University of Amsterdam

Theory of Computer Science

Simulation of the Eval Instruction from MSPea

B. Diertens

Report TCS1503 May 2015

B. Diertens

section Theory of Computer Science
Faculty of Science
University of Amsterdam

Science Park 904
1098 XH Amsterdam
the Netherlands

tel. +31 20 525.7593
e-mail: B.Diertens@uva.nl

Theory of Computer Science Electronic Report Series

Simulation of the Eval Instruction from MSPea

Bob Diertens

section Theory of Computer Science, Faculty of Science, University of Amsterdam

ABSTRACT

We describe the implementation of the stepwise evaluation of programs encoded as molecules in the simulators of the PGA Toolset. This implementation allows for interactive simulation, nesting of evaluation instructions, and for the concurrent execution of evaluation instructions.

Keywords: program algebra, simulation, evaluation

1. Introduction

In [1], Program Algebra (PGA) is introduced. Here, simple program notations are given which are build up from primitive instructions with as parameter a set of basic instructions that perform some operation on a environment. In [3] the development of a toolset for PGA is described. This toolset contains among others a simulator and sets of primitive instructions and sets of basic instructions. A particular set of basic instruction are the Molecular Programming Primitives (MPP) described in [2] and operate on an environment called Molecular Dynamics.

In [4] we introduced the basic instructions set MSP (Molecular Scripting Primitives) that are based on MPP. This basic instruction set is an attempt to make the task of programming in the PGA setting a little bit easier. In the same report we also introduced the basic instruction MSPea (MSP with eval and apply) as an extension of MSP. This variant contains instructions to compile a program contained in a string to a molecule and to evaluate the molecule.

The evaluation instruction is executed as a single instruction, meaning that the whole program encoded in the molecule is executed in one go. Instead of execution as a single instruction we like to simulate it by stepping through its execution. That the evaluation instructions is executed as one big instruction instead of in steps, is because of the current setup of the simulator. This setup consist of a primitive instruction set simulator and a machine (core) that has as instruction set the basic instructions.

In this report we describe how we implemented the stepwise evaluation of programs encoded as molecules in the simulators of the PGA Toolset.

2. PGA Toolset

In this section we give a short description of parts of the PGA Toolset that are essential for this report.

2.1 Simulator

The modular structure of the simulator is shown in Figure 1. It is a generic simulator in that the terms Primitive and Basic are parameters of the system and represent the primitivie and basic instruction sets the simulator must be given as arguments at startup. In this modular structure the module Kernel takes care of the actual simulation. The module Primitive contains all operations on an instruction of the primitive instruction set. This module use the module Basic for operations on an instruction of the basic instruction set. The module Basic uses the module Basiccore for execution of the basic instructions. This splitting of the modules Basic and Basiccore is necessary for the parallel simulator of the PGA Toolset. Several programs can be loaded into the simulator. Storage and switching between them is taken care of by the

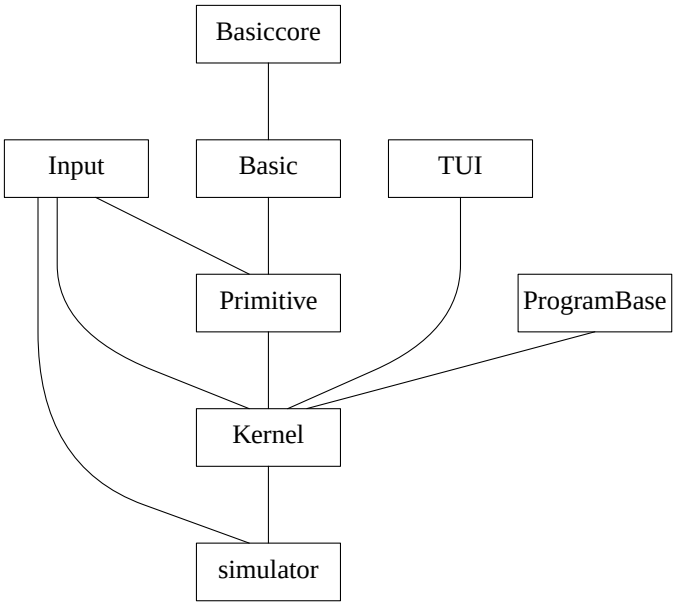

Figure 1. Model of the simulator

module ProgramBase.

2.2 MSPea

We give an overview of the instructions in the basic instruction set MSPea.

`compile` *`extfocus`*
: Compiles the string selected by *`extfocus`* into a molecule and assigns it to *`extfocus`*. It is capable of compiling strings representing programs in the primitive instruction set PGLA with MSP as basic instruction set. Returns `false` when there are errors detected.

`apply` *`extfocus`*
: If *`extfocus`* is a string, it is executed as a basic instruction and returns the value returned by the basic instruction. And otherwise it fails.

`eval` *`extfocus`*
: If *`extfocus`* is a string, it compiles the string into a molecule, after which the molecule is evaluated. If *`extfocus`* is an atom, it is evaluated. And otherwise evaluation fails. It does no assignment of the molecule to *`extfocus`* like `compile` does. The returned value of `eval` is the returned value of the last evaluated basic instruction.

Compilation results in a molecule that represents a null-terminated list of instructions connected by `next` fields. The instructions are represented by an atom with additional fields. Below follows a list of possible instructions.

end
: This instruction consists of an atom with an `end` field, and stops the evaluation.

goto
: Consists of an atom with a `goto` field which selects the next instruction to be evaluated.

test
: Consists of an atom with the fields `test`, `basic`, `T`, and `F`. On the returned value of the basic instruction, selected by the `basic` field, either the `T` (`true`) field or the `F` (`false`) field selects the next instruction to be evaluated.

basic
: Consists of an atom with the field `basic` which selects a basic instruction to be evaluated. Evaluation continues with the next instruction in the list.

The basic instructions are represented by a string. For compilation the basic instruction may be anything, but for evaluation it has to be a MSP instruction. In evaluation, if a string is not recognized as a MSP instruction `false` is returned as the result of this instruction.

Consider the following program, that counts until 10 and then stops:

```
x = 0;
incr x;
+ x == 10;
!;
\\#3
```

We can put this program in a string with the instruction (note the escaping of the backslashes):

```
count = "x = 0; incr x; + x == 10; !; \\\\#3"
```

Compilation of `count` gives the molecule in Figure 2.

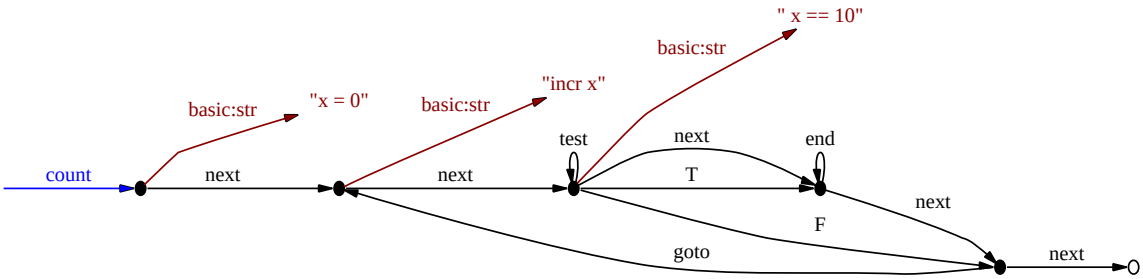

Figure 2. Result of compilation of `count`

3. Simulating evaluation

In order to take the eval instruction form the core we have to provide a mechanism that gives the same functionality. We could build it in the current setup of the Kernel, but then we have to adjust every module that executes a set of primitive instructions. That is against the generic nature of our simulator in which the primitive instruction set and the basic instruction set are two parameters for the simulator. The solution is to have a separate kernel that simulates the eval instruction. Below we describe this solution with more detail. Furthermore we indicate some difficulties with this implementation and describe how we solved these difficulties.

3.1 Evaluation kernel

To provide for a separate kernel for the execution of the eval instruction we take the eval instruction from the core (MSPeacore) and replace it with a routine in the module MSPea. We can use the same algorithm as that was used for the routine in the core, but instead of using facilities available in the core we have to use the instructions that are executed by the core.

The evaluation kernel uses the same step and run/stop mechanism as the normal kernel for execution instruction. This makes the stepwise execution of the eval instruction possible.

3.2 User interactions

Apart from the step and run/stop mechanisme, we like to have the other user interactions as well during the execution of an eval instruction. But some of these interaction disrupt the execution of the eval instruction. The solution is to divide the interactions in disruptive and non-disruptive interactions. The non-disruptive

interactions are handled during an evaluation. When a disruptive interaction takes place the evaluation kernel stops and the handling of the interaction is left to the standard kernel.

3.3 Nested eval instructions

To deal with nested eval instructions we have to make a few adjustments. First, we have to recognize an eval instruction in our evaluation kernel. Second, we have to convert a reference to the molecule to be evaluated from a string to field selection. And last, we have to keep track of the locations in the molecules where evaluation has to continued. Below we describe the adjustments.

We adjust the compiler so that it recognizes an eval instruction in the strings that old a basic instruction. Instead of making a field *basic* that points to a string, we now let the field *basic* point to an atom that has a field *eval* that points to string holding the reference to the molecule to be evaluated. The evaluation kernel can now check for a field *eval* in the field *basic* and take the proper actions.

We introduce a core instruction for setting a reference to a molecule that is to be evaluated.

SETREF <focus> <extfocus:string>

This instruction takes the string denoted by the second argument. The contents of the string is converted to an extended focus that is supposed to indicate a molecular structure to be evaluated. The extended focus is assigned to the focus in the first argument.

We introduce core instruction for pushing elements on and popping elements of a stack.

PUSH <focus:stack> <focus:item>
POP <focus:stack> <focus:item>

We use these instruction for storing and retreiving the program counter that points to the location in a molecule where the evaluation has to continue.

3.4 Concurrent execution

In the PGA Toolset we have simulators that are able to execute instructions in concurrency. Actually these consist of several instances of a simulator that operate on a single core. This means that it is possible that two or more evaluation kernels are active at the same time. To make this work properly each evaluation kernel has to have its own environment variables (like program counter, stack, etc) in the core for processing the eval instruction. For this reason, every simulator gets his own unique identifier that the evaluation kernel can use in the names for the environment variables.

3.5 Tracing

As we have taking the eval instruction from the core and replaced it by an evaluation kernel in MSPea, we can easily add the displaying of tracing information to the user. But we cannot show all information. We can show which primitive instruction is executed and we can show the boolean result of the execution of a basic instruction. But we cannot show the basic instruction itself that is executed, as this information is only available in the core and we have no way to retreive this inforamtion from the core.

4. Conclusions

We have described the implementation for the eval instruction from MSPea that can be simulated in a stepwise manner by he simulators in the PGA Toolset. This implementation makes it possible to interactively simulate a program encoded as molecular structure. It also allows for nested use of the eval instruction and for the concurrent execution of the eval instruction.

References

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

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

[3] B. Diertens, ‘‘A Toolset for PGA,’’ report PRG0302, Programming Research Group - University of Amsterdam, 2003.

[4] B. Diertens, ‘‘Molecular Scripting Primitives,’’ report PRG0401, Programming Research Group - University of Amsterdam, 2004.

Electronic Reports Series of section Theory of Computer Science

Within this series the following reports appeared.

[TCS1502] J.A. Bergstra, *Architectural Adequacy and Evolutionary Adequacy as Characterics of a Candidate Informational Money*, section Theory of Computer Science - University of Amsterdam, 2015.

[TCS1501] B. Diertens, *Composition in the Function-Behaviour-Structure Framework*, section Theory of Computer Science - University of Amsterdam, 2015.

[TCS1301v2] B. Diertens, *Refinement in the Function-Behaviour-Structure Framework (version 2)*, section Theory of Computer Science - University of Amsterdam, 2015.

[TCS1410v2] J.A. Bergstra and A. Ponse, *Division by Zero in Common Meadows (version 2)*, section Theory of Computer Science - University of Amsterdam, 2014.

[TCS1414] J.A. Bergstra, *From Softwae Crisis to Informational Money*, section Theory of Computer Science - University of Amsterdam, 2014.

[TCS1413] J.A. Bergstra and I. Bethke, *Note on Paraconsistency on the Logic of Fractions*, section Theory of Computer Science - University of Amsterdam, 2014.

[TCS1412] J.A. Bergstra, I. Bethke, and A. Ponse, *Rekenen-Informatica*, section Theory of Computer Science - University of Amsterdam, 2014.

[TCS1411] J.A. Bergstra, *Bitcoin: not a Currency-like Informational Commodity*, section Theory of Computer Science - University of Amsterdam, 2014.

[TCS1409v2] J.A. Bergstra and A. Ponse, *Three Datatype Defining Rewrite Systems for Datatypes of Integers each extending a Datatype of Naturals (version 2)*, section Theory of Computer Science - University of Amsterdam, 2014.

[TCS1410] J.A. Bergstra and A. Ponse, *Division by Zero in Common Meadows*, section Theory of Computer Science - University of Amsterdam, 2014.

[TCS1407v3] J.A. Bergstra, *Four Complete Datatype Defining Rewrite Systems for an Abstract Datatype of Natural Numbers (version 3)*, section Theory of Computer Science - University of Amsterdam, 2014.

[TCS1409] J.A. Bergstra and A. Ponse, *Three Datatype Defining Rewrite Systems for Datatypes of Integers each extending a Datatype of Naturals*, section Theory of Computer Science - University of Amsterdam, 2014.

[TCS1406v3] J.A. Bergstra, *Bitcoin and Islamic Finance (version 3)*, section Theory of Computer Science - University of Amsterdam, 2014.

[TCS1407v2] J.A. Bergstra, *Four Complete Datatype Defining Rewrite Systems for an Abstract Datatype of Natural Numbers (version 2)*, section Theory of Computer Science - University of Amsterdam, 2014.

[TCS1408] J.A. Bergstra, *Bitcoin: Informational Money en het Einde van Gewoon Geld*, section Theory of Computer Science - University of Amsterdam, 2014.

[TCS1407] J.A. Bergstra, *Four Complete Datatype Defining Rewrite Systems for an Abstract Datatype of Natural Numbers*, section Theory of Computer Science - University of Amsterdam, 2014.

[TCS1406v2] J.A. Bergstra, *Bitcoin and Islamic Finance (version 2)*, section Theory of Computer Science - University of Amsterdam, 2014.

[TCS1406] J.A. Bergstra, *Bitcoin and Islamic Finance*, section Theory of Computer Science - University of Amsterdam, 2014.

[TCS1405] J.A. Bergstra, *Rekenen in een Conservatieve Schrapwet Weide*, section Theory of Computer Science - University of Amsterdam, 2014.

[TCS1404] J.A. Bergstra, *Division by Zero and Abstract Data Types*, section Theory of Computer Science - University of Amsterdam, 2014.

[TCS1403] J.A. Bergstra, I. Bethke, and A. Ponse, *Equations for Formally Real Meadows*, section Theory of Computer Science - University of Amsterdam, 2014.

[TCS1402] J.A. Bergstra and W.P. Weijland, *Bitcoin, a Money-like Informational Commodity*, section Theory of Computer Science - University of Amsterdam, 2014.

[TCS1401] J.A. Bergstra, *Bitcoin, een "money-like informational commodity"*, section Theory of Computer Science - University of Amsterdam, 2014.

[TCS1301] B. Diertens, *The Refined Function-Behaviour-Structure Framework*, section Theory of Computer Science - University of Amsterdam, 2013.

[TCS1202] B. Diertens, *From Functions to Object-Orientation by Abstraction*, section Theory of Computer Science - University of Amsterdam, 2012.

[TCS1201] B. Diertens, *Concurrent Models for Object Execution*, section Theory of Computer Science - University of Amsterdam, 2012.

[TCS1102] B. Diertens, *Communicating Concurrent Functions*, section Theory of Computer Science - University of Amsterdam, 2011.

[TCS1101] B. Diertens, *Concurrent Models for Function Execution*, section Theory of Computer Science - University of Amsterdam, 2011.

[TCS1001] B. Diertens, *On Object-Orientation*, section Theory of Computer Science - University of Amsterdam, 2010.

The above reports and more are available through the website: ivi.fnwi.uva.nl/tcs/

Electronic Report Series

section Theory of Computer Science
Faculty of Science
University of Amsterdam

Science Park 904
1098 XG Amsterdam
the Netherlands

ivi.fnwi.uva.nl/tcs/