Building a Neural Computer

A compiler and simulator for partial recursive functions over neural networks

Paulo J. F. Carreira, Miguel A. Rosa,
J. Pedro Neto and J. Félix Costa

DI-FCUL TR-98-8

December 1998

Departamento de Informática
Faculdade de Ciências da Universidade de Lisboa
Campo Grande, 1700 Lisboa
Portugal

The technical reports are available at `http://www.di.fc.ul.pt/biblioteca/tech-reports`. The files are stored in PDF format, with the report number as filename. Alternatively, reports are available by post from the above address.

Building a Neural Computer[*†]

A compiler and simulator for partial recursive functions over neural networks

Paulo J. F. Carreira,
paulojfc@caravela.di.fc.ul.pt

Miguel A. Rosa,
mrosa@fc.ul.pt

J. Pedro Neto,
jpn@di.fc.ul.pt

J. Félix Costa
fgc@di.fc.ul.pt

Departamento de Informática
Faculdade de Ciências da Universidade de Lisboa
BLOCO C5 – PISO 1, 1700 LISBOA, PORTUGAL

Abstract

In the work of [Siegelmann 95] it was showed that Artificial Recursive Neural Networks have the same computing power as Turing machines. A Turing machine can be programmed in a proper high-level language - the language of partial recursive functions. In this paper we present the implementation of a compiler that directly translates high-level Turing machine programs to Artificial Recursive Neural Networks. The application contains a simulator that can be used to test the resulting networks. We also argue that experiments like this compiler may give us clues on procedures for automatic synthesis of Artificial Recursive Neural Networks from high-level descriptions.

1. Introduction

The field of Artificial Recursive Neural Networks (ARNN's) is meeting a lot of excitement nowadays. Both because of their achievements in solving real world problems and their simplicity of the underlying principles that still allow them to mimic their biological counterparts. All this excitement attracts people from many different fields such as Neurophisiology, Biology and Computer Science, among others.

We present our work in the Computer Science perspective. The view we are interested in, is the one in which an ARNN can be seen as a computing mechanism able to perform some kind of computation based on a program coded as a specific arrangement of neural artefacts, like neurons and synapses.

This work implements a compiler and a simulator based on the previous *Turing Universality of Neural Nets (Revisited)* [Neto et al. 97a] paper. In [Gruau et al 94], [Siegelmann 96] and [Neto et al. 97b] similar ideas are given but they are based on higher level languages.

To help the user getting into this work we start by giving the underlying theoretical context on which it is based. In section 2, we give a brief review of the partial recursive function theory. In section 3 we present our approach for constructing neural networks from partial recursive functions, together with some minor modifications we introduced. The explanation of how we adapted [Neto et al. 97a]

[*] This work was supported by JNICT PBIC/TIT/2527/95
[†] Completed during May 1998.

theoretical material into a compiler is given in section 4. Section 5 refers to the simulator and usage examples and section 6 concludes this work. The simulator is freely available at http://www.di.fc.ul.pt/~jpn/nwb/nwb.html.

2. Partial Recursive Function theory

When informally speaking of a neural computer, one could be motivated about what could it be like, the language to program such a machine. The language that we will use is the one of partial recursive functions (PRF). Although primitive when compared to modern computer languages, it is simple and powerful enough to program any mechanism with same computing power as a Turing machine w.r.t. the class of functions it can compute. Surely, building complex programs with this language would be very difficult and more appropriate languages exist. For our purposes however, this language is suited.

The PRF theory identifies the set of computable functions with the set of computable functions on $\mathbb{N}$. Such functions are called partial because we do not require them to be defined for every value of the domain. Technically this is accomplished by adjoining a special symbol, "$\perp$", to the function co-domain on which the undefined values are mapped. These functions are also called recursive, because each one can be obtained by recursive application of 3 rules (operations) over the axioms (the base set).
We shall use $\alpha(x_1,\ldots,x_n) \simeq \beta(x_1,\ldots,x_n)$ to denote equality of the expressions $\alpha(x_1,\ldots,x_n)$ and $\beta(x_1,\ldots,x_n)$, if and only if, $\alpha(x_1,\ldots,x_n)$ and $\beta(x_1,\ldots,x_n)$ are defined for any $(x_1,\ldots,x_n)$ or both undefined.

The axioms also called primitive functions are:

- **W** that denotes the zero-ary constant 0;
- **S** that denotes the unary successor function S(x)=x+1;
- **U(i,n)** that for i and n fixed, denotes the corresponding element **U**i,n(x1,..,xn) = xi ($1 \le i \le n$) of the n-ary projection function family.

The construction rules are:

- **C**, that denotes the **composition**. Formally: if $f_1(x_1,\ldots,x_n)$, …, $f_k(x_1,\ldots,x_n)$ and $g(y_1,\ldots,y_k)$ are PRFs, then $h(x_1,\ldots,x_n) \simeq g(f_1(x_1,\ldots,x_n), \ldots, f_k(x_1,\ldots,x_n))$ is also a PRF**;**
- **R,** that denotes the **recursion**. Formally: if $f(x_1,\ldots,x_n)$ and $g(x_1,\ldots,x_n,y,z)$ are PRFs, then the unique function $h(x_1,\ldots,x_n,y)$ defined by

  1) $h(x_1,\ldots,x_n,0) \simeq f(x_1,\ldots,x_n)$ and
  2) $h(x_1,\ldots,x_n,y+1) \simeq g(x_1,\ldots,x_n,y,h(x_1,\ldots,x_n,y))$ is also a PRF;

- **M,** that denotes the **minimalisation**. Formally: if $f(x_1,\ldots,x_n,y)$ is a PRF, then $h(x_1,\ldots,x_n) \simeq \mu_y(f(x_1,\ldots,x_n,y)=0)$ is also a PRF. Where,

$$\mu_y(f(x_1,\ldots,x_n,y)=0) = \left\{ \begin{array}{l} \text{Least y such that } f(x_1,..,x_n,y)=0 \text{ and } \forall z \le y,\ f(x_1,\ldots,x_n,z) \neq \perp. \\ \perp, \text{ there is no such y.} \end{array} \right\}$$

for instance f(x,y)=x+1 is a PRF and is given by the expression `C(U(1,2),S)`. The function f(x,y)=x+y is also a PRF given by the expression `R(U(1,1),C(U(3,3),S)`. In fact, it is shown that every Turing computable function is a PRF. More details on PRF theory can be found in [Cutland 80] and [Boolos 80].

3. Coding PRF into ARNNs

Finding a systematic way of generating ARNN's from given descriptions of PRF's greatly simplifies the task of producing neural nets to perform certain specific tasks. Furthermore, it also gives a proof that neural nets can effectively compute all Turing computable functions as treated in [Neto et al 97a].
In this section we briefly describe the computing rule of each processing element, i.e. each neuron. Further we present the coding strategy of natural numbers to load the network. Finally we will see how to code a PRF into an ARNN.

3.1 *How do the processing elements work ?*

Like in [Neto et al 97a] we make use of σ-processors. In each instant t each neuron i updates its activity x_i in the following non-linear way:

$$x_i(t+1) = \sigma\left(\sum_{j=1}^{N} a_{ij}x_j(t) + \sum_{j=1}^{M} b_{ij}u_j(t) + c_i\right)$$

where a_{ij}, b_{ij} and c_i are rational weights; N is the number of neurons, M the number of input streams u_j; and σ is the continuous function defined below:

$$\sigma(x) = \begin{cases} 0, & x < 0 \\ x, & 0 \le x \le 1 \\ 1, & x > 1 \end{cases}$$

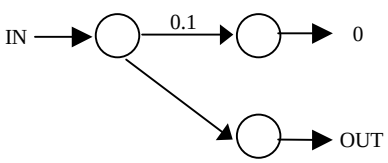
3.2 *How can we represent the numbers ?*

We use an unary representation where each natural number is represented as a rational number by means of a code mapping $\alpha(n) : \mathbb{N} \rightarrow \mathbb{Q}$ where,

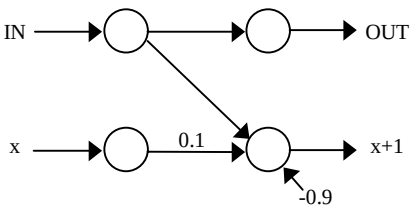
$\alpha(n)$ is given by $\sum_{i=0}^{n} 10^{-i+1}$

3.3 *How to construct the ARNN ?*

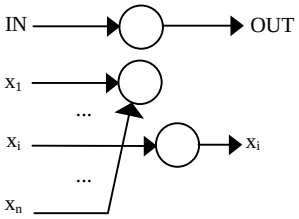
The following three net schemas were implemented to compute the corresponding three axioms of recursive functions theory. Changes were made to [Neto et al 97a] presentation. First, the W axiom is provided with two additional neurons. Second, each U(i,n) axiom is constructed with only three neurons making it more efficient.



(i) W

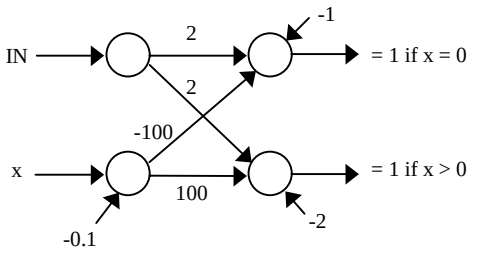


(ii) S

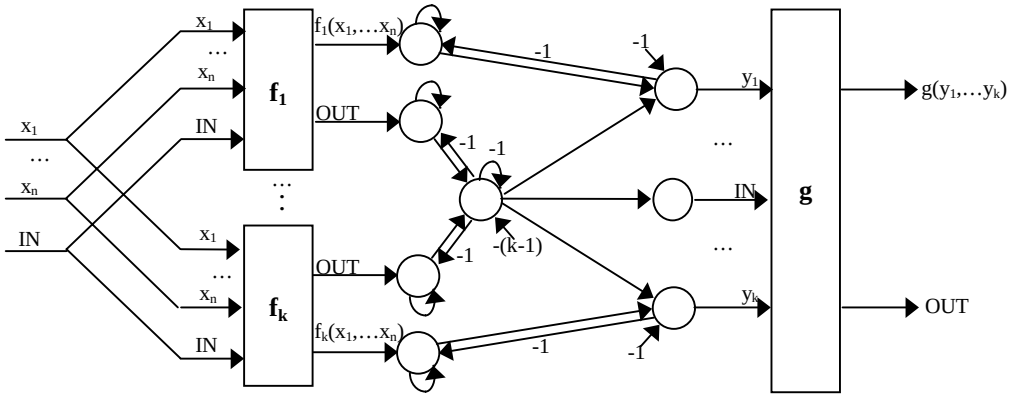


(iii) U(i,n)

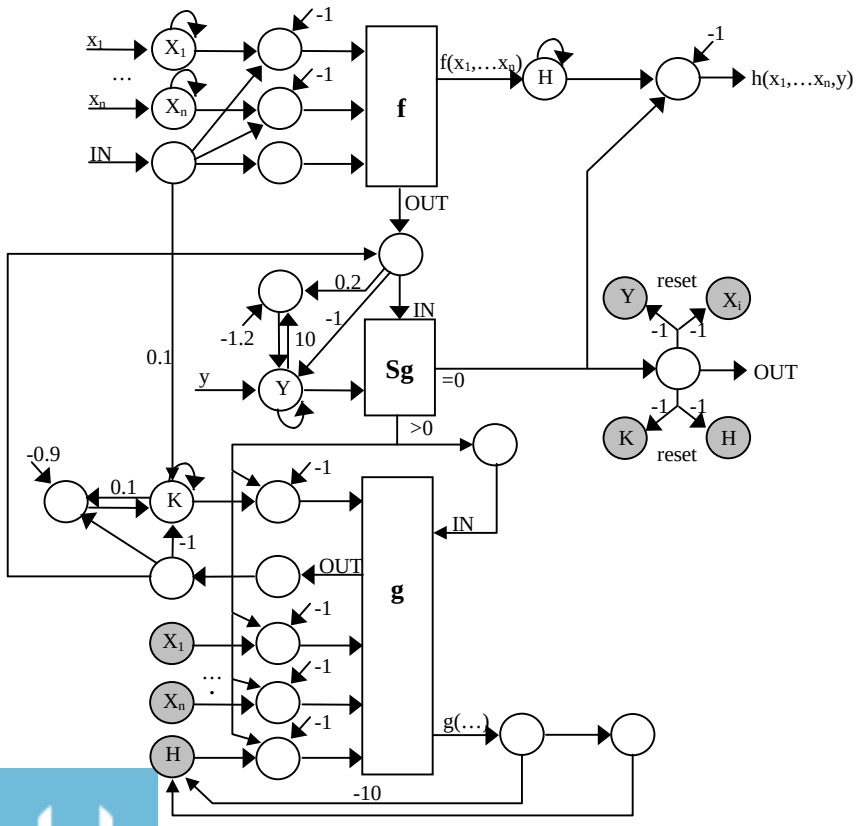
The rules are illustrated by the net schemas of figures v), vi) and vii), where grey coloured circles represent repeated neurons. The Sg box represents a subnet that finds if a given number is positive or zero:



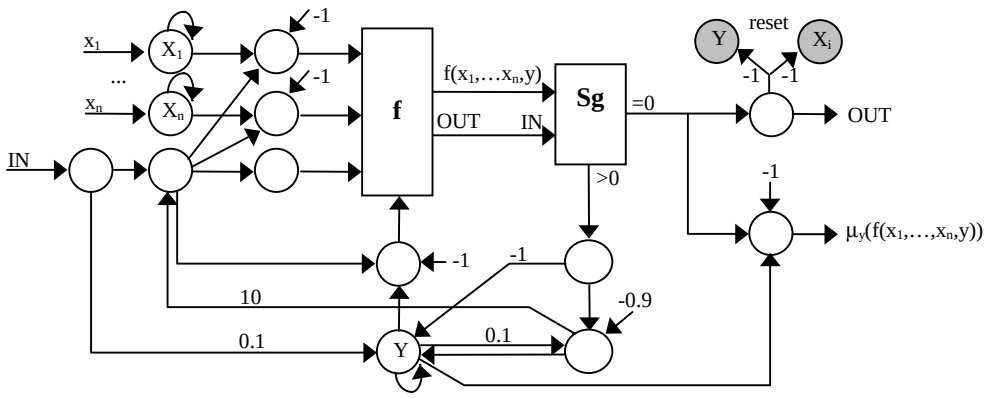
(iv) Signal (Sg)



(v) Composition

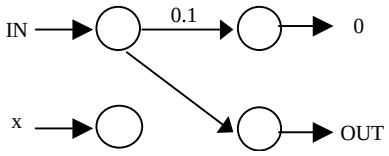


(vi) Recursion



(vii) Minimalisation

The unary zero net represents the function f(x)=0 and it is provided for optimisation purposes.



(viii) Unary zero (Z)

For each PRF expression an ARNN is generated using the structure of the expression. The construction of the ARNNs is in a top-down fashion, beginning with the outermost ARNN and then continuously instantiating ARNNs until we reach the axioms. In the expression `R(U(1,1),C(S,U(3,3))` we would first build the network for the recursion, then instantiate with the projection axiom network and with the composition rule network, that in turn would accommodate the successor and projection axiom networks.

This instantiation mechanism for the network schemas consists of replacing the boxes by compatible network schemas. A box is said to be compatible with a network schema if the number of inputs (respectively outputs) of the box is the same as the number of inputs (respectively outputs) of the network schema. The substitution operation of a box by a network schema consists of connecting the box inputs (respectively outputs) to the network inputs (respectively outputs).

4. Putting the pieces together and building the compiler

The tool we want to present to the user, should be capable of handling more complex functions than the simple examples used in the previous sections. Specifying a more complex function implies writing a more complex expression. This motivates the use of a modular and yet simple language to increase readability.

In this section we present the language we built to write PRFs, which is no more than some syntactic sugar added to the mathematical language of PRFs described in section 2, to allow for modular description of a target PRF. The programming *"methodology"* and semantic constraints are presented afterwards. We finish by giving some insight of how the compiler is itself implemented.

4.1 *Language grammar*

PRF descriptions are written using the following grammar:

1. *Funct_spec* → *stmt_seq*
2. *Stmt_seq* → *stmt* | *stmt_seq stmt*
3. *Stmt* → '**#define**' *id expr* | '**#target**' *id*
4. *Expr* → *rule* | *axiom* | *id*
5. *Rule* → '**C(**' *expr_list* '**)**' | '**R(**' *expr_list* '**)**' | '**M(**' *expr_list* '**)**'
6. *Axiom* → '**W**' | '**S**' | '**U(**' *num* '**,**' *num* '**)**'
7. *Expr_list* → *expr* | *expr_list* '**,**' *expr*
8. *Id* → *char* (*char* | *digit* | *symbol*)*
9. *Num* → *digit*+
10. *Digit* → '**0**' | '**1**' | '**2**' | '**3**' | '**4**' | '**5**' | '**6**' | '**7**' | '**8**' | '**9**'
11. *Char* → '**A**' | … | '**Z**' | '**a**' | … | '**z**'
12. *Symbol* → '**!**' | '**"**' | '**#**' | '**$**' | '**%**' | '**&**' | '**+**' | '**-**' | '**.**' | '**/**' | '**[**' | '****' | '**]**' | '**^**' | '**_**' | '**{**' | '**}**'

The grammar presented above is compatible with our initial requirements. To allow for PRFs to be specified easily we made the following modification on rule n.3 and added rule n.13:

3. *Stmt* → *expr_stmt* | '**#target**' *id*
13. *Expr_stmt* → '**#define**' *id expr* | *id expr* | *expr*

This means that we can omit `#define` and also *id* from rule n. 3. In the first case *id* will be used as a reference to *expr* and in the later case an *id* of the form `expr`*n* (where *n* is the expression number) is generated at compile-time.

By default the compiler assumes that the PRF being specified is the last one. We can change this, by using the `#target` directive.

4.2 *How to build a PRF description ?*

A PRF description is a sequence of statements where each statement is a pair that contains an identifier and an expression. The identifiers label the statement of an associated expression. Each expression is written in the language of PRF described in section 2 (recursive application of rules to rules and axioms) and may reference a previous statement. For example, when specifying the product PRF we can initially write the statement for the sum PRF and follow it by a shortened product PRF expression in which the sum sub-expression is substituted by the initial `sum` identifier as shown below.

```
sum R(U(1,1), C(U(3,3), S))
product R(Z, C(U(3,3), U(1,3), sum))
```

The idea is to write further expressions making use of the previously defined ones. A more detailed example is given in section 5.

4.3 *Programming constraints*

There are some programming constraints that must be respected when building PRF descriptions but those are not captured by the EBNF grammar described in section 4.1. Mainly, we cannot apply rules (C, R and M) to an arbitrary number of parameters (axioms, rules, identifiers) . Each expression results in a PRF with a well defined arity and in most rules the number of parameters accepted in a rule and their arities depend by turn on each others arities. A summary of these constrains is given in the tables below, where capital X denotes $x_1,\ldots,x_n$.

Axioms

	Arity Constraints	Description	# Param.	Arity
Zero – W	None	0	0	0

Successor – S	None	$f(x) = x+1$	0	1
Projection – U(i,n)	$i \leq n$	$f_I(x_1,\ldots,x_i,\ldots,x_n) = x_i$	2	N

Rules

	Arity Constraints	Description	# Param.	Arity
Composition – $C(f_1,\ldots,f_k,g)$	$f_1.Arity=\ldots=f_k.Arity$ and $g.Arity = k$	$h(X)=g(f_i(X))$	g.Arity+1	f_i.Arity
Recursion – R(f, g)	g.Arity=f.Arity+2	$h(X,0)=f(X)$ $h(X,y+1)=g(X,y,h(X,y))$	2	f.Arity+1
Minimalisation – M(f)	None	min y such that $f(X,y)=0$	1	f.Arity- 1

We also implemented the f(x)=0 PRF (unary zero-function) for convenience. We felt that it will help the user not having to write the R(W, U(2,2)) expression so often.

Built-in Expressions

	Arity constraints	Description	# Param.	Arity
Unary Zero – Z	None	f(x)=0	1	1

Following the trend of most traditional programming languages, references to statements that have not been defined are not allowed. This program is also case sensitive.

4.4 *How does the compiler work ?*

A PRF description written in the presented language is compiled sequentially. A symbol table is present to keep the identifiers used to label the expressions. At each step the compiler tries to build the parsing tree corresponding to the expression being analysed and checks for arity/number of parameters and naming inconsistencies as presented in section 4.3.

When using an identifier to refer to another expression, the compiler builds a link from the expression being parsed to the referenced expression. By means of this process the compiler is capable of constructing the complete tree of the target expression. The tree can then be browsed for some debugging purpose and used to derive the corresponding ARNN.

The ARNN is built from the parsing tree, creating a network schema for each rule and axiom node and then using the ARNN schema instantiating procedure, we build the final network. The reference links are treated as *pass-troughs*.

5. Using the simulator and compiling examples

An application was built to provide the user not only a compiler of PRF into ARNNs but also a simulator that among other things allows step-by-step simulation and inspection groups of neurons.

In our application there are two modes of building ARNNs: by compiling the PRFs or by reading the dynamic equations system that define the neural network. We do not only allow the storage of PRF description, but also of compiled PRF in order to avoid recompiling and to provide input to other simulation tools.

The simulator takes the built ARNN, configures it to read the input. Then it requests each neuron to process data. This process is iterated until a signal is seen in the OUT line of the outmost network.

During the ARNN computation, the program allows the user to inspect the computations that are taking place inside the net, being able to select some sets of neurons and customise the animation. The user can also see the corresponding equation of each neuron and define a limit for the number of iterations (if 0, no limit).

Each neuron is given a name by the compiler with special meaning. The name of each neuron expresses particular information on the neuron we are working with, in the following way:

X*FuncTypeNum_Depth_Id*

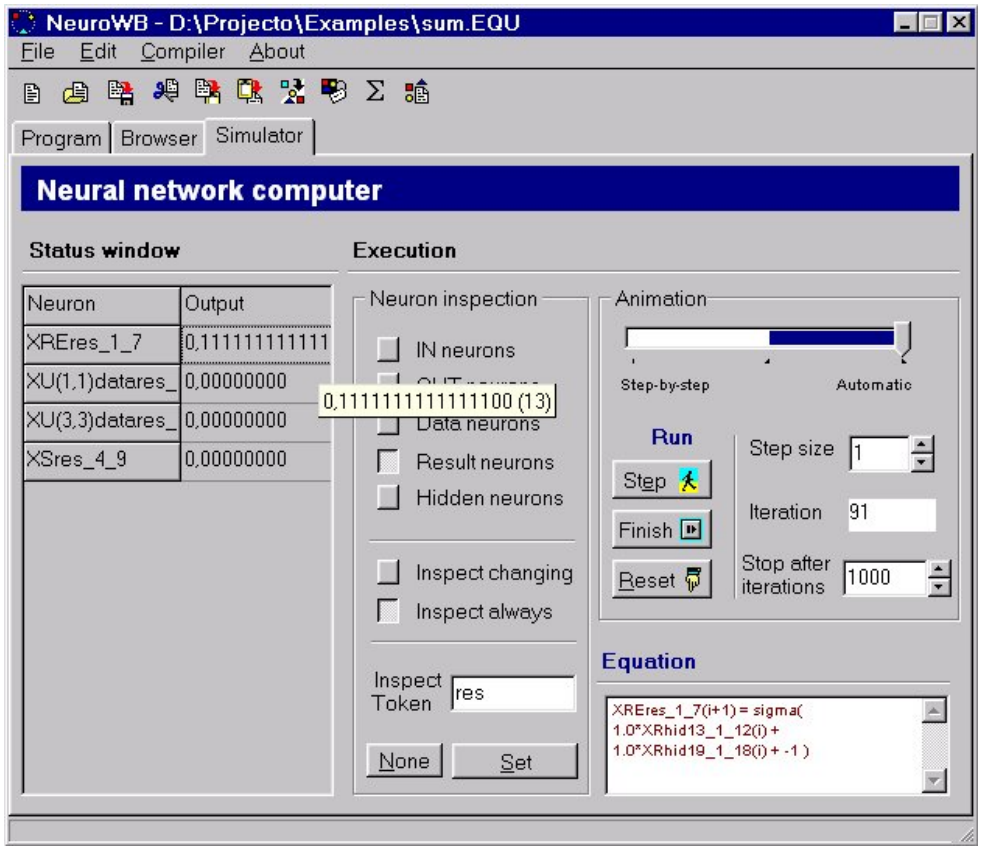
Where,

- *Func* is one of Z, W, S, U(i,j), R, C, M
- *Type* is one or more of the following:

In	Stands for the IN line supporting neurons.
Out	Stands for the OUT line supporting neurons.
Data	Stands for neurons that receive parameters from external networks or that send result of computation to other networks.
Res	Stands for neurons that handle results of functions.
Hid	Stands for neurons that are not in one of the border layers.

- *Num* is the number of the neuron inside its network schema.
- *Depth* is the same as the depth in the parsing tree of the PRF that the network corresponds to.
- *Id* is used to ensure an unique identifier for each neuron.

5.1 *A screenshot of the application (simulator)*



5.2 *Function description examples*

Some examples of function descriptions are given in the table bellow:

Function	Expression
f=0	`W`
f(x)=0	`Z`
f(x)=x+1	`S`
f(x,y,z)=y	`U(2,3)`
f(x,y,z)=z+1	`C(U(3,3),S)`
f(x,y)=x+y	`R(U(1,1),z+1)`
f(x,y)=x*y	`R(Z,C(U(3,3),U(1,3),x+y))`
f(x)=x!	`R(C(Z,S),C(U(2,2),C(U(1,2),S),x*y)`
f(x)=pot(x,y)	`R(C(Z,S),C(U(1,3),U(3,3),x*y)`
f(x)=sg(x)	`R(W,C(U(1,2),C(Z,S)))`
f(x)=$\underline{sg}$(x)	`R(C(W,S),C(W,U(1,2)))`
f(x,y)=x-1	`R(W,U(1,2))`
f(x,y)=x-y	`R(U(1,1),C(U(3,3),x-1))`
f(x,y)=|x-y|	`C(x-y,C(U(2,2),U(1,2),x-y),x+y)`
f(x,y)=min(x,y)	`C(U(1,2),x-y,x-y)`
f(x,y)=max(x,y)	`C(C(x-y,x+y),C(C(x-y,x+y),C(x-y,x-y),x-y),x+y)`
f(x,y)=rm(x,y)	`R(Z,C(C(U(3,3),S),C(C(U(1,3),C(U(3,3),S),|x-y|),sg),x*y))`
f(x,y)=x|y	`C(rm,sg)`
$f^{-1}(y)$	`M(C(C(C(U(2,2),fx),U(1,2),|x-y|),sg)`

As an example, let us consider the following description of the sum function:

```
proj/1 U(1,1)
proj/2 U(3,3)
comp C(proj/2,S)
sum R(proj/1,comp)
```

After the compilation of these four statements we have an ARNN with 39 neurons and 70 synapses with the following dynamic equation system:

```
XRin_1_0(i+1) = σ(Ein(i))
XRhid2_1_1(i+1) = σ(XRin_1_0(i))
XRhid3_1_2(i+1) = σ(0.1*XRin_1_0(i)+XRhid3_1_2(i)+XRhid4_1_3(i)-XRhid5_1_4(i)
       -XREout_1_15(i))
XRhid4_1_3(i+1) = σ(0.1*XRhid3_1_2(i)+XRhid5_1_4(i)-0.9)
XRhid5_1_4(i+1) = σ(XRhid6_1_5(i))
XRhid6_1_5(i+1) = σ(XSout_4_7(i))
XRhid7_1_6(i+1) = σ(XRhid3_1_2(i)+XRhid15_1_14(i)-1)
XREres_1_7(i+1) = σ(XRhid13_1_12(i)+XRhid19_1_18(i)-1)
XRhid9_1_8(i+1) = σ(XRhid5_1_4(i)+XU(1,1)inout_3_0(i))
XRhid10_1_9(i+1) = σ(0.2*XRhid9_1_8(i)+10*XRdatay_1_10(i)-1.2)
XRdatay_1_10(i+1) = σ(-XRhid9_1_8(i)+XRhid10_1_9(i)+XRdatay_1_10(i)-XREout_1_15(i)
       +Edata2(i))
XRhid12_1_11(i+1) = σ(XRhid9_1_8(i))
XRhid13_1_12(i+1) = σ(2*XRhid12_1_11(i)-100*XRhid14_1_13(i)-1)
XRhid14_1_13(i+1) = σ(XRdatay_1_10(i)-0.1)
XRhid15_1_14(i+1) = σ(2*XRhid12_1_11(i)+100*XRhid14_1_13(i)-2)
XREout_1_15(i+1) = σ(XRhid13_1_12(i))
XRhid17_1_16(i+1) = σ(XRhid15_1_14(i))
XRhid18_1_17(i+1) = σ(XRhid15_1_14(i)+XRhid19_1_18(i)-1)
XRhid19_1_18(i+1) = σ(-XREout_1_15(i)+XRhid19_1_18(i)-10*XRhid20_1_19(i)
       +XRhid21_1_20(i)+XU(1,1)datares_3_1(i))
XRhid20_1_19(i+1) = σ(XSres_4_9(i))
XRhid21_1_20(i+1) = σ(XRhid20_1_19(i))
XRdata1_1_21(i+1) = σ(XRdata1_1_21(i)-XREout_1_15(i)+Edata1(i))
XRhid22_1_22(i+1) = σ(XRdata1_1_21(i)+XRin_1_0(i)-1)
XRhid23_1_23(i+1) = σ(XRdata1_1_21(i)+XRhid15_1_14(i)-1)
XU(1,1)inout_3_0(i+1) = σ(XRhid2_1_1(i))
XU(1,1)datares_3_1(i+1) = σ(XRhid22_1_22(i))
```

```
XU(1,1)data_3_2(i+1) = σ(0)
XChid1_3_3(i+1) = σ(-XChid1_3_3(i)+XChid8_3_7(i))
XChid2_3_4(i+1) = σ(XChid1_3_3(i))
XChid6_3_5(i+1) = σ(-XChid7_3_6(i)+XChid6_3_5(i)+XU(3,3)datares_5_4(i))
XChid7_3_6(i+1) = σ(XChid6_3_5(i)+XChid1_3_3(i)-1)
XChid8_3_7(i+1) = σ(-XChid1_3_3(i)+XChid8_3_7(i)+XU(3,3)inout_5_3(i))
XU(3,3)inout_5_3(i+1) = σ(XRhid17_1_16(i))
XU(3,3)datares_5_4(i+1) = σ(XRhid18_1_17(i))
XU(3,3)data_5_5(i+1) = σ(XRhid23_1_23(i)+XRhid7_1_6(i))
XSin_4_6(i+1) = σ(XChid2_3_4(i))
XSout_4_7(i+1) = σ(XSin_4_6(i))
XSdata_4_8(i+1) = σ(XChid7_3_6(i))
XSres_4_9(i+1) = σ(XSin_4_6(i)+0.1*XSdata_4_8(i)-0.9)
```

6. Conclusion

We hope to have taken one step further in understanding the relationship between programming languages and ARNNs. We started presenting a theoretical framework of a very simple programming language. Next we described a systematic way of obtaining ARNNs from programs written in the presented language and concluded with the presentation of a tool where one can get a practical insight of the presented results. Although the ideas here contained may seem oversimplified w.r.t. real-world applications, adapting them to other more elaborate frameworks like those of process calculi appears to be straightforward. Finally, we also defend that the search for a language more suited to the architecture of ARNNs can give us fresh views on efficient approaches for systematic construction of ARNN.

References

[Boolos 80]
Boolos G. and Jeffrey, R., *Computability and Logic*, (2nd Ed), Cambridge University Press 1980.

[Cutland 80]
Cutland Nigel, *Computability – An introduction to recursive function theory*, Cambridge University Press.

[Gruau et al 94]
Gruau, Fréderic, Ratajszcza Jean-Yves, Wiber Giles, *Fundamental study – A Neural Compiler, Theoretical Computer Science*, Elsevier, 141 1-52 1995.

[Neto et al 97a]
J. Pedro Neto, Hava T. Siegelmann, and C.P. Suárez Araujo. *Turing Universality of Neural Nets (Revisited)*, Proceedings of Computer Aided Systems Theory – EUROCAST' 97, 1997.

[Neto et al 97b]
J. Pedro Neto, J. Félix Costa, Hava T. Siegelmann , *Building Neural Net Software*, To be published.

[Siegelmann 95]
Hava T. Siegelmann, *On The Computational Power of Neural Nets, Journal of Computer and system Sciences, Academic Press*, vol. 50. No.1, Feb. 1996.

[Siegelmann 96]
Hava T. Siegelmann, *ON NIL: The Software Constructor of Neural Nets*, P.P.L. vol. 6. No.4 1996 575-582.