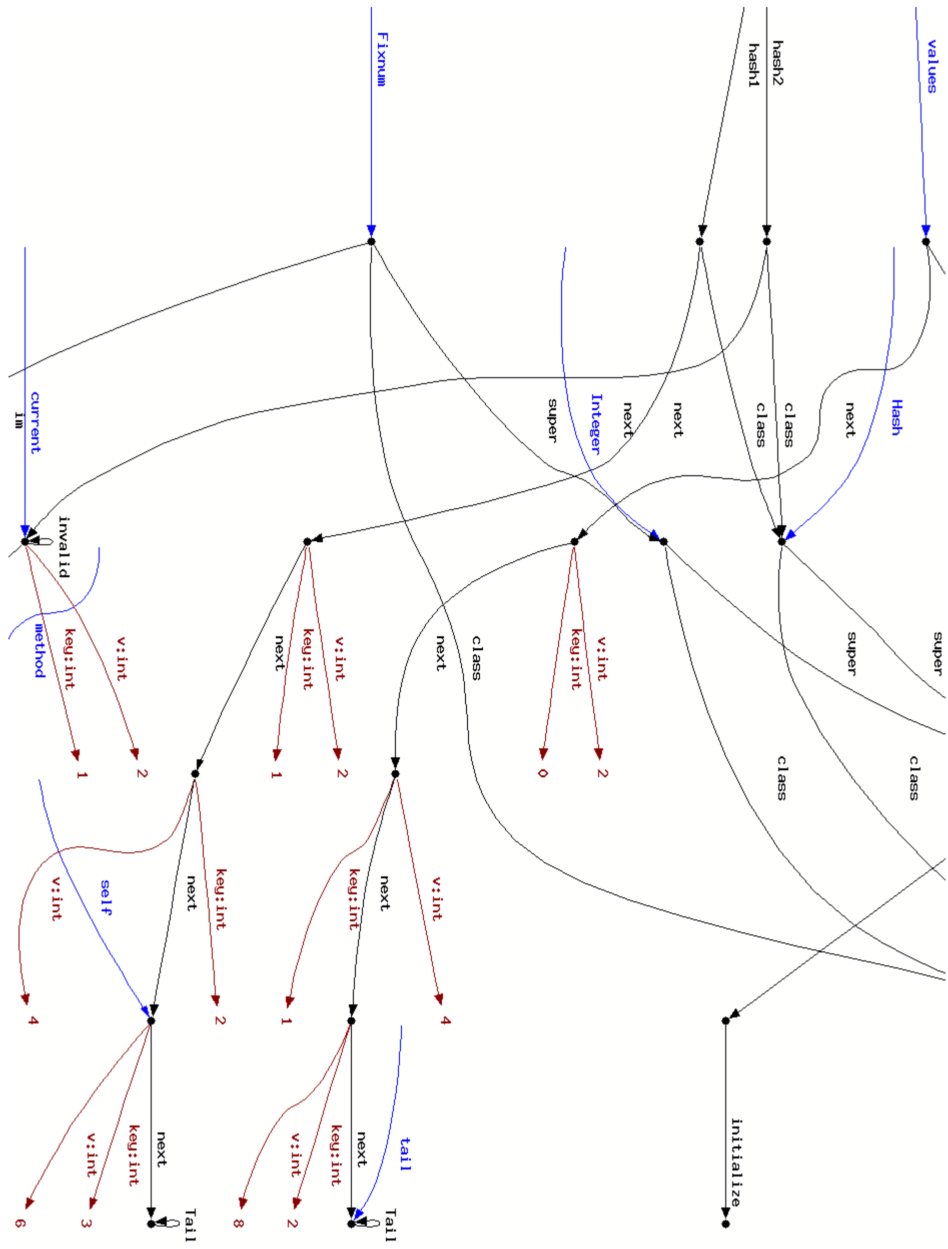


Figure 8.2 Part of the execution result of command: hash2.values

Chapter 9

Ruby Core Seven (RC7)

One of Ruby's particular strengths is the block, which is mainly used to structure programs by abstracting from loops. Blocks can be used to implement callbacks, conditionals, to pass around chunks of code and to implement iterators.

An important difference between Ruby and other OO languages is that blocks in Ruby can be invoked in any method, while in Java or C++ a block is just the code between *begin* and *end* commands that cannot be invoked by other methods or functions.

RC7 extends RC6 with blocks, iterators, objectified blocks—closures.

9.1 Blocks and iterators

The blocks we discuss here are different from those in [9]. In [9] blocks are consecutive parts of code, which can be regarded as units. In this thesis, blocks are chunks of code between braces { } or `do...end` that can be associated with method invocation, almost as if they were parameters.

In Ruby, a block may appear adjacent to a method call. Both the block and the method have the same name. That means if there is a block named `test`, then the method `test` is used to invoke this block. We can call this kind of blocks as "named" blocks. The code in the block is not executed at the time it is encountered, instead, Ruby remembers the context where the block appears (the local variables, the current object, etc.) and then enters the method.

The method can invoke the block one or more times by using the Ruby `yield` statement with or without parameters. In the block, the variables between two vertical lines | | are used to accept the parameters. If there is more than one parameter they can be separated by commas. Multiple parameters passed to a `yield` are converted to an array if the block has just one argument. Blocks can return values to the method or call a method. When the block exits,

control turns to the code immediately after the `yield` statement. Here is an example of a block and its associated method:

```
def test;
  puts “method”;
  yield;
  puts “method again”;
  end;
test {puts “block”};
```

In this example, the method and the block share the name `test`. First, the statement `puts “method”` is executed. Then the `yield` statement transfers the control from the method to the block. After the execution of the block, the control is transferred back to the method, continuing with the statement after the `yield` statement. The output of the code is:

```
method
block
method again.
```

In the following example, a parameter is passed with the `yield` statement.

```
def aparameter;
  yield 5;
end;
aparameter {|i| puts “We have #{i} blocks.”};
```

In the preceding code, the `yield` statement is followed by a 5, it passes the value 5 as a parameter to the block and the value 5 is received in the variable `i`. The output of the code is:

```
We have 5 blocks.
```

The next example is about returning a value to the method from the block.

```
def find(max);
  for i in 0…max;
       value=i;
       print(value,” ”)if yield value;
  end;
  return nil;
end;
find(10) {|v| v>5};
```

In this example, the value of the expression `v>5`, evaluated in the block, is passed back to the method as the value of the `yield` statement. The code above produces:

```
6 7 8 9
```

In the following example, a method call is used as the statement in the block.

```
def result;
```

```
    puts "Hello!";
  end;
  def threetimes;
    yield;
    yield;
    yield;
  end;
  threetimes {result};
```

The preceding code calls the method `result` three times and produces:

```
Hello!
Hello!
Hello!
```

As mentioned above, the main use of blocks is to produce user-defined control structures - especially loops. Blocks help to realize iterators. Iterators are special methods supported by collections. Collections are objects that store a group of data members, such as arrays and hashes in Ruby. Iterators can access the items in a collection one at a time. Blocks associate with iterators are normally "unnamed" blocks in contrast to "named" blocks that associate with methods.

We introduce two kinds of iterators here. The first one is `each`. This iterator returns all the elements of an array or a hash and it is always associated with a block. For example,

```
a=[1,2,3,4,5,6];
a.each{|i| puts i};
```

Here the iterator `each` returns the elements of the array one by one to the block and the value is stored in the variable `i`. The output of the code is

```
1
2
3
4
5
6
```

Another iterator introduced here is `collect`. This iterator is normally used to do something with each of the items to get a new collection. It returns a new collection by invoking the block once for all the elements, passing each element as a parameter to the block. For example,

```
a=[1,2,3,4,5,6];
b=a.collect{|i| i+1};
```

The code produces a new array `b=[2,3,4,5,6,7]` while the array `a` keeps its content as original.

There also exist many other iterators to take the place of loops in Ruby. For example, such kind of code is normally used to access each character of a string with a while loop:

```
str="abc";
i=0;
while i<str.length do;
  putc(str[i]);
   puts "\n";
  i+=1;
end;
```

But with one of the string's iterators, `each_byte`[14], the above code block can be simplified:

```
"abc".each_byte{|c| putc(c);puts "\n"};.
```

Here each character of the string "`abc`" is substituted into the block's local variable `c`. The output of this statement is the same as the previous block of code:

```
a
b
c
```

In the book "Ruby in a Nutshell"[6], the author wrote " Methods may be called with blocks of code specified that will be called from within the method "(P.21). It seems that the block is a special kind of method that can be called inside a method with the same name. The block has its own character: first, variables are local to a block unless a variable with that name already exists. This means that a block introduces its own scope for new local variables. Here is an example of a closure[15]:

```
fact=proc{|n|
  if n<=1
        1;
  else
        fact.call(n-1)*n;
  end;
```

[14] `each_byte` is an iterator, it passes each character in the string to a given block.
[15] We will introduce closures in 9.3.

```
};
fact.call(8);
```

The program produces 40320 as the result, which is the factorial of 8. In this example, `n` is defined inside the block definition and no such variable exists outside. The variable `n` does not persist beyond the block, and if the block is run multiple times, each invocation gets its own fresh copy. On the other hand, if a variable with that name already exists, then the block refers to the same variable, its value persists between the block invocations and after the last invocation. For example,

```
max=0;
[7,3,19,4].each{|v| max=v if v>max};
puts max;
```

Here the `each` iterator executes four times. In the end, `max` is 19. Outside the block, `max` keeps this value. This explains the sentence "The scope introduced by a block can refer to local variables of outer scope"[6]: a block can use the local variables that already exist. At the same time, a method has its own local variables. A method starts with a completely clean state as far as local variables are concerned; it cannot refer to any local variable defined outside.

```
i="hello";
def putsi;
  puts i;
end;
putsi;
```

This program has an error in 'putsi': `undefined local variable or method 'i' for main:Object (NameError)`, implying that we cannot use the local variable `i` inside the method.

Secondly, a block which takes parameters such as `|a,b,c|`, looks like a method definition with formal parameters, but actually these are just assignments, where the right-hand side is given at block invocation time. So the following two programs are the same:

```
proc1=proc{|a,b,c| puts a,b,c};
proc1.call("hello","to","you");
```

and

```
proc1=proc{a,b,c="hello","to","you"; puts a,b,c};
proc1.call;
```

They both produce:

```
hello
to
you
```

9.1.1 Block definitions

Just like instance methods, blocks can be called on instances of the class for which they are defined. If a block is defined in the main section, the implicit class is `Object` and the methods have to invoke this block from `Object`.

9.1.1.1 Syntax

- Block definition:
  m { |*p1,……, pn*| *statement* }
  m is the name of the block, it should be the same as the name of the method, which invokes the block. *p1* to *pn* are the names of the arguments. The *statement* makes up the body of the block definition.

9.1.1.2 Projection

Blocks are added to a branch of the class `Object` when the block is defined in the main section, `StackFrame.self` points to the `main` object and the block is added to the `Object` class. All blocks are stored in *C*`.bk`, where *C* is the name of the class and `bk` is the field that points to the object that holds all the blocks. The local variables of the block are stored in the `blv` field of `StackFrame`.

Whenever a block is invoked, the program jumps to the label instruction `L"`*C*`:bk"` in the definition below. The jump before it `##L"`*C*`:bk:skip"` is necessary to prevent the program from executing the block when it encounters the definition. After jumping to invoke a block and before executing the statements inside, the local variables are initialized with the block arguments and the `self` reference is updated. The focus `arg1` to `argn` and `self` are supported by the `yield` statement, which invokes the block.

- ψ_{class} (*m* {|*p1,……,pn*| *statement*}) =

```
self=StackFrame.self;
+self==main{;cl=Object;}{;cl=self;};
-cl/bk{;aux=new;cl.+bk=aux;};
blocks=cl.bk;
block=new;
blocks.+m=block;
```

```
block.+label:str="C:bk";
##L"C:bk:skip";

L"C:bk";
arguments=new;
arguments.+p1=arg1;……;arguments.+pn=argn;
StackFrame.+blv=arguments;
StackFrame.+self=self;
ψ(statement);
R;
L"C:bk:skip"
```

9.1.2 Invoking blocks

Invoking a block is calling a block with the same name as the associated method and executes the statements inside the block. When the block 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.

9.1.2.1 Syntax

- A block is invoked by a *yield* statement and the expressions *exp1* to *expn* supply the actual arguments when being evaluated.
  yield exp1,…… , expn

9.1.2.2 Projection

`StackFrame` keeps track of the local variables in the block invocation. The returning goto instruction is used to jump to the body of the block. When it encounters an `R` instruction, it looks up the `StackFrame` for the label and jumps back to where the block is invoked. The focuses `arg1` to `argn` are assigned to the block arguments, which are put in the local variables when the program jumps to the block.

- First, all the expressions are evaluated to objects. Then the macro *search-block* is executed. If it does not find the block, the program terminates, otherwise the block is focused by *m*, which holds the jump label. *m* is the name of the method that invokes a block with the same name.

ψ (*yield exp1,……,expn*) =

```
ψ(m);
x=result;
ψ(exp1);arg1=result;……;ψ(expn);argn=result;
```

$\varphi_{search-block}$ (m);

```
self=x;
result=nil;
label=block.label;
R##L[label]
```

- The actual block name *m* is substituted in the macro. The block invocation is sent to the object in focus `x`.

$\varphi_{search-block}$ (m) =

```
cl=x.class;
```

$\varphi_{search\text{-}supers}$ (`cl,bk,`m);

```
+found==false;!;method=res
```

- This marco is defined in [9] to search the class hierarchy for a method. Here we use $\varphi_{search\text{-}supers}$ (x,s,i) to search the class hierarchy for a block *i* in field s of class *x*. If the class *x* doesn't have the field *s*, then this class is skipped and the search continues with the `super`, the superclass of class *x*. If the block *i* is found, it is saved in focus `res`. Otherwise `false` in returned.

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

```
loop=true;found=false;sp=x;
L"search:x:s:i"
loop==true{;
        +sp/s{;br=sp.s;
              +br/i{;loop=false;found=true;res=br.i;};};
              +sp/super{;sp=sp.super;}{;loop=false;};
        ##L"search:x:s:i";}
```

9.1.3 Iterators

The iterators `each` and `collect`, introduced above, are both instance methods for arrays and hashes. In order to simplify the problem, they are treated as instance methods only for arrays in the projection.

9.1.3.1 Syntax

The syntax of the two iterators:

- *ah.each* { */p1,……, pn/ statement* }
- *ah.collect* { */p1,……, pn/ statement* }

  ah is the name of an array or a hash. *p1* to *pn* are the names of the arguments. The *statement* makes up the body of the block.

9.1.3.2 Projection

- The projection for these two iterators is similar to that for blocks. The difference is that a block that is defined by the iterator doesn't have a name, so the projection for a block's name is omitted and *p1,……, pn* are actual parameters already . *ah* denoted the name of the array or the hash that invokes this iterator.

$\psi_{iterator}$ (*ah.each* { */p1,……, pn/ statement* }) =

```
result=new;
L"ah:each";
-self/Tail{;
            result=self.v;arguments=new;
            arguments.+p1=arg1;…;arguments.+pn=argn;
            StackFrame.+blv=arguments;
            StackFrame.+self=self;
            ψ(statement);##L" ah:each";
        }
```

$\psi_{iterator}$ (*ah.collect* { */p1,……, pn/ statement* }) =

```
result=new;result.+class=Array;
tail=new;tail.+Tail;result.+next=tail;
```

```
temp=result;num=new;num.+v:int=0;
L" ah:collect";
-self/Tail{;
              aux=new;aux.+key:int=self.key;
              aux.+v:int=self.v;aux.+next;
              aux.next=tail;
              result.next=aux;result=result.next;
              self=self.next;
              arguments=new;
              arguments.+p1=arg1;…;arguments.+pn=argn;
              StackFrame.+blv=arguments;
              StackFrame.+self=self;
              ψ(statement);##L" ah:collect";
          };
result=temp.next;temp.-next
```

9.1.4 Examples of the blocks

In this example we make use of a class `Customer`. Two instances of this class are created.

```
c1=({"name"=>"George Black", "id"=>"00001"});
c2=({"name"=>"John Lee", "id"=>"00002"});
c1.each{|key, value| print key, " is ", value, "\n"};
c2.each{|key, value| print key, " is ", value, "\n"};
```

This program produces:

```
name is George Black
id is 00001
name is John Lee
id is 00002
```

Another example generates a list of **Fibonacci numbers**[16] from 0 to the max value passed from the block.

[16] The Fibonacci numbers satisfy the same recurrence relation $F_n = F_{n-1} + F_{n-2}$ for n=3,4,…, with $F_1 = 0, F_2 = 1$

```
def Upto(max);
  i1,i2=0,1;
  while i1<=max;
        yield i1;
        i1,i2=i2,i1+i2;
  end;
end;
Upto(1000) {|f| puts f};
```

The output of this program is:

```
0
1
1
2
3
5
8
13
21
34
55
89
144
233
377
610
987
```

9.2 Closures

In Ruby, blocks can be objectified explicitly and these kinds of blocks are called closures if these code blocks are wrapped along with local variable bindings. For example, `c=Proc.new{|x| x*2}`. In this assignment, the block `{|x| x*2}` is objectified and turned into a closure, as we will explain below.

Ruby's closure is a `Proc` object. The objects of the class `Proc` are chunks of code that are bound to have a set of local variables. Once bound, the chunks of code can be called in different

contexts but still access the same local variables. These local variables can be referred inside the closure, but even after the function has returned and its local scope has been destroyed, these local variables still exist as part of the closure object. These local variables are shared with the closure object and the method.

Before we talk about more features of closures, an important instance method of the class `Proc` must be introduced: `call`. The instance method `call` is to invoke the code block and returns the value of the last expression that is evaluated in the code block. It has a synonym `[]`. In the last example, both `c.call(5)` and `c[5]`evaluate the closure and return `10` as the result.

In a conventional block, once something goes out of the scope of the block, it is lost. But closures keep the context as its original appearance. For example:

```
a=1;
c=Proc.new{|v| puts v};
c.call(a);
a=20;
c.call(a);
```

Here the first `c.call` expression produces the output 1 while the second produces 20. This shows that the closure keeps the context rather than the initial value of `a`.

9.2.1 Syntax

The syntax of RC7 is extended with closures, which are translated into the objects of class `Proc`.

- `Proc.new{|`*p1*`,……,`*pn*`|` *statement* `}`
  p1 to *pn* are the names of the arguments. The *statement* makes up the body of the closure.

9.2.2 Projection

- The `Proc` class initialization should be done before the user program is executed. The projection function of the program is changed accordingly.
  `rc2ipl(u1;……;uk)=`

  $\varphi_{init-classes}$; $\varphi_{init-methods}$; $\varphi_{init\text{-}integer}$; $\varphi_{init-string}$; $\varphi_{init-proc}$; $\varphi_{init-StackFrame}$;

  ψ_{main} `(u1);……;` ψ_{main} `(uk)`

- The `Proc` class has one instance method defined in RC7: `call ([])`.

$\varphi_{init-proc}$ =

```
Proc=new;
Proc.+class=Class;Proc.+super=Object;
```

$\varphi_{im-begin}$ `(Proc,call);`

```
label=x.label;
R##L[label];
```

φ_{im-end} `(Proc,call)`

- ψ `(Proc.new{|`*p1,……,pn*`|` *statement* `} )=`

```
result=new;result.+class=Proc;
result.+label:str="C:c";
##L"C:c:skip";

L"C:c";
arguments=new;arguments.+p1=arg1;……;arguments.+pn=argn;
StackFrame.+lv=arguments;
StackFrame.+self=self;
ψ(statement);
R;
L"C:c:skip"
```

9.2.3 Example

This is a very simple example about a method `repeat_n`, which multiplies the value of a local variable with the parameter of the closure.

```
def repeat_n(n);
   return Proc.new{|m| m*n};
end;
p3=repeat_n(3);
p4=repeat_n(4);
p3.call(5);
p4.call(p3.call(3));
```

The return values of the last two expressions are `15` and `36`.

Chapter 10

The Implementation of Our Projections in IPL

Up to Chapter 9, four new subsets, which are consecutive extensions to the former four subsets defined in [9], have been presented starting with RC4+, RC5, RC6 and RC7. RC4+, RC5 and RC6 extended the subset incrementally with features. In RC4+, three new instance methods of integers are extended: *monus* (-), *greater than test* (>) and *less than test* (<). RC5 extends RC4+ by adding two new types: Boolean and string. RC6 extends the previous projection with another two important type constructors: arrays and hashes. RC7 adds one of Ruby's particular features: blocks. In this section, only the projections of RC7 to program algebra has been presented. To simplify matters, the programming work is left out.

In this chapter, we will explain how the projection functions in RC4+, RC5 and RC6 are used in the parser. Some parts of our programs (in Appendix B) will be mentioned and explained in this chapter.

10.1 Framework

The programming language used in our projections is the same as the language in the implementation of Geerlings' projections. We continue to use the framework we designed in Chapter 5 with some extensions.

First, the subset of Ruby is extended to RC6. The commands in between that are used for program execution are not changed.

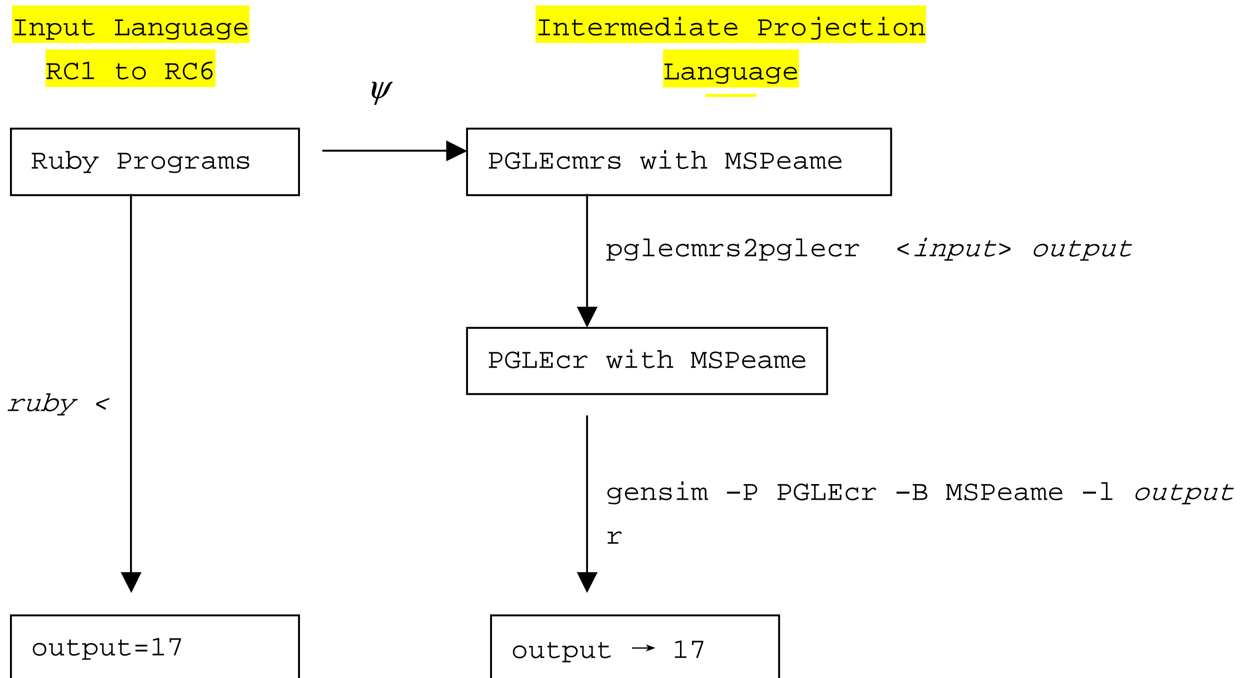


Figure 10.1 Scheme

Secondly, the keyword table is extended with several new keywords defined in RC4+ to RC6.

Keywords	**Projection**
-	Monus
>	GT
<	LT
&&	And
||	Or
[]	ArrayDefinition
=>	HashDefinition
&	Intersection
+	Concatenation
{}	Block

Table 9.1 Keywords and their responding methods continued

10.2 Implementation of the projections in IPL

10.2.1 Projections implemented in RC4+

RC4+ adds three instance methods of integers: *monus* (-), *greater than test* (>) and *less than test* (<). Because integers in PGA are non-negative numbers, our projection is about monus instead of minus, which means if the minuend is less than the subtrahend, 0 is returned. And the last two instance methods are used later in Boolean examples.

- **Monus expression:**

  `exp0-exp1`

  Using the $\varphi_{keyword-fetch}$ method, the Ruby instruction: `x-y` will be read character by character until we find the keyword "-". And the two operands will be assigned to `self` and `arg1`.

```
self   keyword   arg1

x         -         y
```

  According to the keyword table, the method "`Monus`" will be called. Then the projection functions inside the braces will be applied.

```
{result=new;result.+class=Fixnum;result.+v:int;
result.v=self.v;
decr result.v arg1.v;}
```

- **Greater Than expression:**

  `exp0>exp1`

  Using the $\varphi_{keyword-fetch}$ method, the Ruby instruction: `x>y` will be analyzed as below.

```
self   keyword   arg1

x         >         y
```

  In this example the keyword is ">" and the two operands will be assigned to `self` and `arg1`. According to the keyword table, the method "`GT`" will be called. Then the projection functions inside the braces will be applied.

```
{i=self.v; j=arg1.v;
+decr i j{;result=new;
              result.+boolvalue=True;
              result.+class=TrueClass;
           }{;result=new;
               result.+boolvalue=False;
               result.+class=TrueClass; };
}
```

- **Less Than expression:**
  `exp0<exp1`

  Using the $\varphi_{keyword-fetch}$ method, the Ruby instruction: `x<y` will be analyzed as below.

```
                     self   keyword   arg1

                     x          <         y
```

In this example the keyword is "<" and the two operands will be assigned to `self` and `arg1`. According to the keyword table, the method "LT" will be called. Then the projection functions inside the braces will be applied.

```
{i=self.v; j=arg1.v;
  +decr j i{;result=new;
                result.+boolvalue=True;
              result.+class=TrueClass;
           }{; result=new;
              result.+boolvalue=False;
              result.+class=TrueClass;};
}
```

10.2.2 Projections implemented in RC5

RC5 add two basic types: Boolean and string. Ruby supports all the standard Boolean operators. String is probably the largest Ruby class, with over 75 standard methods. We won't go through them all here. Instead, we'll look at some common idioms---things that are likely to pop up during day-to-day programming.

- **And expression:**

  `exp0&&exp1`

  Using the $\varphi_{keyword-fetch}$ method, the Ruby instruction: `true&&true` will be analyzed as below.

```
self    keyword    arg1

true      &&         true
```

In this example the keyword is "&&". According to the keyword table, the method `And(true,true)` will be called and the two operands will be assigned to `self` and `arg1`. The projection functions inside the braces will be applied.

```
{+self==true{;
        result=new;result.+class;result.+boolvalue;
         +arg1==true{;
                        result.boolvalue=True;
                         result.class=TrueClass;
                       }{;
                              result.boolvalue=False;
                              result.class=FalseClass; };
  };
  +self==false{;
        result=new;result.+class;result.+boolvalue;
        result.boolvalue=False;result.class=FalseClass;
  };
}
```

In this example, the first operand is true then it evaluates the second operand, which is also true. As a result, the final result will be true. If the first operand is false, it won't evaluate the second operand and the result will be false.

- **Or expression:**

  `exp0||exp1`

  Using the $\varphi_{keyword-fetch}$ method, the Ruby instruction: `true||true` will be analyzed as below.

```
self    keyword    arg1

true    ||         true
```

According to the keyword table, the method `Or(true,true)` will be called and the two operands will be assigned to `self` and `arg1`. The projection functions inside the braces will be applied.

```
{+self==true{;
  result=new;result.+class=TrueClass;result.+boolvalue=True;};
 +self==false{;
    result=new;result.+class;result.+boolvalue;
   +arg1==true{;
                              result.boolvalue=True;
                  result.class=TrueClass;
                }{;
                  result.boolvalue=False;
                  result.class=FalseClass; };
  }
```

Here in this example, the first operand is true. Then it won't evaluate the second operand and returns true. If the first operand is false, it will evaluate the second operand. If the second operand is true it returns true, otherwise it returns false.

- **Not expression:**
  `!exp0`

Using the $\varphi_{keyword-fetch}$ method, the Ruby instruction: `!true` will be analyzed as below.

```
keyword    self

!          true
```

According to the keyword table, the method `Not(true)` will be called and the operand will be assigned to `self`. The projection functions inside the braces will be applied.

```
{+self==true{;
result=new;result.+class=FalseClass;result.+boolvalue=False;};
 +self==false{;
```

```
  result=new;result.+class=TrueClass;result.+boolvalue=True;}
}
```

In this example, the operand is true. It returns the opposite of its operand, which is false.

- **Equality test statement:**

  `string1==string2`

  Using the $\varphi_{keyword-fetch}$ method, the Ruby instruction: `string1==string2` will be analyzed as below.

```
self     keyword     arg1

string1     ==     string2
```

In this example the keyword is "==" and the two operands will be assigned to `self` and `arg1`. According to the keyword table, the method `Equaltest` will be called. Then the projection functions inside the braces will be applied.

```
{result=new;result.+class;result.+boolvalue;
  +self.content==arg1.content{;
                               result.boolvalue=True;
                               result.class=TrueClass;
                              }{;
                               result.boolvalue=False;
                               result.class=FalseClass};
}
```

- **Concatenation statement:**

  `string1+string2`

  Using the $\varphi_{keyword-fetch}$ method, the Ruby instruction: `string1+string2` will be analyzed as below.

```
self     keyword     arg1

string1     ==     string2
```

In this example the keyword is "+" and the two operands will be assigned to `self` and `arg1`. According to the keyword table, the method `Concatenation` will be called. Then the projection functions inside the braces will be applied.

```
{
  result=new;result.+class=String;result.+content:str;
  temp=new;temp+content:str=self.content;
  append temp.content arg1.content;
  result.content=temp.content;
}
```

10.2.3 Projections implemented in RC6

RC6 extends the previous projection with another two important type constructors: arrays and hashes. Ruby's arrays and hashes are indexed collections. Both store collections of objects, accessible using a key. With arrays, the key is an integer, whereas hashes support any object as a key. Both arrays and hashes provide several instance methods. I won't repeat the projections of index methods explained in Chapter 8 here.

- **Concatenation expression:**
  `exp0+exp1`

  Using the $\varphi_{keyword-fetch}$ method, the Ruby instruction: `array1+array2` will be analyzed as below.

```
exp0     keyword     exp1

array1      +        array2
```

According to the keyword table, the method `Concatenation(array1,array2)` will be called and the two operands will be assigned to `self` and `arg1`. The projection functions inside the braces will be applied.

```
Concatenation(array1,array2){
tail=new;tail.+Tail;result=new;result.+next=tail;
result.+class=Array;current=result;num=0;

L"Aconcatenation:1";
-self/Tail{;
                aux=new;aux.+key:int;aux.+v:int;aux.+next;
                aux.key=self.key;aux.v=self.v;aux.next=tail;
                result.next=aux;result=result.next;
                self=self.next;incr num;##L"Aconcatenation:1";
```

```
};

L"Aconcatenation:2";
-arg1/Tail{;
                aux=new;aux.+key:int;aux.+v:str;aux.+next;
                aux.key=arg1.key;incr aux.key
                num;aux.v=arg1.v;
                aux.next=tail;result.next=aux;
                result=result.next;
                arg1=arg1.next;##L"Aconcatenation:2";
};
result=current;
}
```

- **Intersection expression:**
  `exp0&exp1`

  Using the $\varphi_{keyword-fetch}$ method, the Ruby instruction: `array1&array2` will be analyzed as below.

```
exp0    keyword    exp1

array1     &       array2
```

According to the keyword table, the method `Intersection(array1,array2)` will be called and the two operands will be assigned to `self` and `arg1`. The projection functions inside the braces will be applied.

```
Intersection(array1,array2){
tail=new;tail.+Tail;result=new;result.+class=Array;
result.+next=tail;current=result;

L"Aintersection:1";
-self/Tail{;
   L"Aintersection:2";
      -arg1/Tail{;
             +self.v==arg1.v{;
                            aux=new;aux.+key:int;
                            aux.+v:int;aux.+next;
```

```
                                        aux.key=self.key;
                                        aux.v=self.v;aux.next=tail;
                                        result.next=aux;
                                        result=result.next;
                                        self=self.next;arg1=current;
                                        ##L"Aintersection:1";
                                    }{;
                                        arg1=arg1.next;
                                        ##L"Aintersection:2";
                                  };
                        };
        self=self.next;##L"Aintersection:1";
};
result=current;
}
```

10.3 Easy and difficult aspects of this implementation

We have built up the construct of parser in Chapter 5. So it is easy for us to use it as foundation and do some extensions.

But at the same time, some new problems arise. For example, there is confusion of keywords. In the former projection functions, there are a lot of classes having the same instance methods such as integers, strings and array classes. All of them contain + and == operations. The keywords are limited in the keyword table, which means method names are no longer mapped one-to-one to the keywords. How can we distinguish the different instance methods with the same keyword?

In our programs, we parse the first operand. If it is an integer, then we go to the addition methods defined in the `integer` object. If it is a string, then we call the concatenation methods defined in the `string` object. Or else, we will call the concatenation methods defined in the `array` object.

Chapter 11

Other Interesting Features in Our Project

In this project, we first gave the theoretical projection functions of a Ruby subset to PGA in the IPL we defined, and then implemented these functions. In this chapter, some interesting features we found in the process of this implementation will be discussed. These features have nothing to do with the projection functions, but they helped us with the implementation.

11.1 Projection versus compilation

In the implementation of the projection, the parser starts with a `compile` instruction. Compilation of the input Ruby program gives the molecule that is a null-terminated list of instructions connected by `next` fields. At the entrance of the list, an atom with the field `basic:str`, which selects a basic Ruby instruction, will be evaluated. Evaluation continues with the next instruction in the list.

The Ruby instructions will be read two times. The first time, as explained in the last paragraph, we transfer the Ruby instructions into a molecule. The second time we deal with the instructions saved in the molecule list and this is the beginning of the parsing. In fact the first time of reading the Ruby instructions is not a part of projection. It leaves all the input Ruby instructions untouched but separates them by semicolons. This process turns the input Ruby program, stored in a string, into a list of instructions. It helps the parser to dispose the Ruby instructions more easily. For example, we can put a small Ruby program in a string `x`. Compilation of `x` gives the molecule in Figure 11.1.

```
x="class Shape;
       def initialize(a);
           @e=a;
       end;
   end;";
compile x;
```

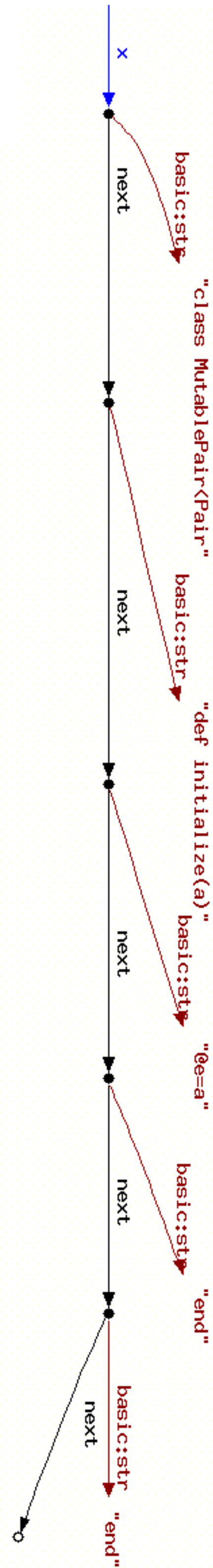


Figure 11.1 Compilation result of `x`

11.2 Operations on strings

In MSP, five operations on strings have been defined. They are `first`, `delfirst`, `append`, `int` and `str`. We introduced these operations in section 3.2.1. In our project, we can define other operations on strings with the help of the existing basic instructions.

In Figure 11.1 we can see that after "compilation", all the input Ruby instructions are stored in the fields named `basic:str` of the list of atoms in the molecule. We take the Ruby instruction stored in the first atom of the list, `"Class Shape"` as an example. This instruction represents the beginning of a class definition in Ruby. It will be separated into two parts later on. The first part is the key word `"Class"` used to call the *method definition* method. The second part is the name of the class "`Shape`". It is a parameter that will be transferred to the *method definition* method. In the projection functions defined in [9], we use an atom in the focus of the class name to represent the class. We will create an atom in focus `Shape` in this example. Here comes the problem. Till now, `"Shape"` is a string in a value field of an atom, how can we turn this string to the name of an atom?

There are various ways to solve this problem. For example, we can define a new operation on strings as a supplement to the current basic instruction set. We name it `strname`.

```
strname extfocus1 extfocus2
```

Here `extfocus1` is a string. This instruction uses the string value selected by `extfocus1` to name `extfocus2`. If `extfocus2` is an existing atom, then it is placed in the focus of the string value selected by `extfocus1`. If it does not exist, a new atom is created in the focus of the string value selected by `extfocs1`. If it is a field selection, then the field is named by the string value selected by `extfocus1`. Otherwise, `false` is returned.

Alternatively, we can define a method and use this method to solve this problem. We prefer this solution because the IPL we defined in this project is suitable to write such a method and we don't need to extend PGA and the PGA Toolset. The core instruction we use here is `eval`, which compiles a string into a molecule and evaluates it. It has been explained in section 3.2.2.

11.2.1 String-to-New-Atom

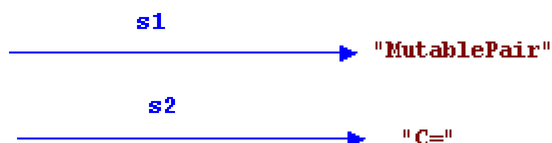
In this section, we define a method `stra` to use a string as the focus name of an atom. It is mainly used in the projection of the class definitions.

```
stra(s1,s2){;
        append s2 s1;
        append s1 "=new;";
        append s1 s2;
        eval s1;
};
```

Let’s use an example to show what this method does.

```
str1="MutablePair";
str2="C=";
stra(str1,str2);

stra(s1,s2){;
  append s2 s1;
  append s1 "=new;";
  append s1 s2;
  eval s1;
};
```

We define two strings: `str1` and `str2` in the first two instructions. Then we call methods `stra` with `str1` and `str2`. `str1` and `str2` are transferred into the method and accepted by `s1` and `s2`.



Then we append the string selected by `s1` to the end of the string selected by `s2`.

In the following step, we append the string “`=new;`” to the end of the string selected by `s1`.

Then, we append the string selected by `s2` to the end of the string selected by `s1`.

At last, the `eval` instruction complies the string selected by `s1` into a molecule and evaluates

the instructions stored in this string. In other words, the command inside the double quotes selected by `s1` is executed in PGA. The execution result is depicted in Figure 11.2.

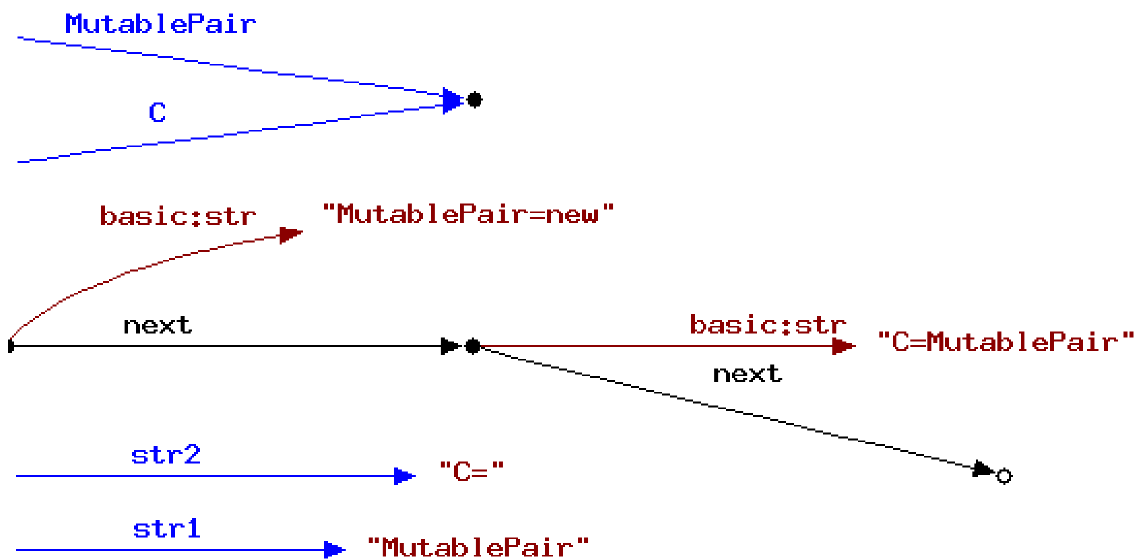


Figure 11.2 Result of the String-to-New-Atom example

11.2.2 String-to-Existing-Atom

In some other cases, we want to place an existing atom in the focus of a string. Here a new method is defined to implement this. It is mainly used in the *search-instance-method* and *search-supers* macros defined in [9].

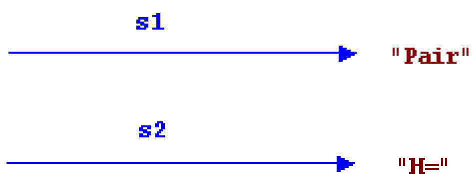
```
stre(s1,s2){;
        append s2 s1;
        eval s2;
};
```

For example, there is a class definition that begins with this instruction: `class MutablePair<Pair`. The `Pair` class, which is the superclass of the `MutablePair` class, must already exist in the molecule when defining its subclass. Thus, when we create the class `MutablePair`, we should first find its superclass in the class hierarchy instead of generating a new class object. Here is the PGA program for this example.

```
str1="Pair";
str2="H=";
stre(str1,str2);
```

```
stre(s1,s2){;
   append s2 s1;
   eval s2;
};
```

Here we first define two strings: `str1` and `str2`. Then we call the method `stre` with two parameters `s1` and `s2`, which are used to accept two strings.



We append the string selected by `s1` to the end of the string selected by `s2`.

At last, the `eval` instruction compiles the string selected by `s2` into a molecule and evaluates the instruction selected by this string. In other words, the command inside the double quotes selected by `s2` is executed in PGA. The execution result is depicted in Figure 11.3.

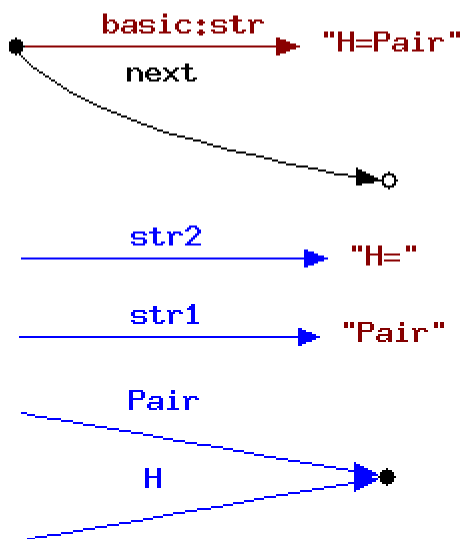


Figure 11.3 Result of evaluation

11.2.3 String-to-Field

All the instance methods are placed in the field `C.im`, where `C` is the class name and `im` is its field pointing to the object that holds all its instance methods. We need a method here to transfer a string to a field name. It is mainly used in the projection of the method definition.

```
strf(s1,s2,s3){;
        append s2 s3;
        append s2 "=";
        append s2 s1;
        append s1 "=new;";
        eval s1;
        eval s2;
};
```

This is a part of the projection function of the method definition in [9].

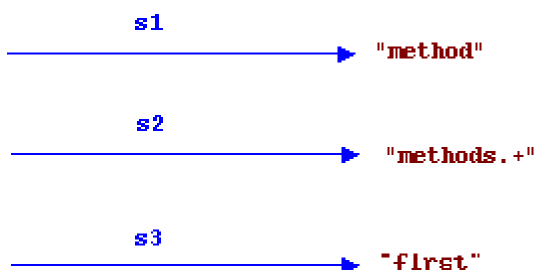
```
methods=cl.im;
method=new;
methods.+first;
methods.first=method;
```

In our implementation, this part of code is changed.

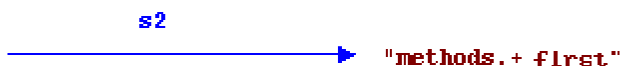
```
str=″first″;
methods=cl.im;
strf(“method″,″methods.+″,str);

strf(s1,s2,s3){;
   append s2 s3;
   append s2 “=″;
   append s2 s1;
   append s1 “=new″;
   eval s1;
   eval s2;
};
```

In this part of the program, we define one string: `str` in the first instruction, which is used to store the method name `″first″`. Then we call the method `strf` with two string constants `“method″` and `″methods.+″` and one string variable `str` as the parameters. They are

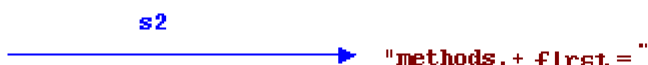
transferred into the method and accepted by s1, s2 and s3.



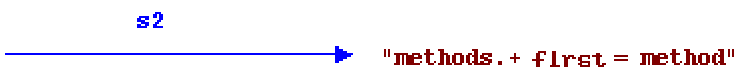
Then we append the string selected by s3 to the end of the string selected by s2. This instruction do the real method definition, which adds a new field ".first" to the "methods.+" branch.



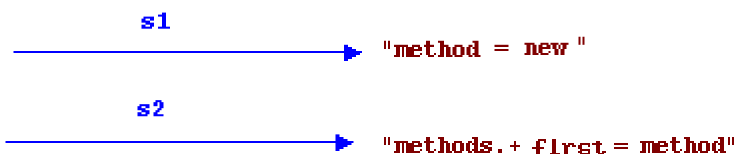
In the following step, we append the string "=" to the end of the string selected by s2.



Then we append the string selected by s1 to the end of the string selected by s2.



Further more, we append the string "=new" to the end of the string selected by s1.



The instructions stored in the string selected by s1 create a new atom and assign it to method. The instructions inside the double quotes selected by s2 add a field first to the atom in focus methods, and assigning it the atom in focus method. At last, the eval instruction compiles the string selected by s1 and the string selected by s2 into molecules and evaluates the instructions selected by these two strings. In other words, the command inside the double quotes selected by s1 and s2 are executed in PGA. The execution result is depicted in Figure 11.4.

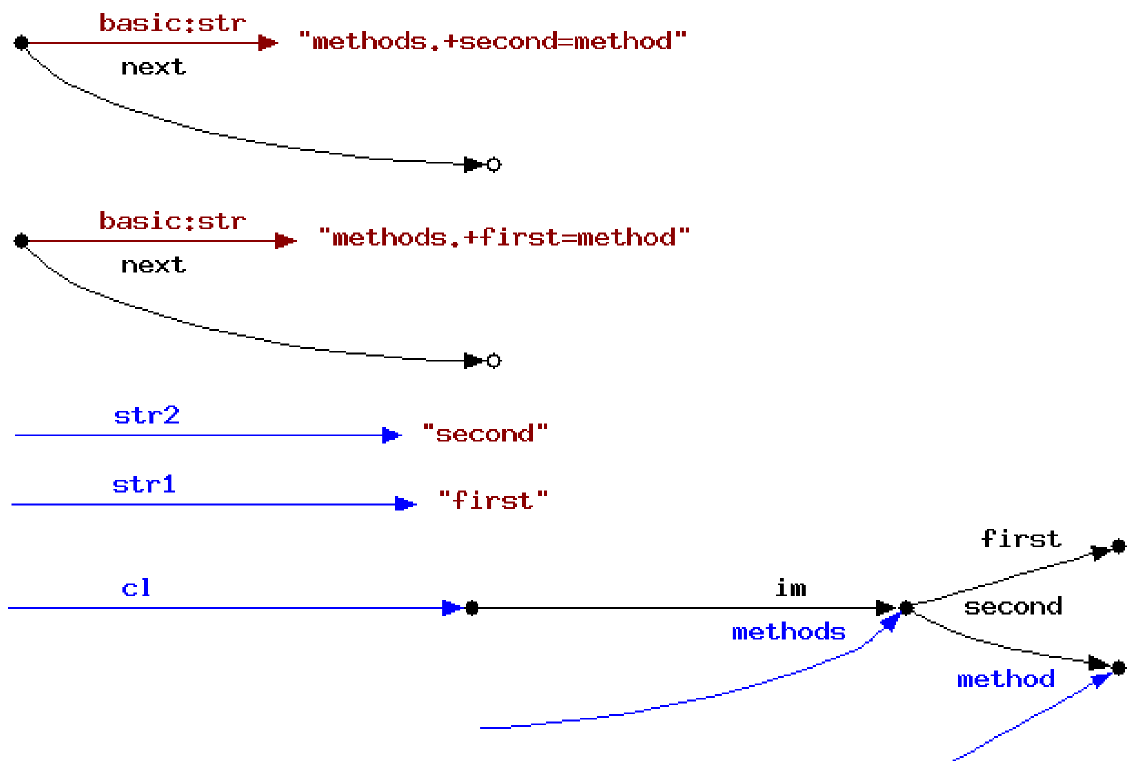


Figure 11.4 Result of evaluation

Chapter 12

Conclusion

In this thesis, a projection of a Ruby subset to PGA has been described as a follow-up of Geerlings' work [9]. The whole projection focuses on Ruby's OO constructors and data types. It can be regarded as a theoretical compiler of this Ruby subset. The projection is designed using 8 Ruby subsets: RC1 to RC7, including RC4+. Each former subset is the basis of the latter one (i.e., RC2 includes RC1, RC3 includes RC2 and so on). RC1 to RC4 are defined in [9] while the other four subsets, RC4+ to RC7 are the extensions introduced in this thesis:

- RC4+ extends RC4 with several integer instance methods.
- RC5 adds two new data types: Boolean and string.
- RC6 adds arrays and hashes.
- RC7 adds blocks, iterators, objectified blocks—closures and continuations.

The projection is defined in an intermediate projection language (IPL): a combination of the primitive instruction set PGLEcmrs and the basic instruction set MSPeame. Both these instruction sets are extensions of PGA that we implemented in the PGA Toolset. The definition and the implementation of these extensions are described in this thesis.

We have also described how to implement the projection of the Ruby subsets to PGA with the IPL we defined. The projection functions defined in [9] are implemented as the first step. The structure of this implementation is built up in this step. In the second step, the projection functions defined in this thesis are implemented. With this implementation, an I/O equivalent PGA program can be generated from a program in the Ruby subsets we described.

Before starting this project, we did some literature study, including Geerlings' master thesis [9], some Ruby books [3][6], some papers about PGA [4][5][7][8] and some relevant websites [1][2][10][12][14]. This helped us to decide the topic of our project and organize this thesis. We also studied a programming language Perl [11][13] because it is the implementation language of the PGA Toolset.

When choosing the features of Ruby to project, we faced some troubles. Ruby is a so interesting language that it has many attractive features, such as blocks and regular expressions. At the same time, we wanted to make our work practical, so we had to narrow down the features of Ruby we worked with. After discussion with our supervisors, we decided to start with the basic elements: data types. Later on, we gave the projection functions of Ruby's blocks and the objectified blocks.

Another problem on which we spent quite some time is the extension of PGA and the implementation in the PGA Toolset. Obviously, there are various extensions we can work with to implement the projections of the Ruby subset to PGA. At last, we worked with the string labels and some operations on the integers because they are practical and sufficient for our project, as well as not too complex.

Doing this project as a follow-up of another master's project, we focused on the methodology described in the latter work [9]. Both [9] and our work show that PGA is not less expressive than some higher level OO programming languages and it is possible to use PGA to model the core of such languages as well as help us to understand those languages.

References

[1] http://www.ruby-doc.org/
The online Ruby documentation project. It provides complete and accurate documentation for the Ruby programming language. It includes the extraction of the book "Programming Ruby - The Pragmatic Programmer's Guide", copyright © 2001 by Addison Wesley Longman.

[2] http://www.science.uva.nl/research/prog/projects/pga/
The homepage of Program Algebra, Programming Research Group, Faculty of Science, University of Amsterdam.

[3] Suresh Mahadevan. Making Use of Ruby.
Wiley Publishing, Inc. ISBN # 0-471-21971-X, 2002.

[4] J.A. Bergstra and I. Bethke. Molecular Dynamics.
Journal of Logic and Algebraic Programming, 51(2):193-214,2002, ISSN#1567-8326.

[5] J.A. Bergstra and M.E. Loots. Program Algebra for Sequential Code.
Journal of Logic and Algebraic Programming, 51(2):125-156,2002, ISSN#1567-8326.

[6] Yukihiro Matsumoto. Ruby in a Nutshell.
O'Reilly & Associates, Inc. ISBN # 0-596-00214-9, 2002.

[7] Bob Diertens. Molecular Scripting Primitives.
Electronic report PRG0401, Programming Research Group, University of Amsterdam, June 2004.

[8] Bob Diertens. A Toolset for PGA.
Electronic report PRG0302, Programming Research Group, University of Amsterdam, October 2003.

[9] Ruben M. Geerlings. A Projection of the Objection Oriented Constructs of Ruby to Program Algebra.
Master thesis, Programming Research Group, University of Amsterdam, November 2003.

[10] Yukihiro Matsumoto. The Ruby Programming Language.
An article written by the creator of the Ruby language. It gives a brief summary of the language. Available from: http://www.informit.com/articles/article.asp?p=18225&redir=1.

[11] L. Wall, T. Christiansen, and J. Orwant. Programming Perl (3rd Edition).
O'Reilly & Associates, Inc. ISBN # 0-596-00027-8, 2000.

[12] Lakshmi Sastry and Venkat VSS Sastry. Tcl/Tk Cookbook.
Available from: http://www.bitd.clrc.ac.uk/Publications/Cookbook/.

[13] R.L. Schwartz and T. Phoenix. Learning Perl (3rd Edition).
O'Reilly & Associates, Inc. ISBN # 0-596-00132-0, 2001.

[14] http://www.ruby-lang.org/en/
Ruby home page. It provides a large amount of resource for Ruby, such as the documents and the Ruby running environments.

Appendix A

Fluids of the Projection Examples' Execution Results

There are seven fluids discussed in this appendix: Initialization.gif, RC1.gif, Integerplus.gif, Boolean.gif, String.gif, Array.gif and Hash.gif. They are the execution results of the projection examples in this thesis. Because they are too large to put in the document, we provide these .gif files via the internet.

- **Initialization.gif**
  This fluid's corresponding Ruby program can be found in section 5.3. It shows the initial program state, including the classes `Object`, `Class`, `Integer`, `Fixnum`, `TrueClass`, `FalseClass` and `NilClass`, as well as their instance methods. Because this initialization is done before we worked with RC4+ to RC7, the classes `Boolean`, `String`, `Array` and `Hash` are not included.
  Initialization.gif is available from:
  http://www.idealworld.hostrocket.com/Thesis/draft/AppendixA/Initialization.gif

- **RC1.gif**
  This fluid's corresponding Ruby program can be found in section 5.5.1. It shows the final state of running RC1's sample program in the PGA Toolset. The result of this sample input Ruby program, which is the atom in focus `result` can be found in this fluid.
  RC1.gif is available from:
  http://www.idealworld.hostrocket.com/Thesis/draft/AppendixA/RC1.gif

- **Integerplus.gif**
  This fluid's corresponding Ruby program can be found in section 6.2. It shows the final state of executing the example for RC4+, which extends RC4 with some instance methods. These new instance methods can be found in the `im` field of the atom in focus `Fixnum`. The atom in focus `result` shows the execution result of this program.
  Integerplus.gif is available from:
  http://www.idealworld.hostrocket.com/Thesis/draft/AppendixA/Integerplus.gif

- **Boolean.gif**
  This fluid's corresponding Ruby program can be found in section 7.2. It shows the final

state of executing the example for Booleans. The atom in focus `result` shows the execution result of this program.
Boolean.gif is available from:
http://www.idealworld.hostrocket.com/Thesis/draft/AppendixA/Boolean.gif

- **String.gif**
  This fluid's corresponding Ruby program can be found in section 7.4. It shows the final state of executing the example for strings. The atom in focus `result` shows the execution result of this program. This atom has a string field: "Making use of Ruby from Suresh Mahadevan".
  String.gif is available from:
  http://www.idealworld.hostrocket.com/Thesis/draft/AppendixA/String.gif

- **Array.gif**
  This fluid's corresponding Ruby program can be found in section 8.2. It shows the final state of executing the example for arrays. The atom in focus `result` shows the execution result of this program.
  Array.gif is available from:
  http://www.idealworld.hostrocket.com/Thesis/draft/AppendixA/Array.gif

- **Hash.gif**
  This fluid's corresponding Ruby program can be found in section 8.4. It shows the final state of executing the example for hashes. The atom in focus `keys` and `values` show the execution result of this program.
  Hash.gif is available from:
  http://www.idealworld.hostrocket.com/Thesis/draft/AppendixA/Hash.gif

Appendix B

Implementation Programs for the Projection of the Ruby Subset to PGA

In this appendix, 6 example programs are given. The core parser used in these programs or modules are explained in Chapter 5 and Chapter 10. A detail explanation of the programs using array examples is also given in this appendix.

- **recursion**
  This example shows the projection of an entire RC1 program to a PGA program. It contains all constructs and instance methods in RC1. A recursive method is used in the input program, which allows multilayer S*tackFrame* for every method call. The input program is based on the primitive instruction set PGLEcmrs with basic instruction set MSPeame. Before the input program is executed in the Toolset, it should be first projected to the PGLEcr program that is executable in program algebra.
  We use the command:
  `pglecmrs2pglecr <recursion> output`
  The result of executing can be seen in a fluid generated by such a command:
  `gensim –P PGLEcr –B MSPeame –v –l output`
  It is available from:
  http://www.idealworld.hostrocket.com/Thesis/draft/AppendixB/recursion

- **integer**
  This example shows the projection of an RC4 program to a PGA program. It contains all the constructs and instance methods in `integer` class. The input program is based on the primitive instruction set PGLEcmrs with basic instruction set MSPeame. Before the execution of the input program in the Toolset, it should be first projected to the PGLEcr program that is executable in program algebra.
  We use the command:
  `pglecmrs2pglecr <integer> output`
  The result of executing can be seen in a fluid generated by such a command:
  `gensim –P PGLEcr –B MSPeame –v –l output`
  It is available from:
  http://www.idealworld.hostrocket.com/Thesis/draft/AppendixB/integer

- **Boolean**

  This is an example of the basic type: Boolean in RC5. It contains all constructs and instance methods in `Boolean` class and shows the projection of an RC5 program to a PGA program. Two instance methods: > and < of `integer` class in RC4+ are also used in the example. The input program is based on the primitive instruction set PGLEcmrs with basic instruction set MSPeame. Before the execution of the input program in the Toolset, it should be first projected to the PGLEcr program that is executable in program algebra.
  We use the command:
  ```
  pglecmrs2pglecr <Boolean> output
  ```
  The result of executing can be seen in a fluid generated by such a command:
  ```
  gensim -P PGLEcr -B MSPeame -v -l output
  ```
  It is available from:
  http://www.idealworld.hostrocket.com/Thesis/draft/AppendixB/Boolean

- **string**

  This is an example of the basic type: string in RC5. It contains all constructs and instance methods in `String` class and shows the projection of an RC5 program to a PGA program. The input program is based on the primitive instruction set PGLEcmrs with basic instruction set MSPeame. Before the execution of the input program in the Toolset, it should be first projected to the PGLEcr program that is executable in program algebra.
  We use the command:
  ```
  pglecmrs2pglecr <string> output
  ```
  The result of executing can be seen in a fluid generated by such a command:
  ```
  gensim -P PGLEcr -B MSPeame -v -l output
  ```
  It is available from:
  http://www.idealworld.hostrocket.com/Thesis/draft/AppendixB/string

- **array**

  This is an example of the basic type: array in RC6. It contains all constructs and instance methods in `array` class and shows the projection of an RC6 program to a PGA program. The input program is based on the primitive instruction set PGLEcmrs with basic instruction set MSPeame. Before the execution of the input program in the Toolset, it should be first projected to the PGLEcr program that is executable in program algebra.
  We use the command:
  ```
  pglecmrs2pglecr <array> output
  ```
  The result of executing can be seen in a fluid generated by such a command:
  ```
  gensim -P PGLEcr -B MSPeame -v -l output
  ```

It is available from:
http://www.idealworld.hostrocket.com/Thesis/draft/AppendixB/array

- **hash**

  This is an example of the basic type: hash in RC6. It contains all constructs and instance methods in `hash` class and shows the projection of an RC6 program to a PGA program. The input program is based on the primitive instruction set PGLEcmrs with basic instruction set MSPeame. Before the execution of the input program in the Toolset, it should be first projected to the PGLEcr program that is executable in program algebra.
  We use the command:

  ```
  pglecmrs2pglecr <hash> output
  ```

  The result of executing can be seen in a fluid generated by such a command:

  ```
  gensim -P PGLEcr -B MSPeame -v -l output
  ```

  It is available from:
  http://www.idealworld.hostrocket.com/Thesis/draft/AppendixB/hash

- **Explanation.pdf**

  This explanation of programs and models is based on the example of arrays. It describes the parser programs and other interesting features of our implementation parts by parts.
  It is available from:
  http://www.idealworld.hostrocket.com/Thesis/draft/AppendixB/Explanation.pdf

Appendix C

Programs and Modules for Implementing the PGA Toolset Extensions

In this appendix, the Perl-programs and Perl-modules that extend the PGA Toolset with the extensions we discussed in Chapter 3 are referred to. The core Perl-scripts in these programs or modules are explained in Chapter 4.

- **PGLEcmrs.pm**
  This Perl-module is used to parse an input PGA program in the primitive instruction set PGLEcmrs. Because this primitive instruction set is not executable in the PGA Toolset, we first project it to PGLEcrs or PGLEcr by running the Perl-program pglecmrs2pglecrs or pglecmrs2pglecr.
  PGLEcmrs.pm is available from:
  http://www.idealworld.hostrocket.com/Thesis/draft/AppendixC/PGLEcmrs.pm

- **PGLEcrs.pm**
  This Perl-module is used to parse and execute an input PGA program in the primitive instruction set PGLEcrs. The result of executing a PGLEcrs program can be seen in a fluid generated by such a command:
  `gensim -L /PATH -P PGLEcrs -B MSP -v -l filename`
  `-L` is the option to load a module. Substitute the path where this module can be found for `/PATH`. If this module is in the current directory, then `-L` can be omitted.
  PGLEcrs.pm is available from:
  http://www.idealworld.hostrocket.com/Thesis/draft/AppendixC/PGLEcrs.pm

- **pglecmrs2pglecrs**
  This Perl-program is used to project a program in PGLEcmrs to one in PGLEcrs. Substitute the path in the second line of this program with the installation location of the PGA Toolset, such as `use lib '/home/sylviahe/pga-1.1';`
  To run this program:
  `pglecmrs2pglecrs < input > output`
  pglecmrs2pglecrs is available from:
  http://www.idealworld.hostrocket.com/Thesis/draft/pglecmrs2pglecrs

pglecrs2pglecr

This Perl-program is used to project a program in PGLEcrs to one in PGLEcr. Substitute the path in the second line of this program with the installation location of the PGA Toolset, such as `use lib '/home/sylviahe/pga-1.1';`

To run this program:

```
pglecrs2pglecr < input > output
```

This program together with the previous one can be used as a filter in a composite command like:

```
pglecmrs2pglecrs < input | pglecrs2pglecr > output
```

pglecmrs2pglecrs is available from:

http://www.idealworld.hostrocket.com/Thesis/draft/AppendixC/pglecrs2pglecr

- **pglecmrs2pglecr**

  This Perl-program is used to project a program in PGLEcmrs directly to one in PGLEcr. Substitute the path in the second line of this program with the installation location of the PGA Toolset, such as `use lib '/home/sylviahe/pga-1.1';`

  To run this program:

  ```
  pglecmrs2pglecr < input > output
  ```

  pglecmrs2pglecr is available from:

  http://www.idealworld.hostrocket.com/Thesis/draft/AppendixC/pglecmrs2pglecrs

- **MSPeame.pm**

  This Perl-module is used to parse an input PGA program in the basic instruction set of MSPeame and output the execution result from another Perl-module MSPeamecore.om. It is an extension of MSPea.

  It can be used in a command like this:

  ```
  gensim -P PGLEcr -L /PATH -B MSPeame -v -l filename
  ```

  `-L` is the option to load a module. Substitute the path where this module can be found for `/PATH`. If this module is in the current directory, then `-L` can be omitted.

  MSPeame.pm is available from:

  http://www.idealworld.hostrocket.com/Thesis/draft/AppendixC/MSPeame.pm

- **MSPeamecore.pm**

  This Perl-module is used to execute the MSPeame instructions. The results are sent to the Perl-module MSPeame.pm and outputted by that module.

  MSPeamecore.pm is available from:

  http://www.idealworld.hostrocket.com/Thesis/draft/AppendixC/MSPeamecore.pm