**"Vasile Alecsandri" University of Bacău**
**Faculty of Sciences**
**Scientific Studies and Research**
**Series Mathematics and Informatics**
**Vol. 19 (2009), No. 2, 393 - 402**

# THE RODIN TECHNICAL REPORT COMPUTING THE EXPRESSION OF A PSEUDOCONSTRUCTOR OVER MONADIC VALUES USABLE AS MODULAR SEMANTIC AUTOEVALUATOR BY EQUATIONAL REWRITING

DAN POPA

**Abstract.** The paper focuses on the act of computing the expression of a pseudoconstructor over monadic values (actions) - usable as a modular semantic autoevaluator - by using equational rewriting. After that, *the syntax is represented by it's semantic.* This paper is a part of The Rodin Technical Report.

## 1. INTRODUCTION

This paper is showing on the steps of creating the semantic of a new statement for a modular language (called Rodin [5],[6],[7] which is in fact a platform for modular language development). The pragmatic goal of this paper is to practically answer a simple question: How can we add a Pascal Like "for" - statement semantics (expressed by syntax) in a C-like language which is rebuilt using the modular monadic semantics introduced by Popa in [4]. We are concentrating here on the development of a piece of that specific modular monadic semantic using equational reasoning. A term over a set of monadic pseudoconstructors [4] provided by the Rodin [12] platform is build, in the pages below, step by step. *Finally the equation which links the abstract syntax with the just developed semantic is simply pasted into the module and compiled as part of the project, becoming immediately usable.*

**Keywords and phrases: modular monadic semantics, interpreters, *pseudoconstructor, autoevaluator,(itselfevaluator),*The Expression Problem**
**AMS (2000) Mathematical Subject Classification:** 68Q55, 68N20, 68Q42

The Rodin Platform for language development is a system able to be used in the development of modular languages - (i.e. languages which can be built by compiling together a set of modules.) Each module includes a modular parser built using the parser combinators from a common used library (Parsec [1],[2] is used but Parselib an older Haskell library [10],[11] and other libraries are also possible choices.) and a semantic description associated to the abstract syntax produced by the parser, creating an equation.

The simplest pieces are in fact semantic atoms called pseudoconstructors [4] over monadic values - because they are functions from monadic values to monadic values. Larger syntax structures have their own pseudoconstructors built in a compositional constructive way. This paper is explaining how to built such a complex term - the pseudoconstructor of a Pascal-like "for " statement.

## 2. THE SYNTAX

The actual syntax we choose to implement is this one, with usual notations and a positive step.

FOR <var> := <exp1> TO/DOWNTO <exp2> STEP <step> DO <com>

Correspondingly, the abstract syntax of this loop (expressed with a pseudoconstructor *forPas*) is:

forPas num e1 semn e2 step com = ....

where

num - is the identifier of the counter variable
e1 - an expression which gave us the first value of the counter
semn - is just a +1 or a -1 corresponding with the semantic of the keyword "To or Downto" used
e2 - expression which gave us the final value
step - the step (should be positive)
com - the statement to be executed repeatedly, called "command 1"

## 3. THE SEMANTICS

A possible non standard operational semantic of this statement is here described inspired by the final paragraph (pp 71) from this e-book [3] concerning the Oberon-2 language. With some minor modifications (which are in fact alpha conversions, like in lambda calculus) and *considering the fact that the step = zero is forbidden by the syntax*, the semantics may looks like

here:

```
v := <exp1>;
temp := <exp2>;
IF <step>  > 0 THEN
   WHILE v <= temp DO <com>; v := v + <step> END
ELSE
   WHILE v >= temp DO <com>; v := v + <step> END
END
```

where *semn* = 1 in cases when TO was used, respectively *semn* = -1 in cases when TO was used, so we can consider the following:

```
v := <exp1>;
temp := <exp2>;
IF <semn> =  1 THEN
   WHILE v <= temp DO <com>; v := v + <step> END
ELSE
   WHILE v >= temp DO <com>; v := v + <step> END
END
```

Let's note the main structure: it is a sequence: the first assignment followed by the rest of the code.

Parsing may be used to reveal the syntax tree and all the visible structures and substructures. So we can write our new semantic as:

(segv i1 i2 ) - where i1 and i2 will be two pseudoconstructors implemented as monadic actions

where i1 will implement the first assignment and i2 will implement the rest: i.e. the entire block composed by the second assignment and the if statement.

So we can write them as:

```
i1 = (attrib2 n1 e1)
i2 = (segv  i3   i4 )
```

And by the replacement (substitution) in the first term we get:

```
(segv  (attrib2 n1 e1)
       (segv  i3
              i4 )      )
```

Remark: the names of the two variables are chosen to be n1 and n2 (i.e. abbreviations of name #1 and name #2).

The variable n1 will have the real name gave by the programmer in order to make the variable usable from the inside of the body of the loop. The second name is hidden and is automatically generated starting from the first

one, by adding the "$" prefix. Because it is not actually an identifier the syntax of the language will forbid it's use by the programmer.

```
n1=num
n2=("$"++num)
```

Carefully looking to the structure, we are able to write the third and the fourth statement:

```
i3 = (attrib2 n2 e2 )
i4 =  (cond (eq semn unu)
            (while (le (var n1) (var n2))
                    (segv com
                            (attrib2 n1 (plus  step (var n1) ) )
                     )
             )
             (while (ge (var n1) (var n2))
                    (segv com
                            (attrib2 n1 (minus (var n1) step ) )
                     )
              )
       )
```

The descriptions is using a set of standard semantic primitives of the Rodin Project: cond, eq, while, le, var , segv, attrib2, plus, ge, minus having sugestive names. (See the annexed text.)

Replacing i3 and i4 in the main term we get a longer one:

```
(segv  (attrib2 num e1)
       (segv (attrib2 n2 e2 )
             (cond (eq semn unu)
                    (while (le (var n1) (var n2))
                            (segv com
                                    (attrib2 n1 (plus  step (var n1) ) )
                             )
                    )
                    (while (ge (var n1) (var n2))
                            (segv com
                                   (attrib2 n1 (minus (var n1) step ) )
                            )
                    )
              )
```

```
        )
)
```

Now it's the moment to substitute all the appearances of n1 and n2 by their description, each of them as term:

```
 n1=num
 n2=("$"++num)
```

and we get this:

```
(segv  (attrib2 num e1)
        (segv  (attrib2 ("$"++num) e2 )
                (cond (eq semn unu)
                        (while (le (var num) (var ("$"++num)))
                                (segv com
                                        (attrib2 num (plus  step (var num) ) )
                                )
                        )
                        (while (ge (var num) (var ("$"++num)))
                                (segv com
                                        (attrib2 num (minus (var num) step ) )
                                )
                        )
                )
        )
)
```

Finally, we have to replace 'unu' by the semantic description of the constant 1 represented by a pseudoconstructor with it's argument (con 1) because

```
  unu = (con 1)
```

And we can write the equation linking the abstract syntax with it's semantic (which is expressed by composing pseudoconstructors over monadic values):

```
forPas num e1 semn e2 step com
 = (segv   (attrib2 num e1)
           (segv (attrib2 ("$"++num) e2 )
                   ( cond (eq semn (con 1))
                           (while (le (var num) (var ("$"++num)))
                                   (segv com
                                           (attrib2 num (plus  step (var num) ) )
```

```
                        )
                    )
                    (while (ge (var num) (var ("$"++num)))
                        (segv com
                            (attrib2 num (minus (var num) step ) )  –
numele se repeta !
                        )
                    )
                )
            )
    )
```

In fact we are expressing the syntax as it's corresponding semantic.

**The final equation: syntax = semantic**

We was starting from the syntax:

FOR <var> := <exp1> TO/DOWNTO <exp2> STEP <step> DO <com>

And the corresponding abstract syntax which is (considered as a subtree):

forPas num e1 semn e2 step com

and by expressing the syntax by a composition of monadic pseudoconstructors following the pattern of the informal semantics we have got the equation which is directly usable as part of one Haskell module of the project. The remaining previous part of that module is the modular parser and we had detailed the construction of such a parser in [7].

```
forPas num e1 semn e2 step com
 = (segv   (attrib2 num e1)
          (segv  (attrib2 ("$"++num) e2 )
                 (cond (eq semn (con 1))
                       (while (le (var num) (var ("$"++num)))
                              (segv com
                                    (attrib2 num (plus  step (var num) ) )
                              )
                       )
                       (while (ge (var num) (var ("$"++num)))
                              (segv com
                                    (attrib2 num (minus (var num) step ) )
```

```
                    )
                )
            )
        )
    )
```

**Next step: Compiling and testing**

Now, we can copy this equation, paste it into the module of the project, and compile it using GHC - the Glasgow Haskell Compiler. The session looks like here, below:

```
[dan@localhost ExperimentExp13]$ date
ven. oct. 30 11:45:26 EET 2009
[dan@localhost ExperimentExp13]$ ghc --make Main.hs
[27 of 31] Compiling ModPforPas       ( ModPforPas.hs,
ModPforPas.o )
Linking Main ...
[dan@localhost ExperimentExp13]$
```

After the compilation the binary of the interpreter, which includes the semantic of this new statement, is ready to be used.

Please note, even if Rodin is a Didactic Programming Language using romanian keywords, the syntax used by this plugin is actually written using usual english keywords.

This way it will be:

- simple for the english reader to see and understand it
- simple for the romanian reader to notice it
- readable by any programmer

## 4. CONCLUSIONS

The goal of producing the expression of a pseudoconstructor over monadic values (actions) - usable as a modular semantic autoevaluator - by using equational rewriting reasoning was achieved.

The final equation is immediately usable as a program, being the second part of the Haskell module which defines such command. This demonstration was made at ICMI-2 / 2009, in Bacău and other supplementary data are provided via the Rodin Project Website [12]: http://www.haskell.org/haskellwiki/Rodin

**Annexa A**:

Some small programs running with Rodin codename Experiment Exp13. The text was simply get with copy and pasted here below. The romanian words included are parts of the usual output of the system and parts of the Rodin language syntax.

```
Programul:./forpas1.txt
{fie i=1000 ;
 for i:=1 to 10 step 1 do
  { scrie i }
}

1
2
3
4
5
6
7
8
9
10

Programul a rulat !
Modular Language written by Dan V Popa, Ro/Haskell Group.
http://www.haskell.org/haskellwiki/Rodin/Download
sept/2009  -      Rodin       - Codename:ExperimentExp13
---------------------------------------------------------------------
Programul:./forPas20.txt
{fie i=0 ;
 for i:=1 to 20 step 2 do
  { scrie i }
}

Programul:./forpas1.txt

1
3
5
```

7
9
11
13
15
17
19

**Annex B:**

Here is the set of modular monadic semantic primitives which I have written as part of The Rodin Project which are used (Full code not included, due to space limitations).

| | |
|---|---|
| cond | - the conditional |
| eq | - the equality |
| while | - the while loop |
| le | - less equal (<=) |
| var | - variable, followed by the identifier, its single parameter |
| segv | - sequence of (at least two) statements |
| attrib2 | - assignment (one kind, there are more) |
| plus | - the addition (+) |
| ge | - greather equal (>=) |
| minus | - the substraction (-) |

The parameters of this kind of pseudoconstructors are easily seen from the example, also they are matching the corresponding syntactic structure.

Acknoledgements: I have to send a "Thank you!" to prof. Philip Wadler for his comments, including those above. He was the first person who noticed the main idea of this solution: the abstract syntax is directly replaced by monadic semantics. Also a warm “Thank You !” addressed to the editorial team of “Studia”, for suggestions and feed-back.

## References

[1] Daan Leijen, Erik Meijer, **Parsec: Direct Style Monadic Parser Combinators For The Real World**, DRAFT, October 4, 2001 http://www.haskell.org/haskellwiki/Parsec.html

[2] Daan Leijen, **Parsec, a fast combinator parser**, University of Utrecht, 4 Oct 2001, http://legacy.cs.uu.nl/daan/download/parsec/parsec.html, http://www.cs.uu.nl/~daan
[3] Dan Popa, **Pascalul mileniului al III-lea, Programarea calculatoarelor in Oberon-2,** Edusoft, Bacău, 2005, ISBN 973-87496-8-9
[4] Dan Popa, **Modular evaluation and interpreters using monads and type classes in Haskell**, Studii şi Cercetări Ştiinţifice, Seria Matematică, Universitatea din Bacău, Nr. 18 (2008), pag. 233 – 248.
[5] Dan POPA, **The Rodin Modular Language vers. 8 aug 2009 – User Manual and Report**, OPEN SOURCE SCIENCE JOURNAL (1/2009) ISSN 2066 – 740X , Info. ASE Bucureşti
[6] Dan Popa, **The Rodin Modular Language - vers 21 aug 2009 - User Manual and Report**, "Gheorghe Vrănceanu" International Conference on Mathematics and Informatics, Second Edition, 8-10 sept 2009, Bacău, România
[7] Dan Popa, **Rodin. Technical Report**, Gheorghe Vrănceanu International Conference on Mathematics and Informatics, Second Edition, 8-10 sept 2009, Bacău, Romania
[8] Dan Popa, **Practica Interpretării Monadice**, MatrixRom, Bucureşti, 2008, ISBN 978-973-755-417-8
[9] Dan Popa, **Introducere în Haskell 98 prin exemple**, Edusoft, Bacău, 2007, ISBN 978-973-8934-48-1
[10] - The Haskell Org Community – www.haskell.org
[11] - The Ro/Haskell Community – www.haskell.org/haskellwiki/Ro/Haskell
[12] - The Rodin Community – www.haskell.org/haskellwiki/Rodin

Dan Popa
Department of Mathematics and Informatics
Faculty of Sciences
"Vasile Alecsandri" University of Bacău
Spiru Haret 8, 600114 Bacău, Romania
e-mail: danvpopa@ub.ro