University of Amsterdam
Theory of Computer Science

Molecular Scripting Primitives with Functies

B. Diertens

Report TCS1504 June 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

Molecular Scripting Primitives with Functions

Bob Diertens

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

ABSTRACT

We extend the basic instruction set MSPea (Molecular Scripting Primiteves with eval/apply complex) with functions. The functions are compiled and encoded as molecules, which can be evaluated in a stepwise manner. The implementation allows for recursive functions calls as well as the concurrent execution of functions. The purpose is to aid in further study of functions and their executions and in education.

Keywords: program algebra, simulation, evaluation, functions

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 is 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.

In [5] we described the implementation for the simulation of the eval instruction from MSPea. This implementation allows for stepwise simulation of the evaluation of molecular structures. In this report we describe the extension of MSP with functions (MSPfunc). The implementation is based on the evaluation instruction from MSPea. It follows the commonly used computational model by compilers for imperative programming languages.

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

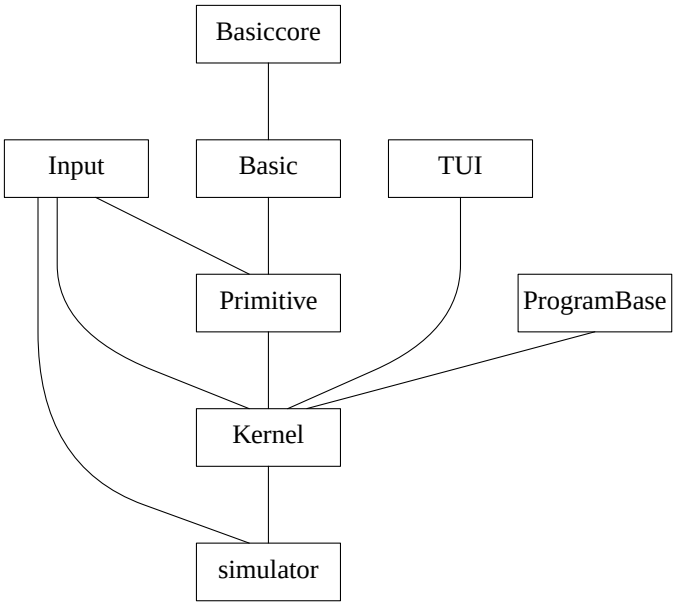

Figure 1. Model of the simulator

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. In the case the basic instruction is an eval instruction, the filed basic points to an atom that has a field

eval that points to a string containing what has to be evaluated. In the other cases, the field basic points to a string containing the basic instruction. Evaluation continues with the next instruction in the list.

The basic instructions are represented by a string, or in the case of an eval instruction the basic field points to an atom that has a field eval that points to a string. For compilation the basic instruction may be anything, but for evaluation it has to be a MSPea instruction. In evaluation, if a string is not recognized as a MSPea 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.

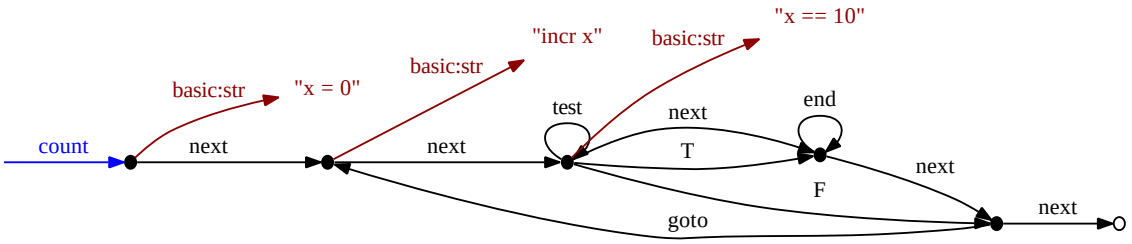

Figure 2. Result of compilation of `count`

3. MSP with functions

In this section we describe the extension of MSP with functions and its implementation in the PGA Toolset. The functions are based on the evaluation of molecular structures from MSPea. We consider such a molecular structure a function without parameters, local variables, and return value. The recursive evaluation of molecular structures is implemented using a stack. This stack is located in the fluid together with the molecular structures. We can use the stack in the implementation of environments in which the functions are executed.

3.1 Instructions

To define functions in a program MSPfunc has the following instructions.

`function` *`name`*`(`*`parameters`*`) body`
`function` *`name`*`(`*`parameters`*`):`*`type body`*

Defines the function *`name`* with optional *`parameters`* and optional return *`type`*. In the case of no return type or when the return type is of type atom, *`type`* is left out (including the ':'). The *`body`* must be string denoting the body of the function in the programming language PGLA with MSPfunc. The *`body`* is compiled into a molecule and assigned to the focus *`name`*. The parameters are added to this molecule. The *`parameters`* must be a comma separated list in which each parameter is denoted as either *`identifier`* or *`identifier`*`:`*`type`*. Returns false when there are errors detected in the compilation of the function.

`return` *`object`*
`return` *`value`*

Sets the value for the function to be returned in a funcation call. The `return` statement does not

end the execution of the function. Returns false when a value of the wrong type is returned.

In the body of a function a parameter can be referenced by `<>.identifier`. Local variables can be added with `<>.+identifier` and `<>.identifier:type`, and referenced in the same way as the parameters. The `<>` indicates the environment in which the function is executed. The parameters as well as the arguments are fields in this environment.

A function can be called with one of the following instructions.

name(*arguments*)
: Executes the function *name* by binding the *arguments* to the parameters and evaluating the molecule in the focus *name*. Returns false when an error occurred in the binding of the arguments to the parameters, and otherwise returns the return value of the last evaluated basic instruction.

object = *name*(*arguments*)
: Executes the function *name* by binding the *arguments* to the parameters and evaluating the molecule in the focus *name*. After execution the value in the return statement is assigned to *object*. Returns false when an error occurred in the binding of the arguments to the parameters, and returns false when the return value of the last evaluated basic instruction is false, and otherwise it returns the return value of the assignment of the returned value by the function to *object*.

3.2 Implementation

The implementation for MSPfuncs consists of a few extensions of the implementation for MSPea.

3.2.1 Compilation

The compiler has the possibility to hook in a routine for adding basic instructions. Such a routine can make different settings for the basic fields that normally contain the basic instructions as a string. The different settings are necessary for the evaluation routine. For MSPfunc the following settings for the field basic are made.

function call
: The field basic points to an atom that has a field call that points to an atom containing the fields func of the type string and args of the type string.

function call with return value
: The same as above, but the field call has an extra field ret of the type string.

return
: The basic field points to an atom which ahs a field return of type string.

Consider the following program that defines the function add3.

```
function add3(x:int, y:int, z:int):int "
<>.+t:int; <>.t = add(<>.x, <>.x);
<>.t = add(<>.t, <>.z); return <>.t;
!";
```

Execution of this program results in the molecule shown in Figure 3.

3.2.2 Evaluation

The evaluation kernel has the possibility to hook in a routine for evaluating basic instructions. We use this construction to hook in a routine that takes care of evaluating the instructions from the our basic instruction set MSPfunc. To support this evaluation the following instructions are added to the core.

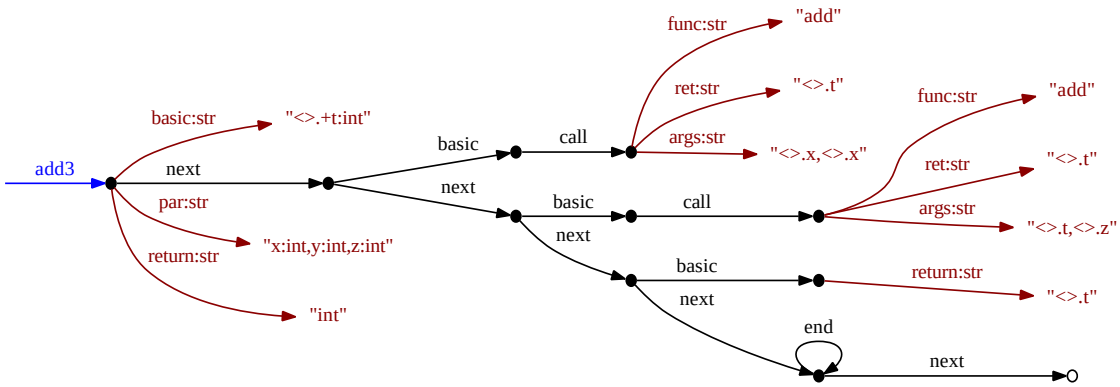

Figure 3. Result of the definition of function `add3`

ADDTYPEPAR <focus:function> <string:type> <list:parameters>
Adds type and parameters to the molecule of the compiled function.

BINDPAR <focus:environment> <focus:function> <list:arguments>
Binds arguments for the function to the parameters and making them availabel in the environment.

BINDPARSTR <focus:environment> <focus:function> <extfocus:arguments:string>
As above, but now the arguments are taking from a string.

SETFUNCRETURN <focus:environment> <extfocus:value>
Sets the return value for a function.

GETFUNCRETURN <focus:environment> <extfocus:variable>
Gets the return value for a function.

The above instructions take care of the necessary replacement of '<>' occurences with a reference to the current environment in which the function is executed.

4. Conclusions

We have described the basic instruction set MSPfunc and its implementation in the PGA Toolset. The functions can be simulated in a stepwise manner by the simulators in the PGA Toolset. The implementation allows for recursive functions calls and for the concurrent execution of functions. As the implementation follows the commonly used computational model by the compilers for imperative programming languages, it can be used for the further study of functions and their execution as well as for educational purposes.

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.

[5] B. Diertens, ‘‘Simulation of the Eval Instruction from MSPea,’’ report TCS1503, section Theory of Computer Science - University of Amsterdam, 2015.

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/