ImNet: An Imperative Network Programming Language

Mohamed A. El-Zawawy[1,2,*]
[1]College of Computer and Information Sciences
Al Imam Mohammad Ibn Saud Islamic University
(IMSIU)
Riyadh
Kingdom of Saudi Arabia

[2]Department of Mathematics
Faculty of Science
Cairo University
Giza 12613
Egypt
maelzawawy@cu.edu.eg

Adel I. AlSalem
College of Computer and Information Sciences
Al Imam Mohammad Ibn Saud Islamic University
(IMSIU)
Riyadh
Kingdom of Saudi Arabia
alsalem@ccis.imamu.edu.sa

Abstract: One of the most recent architectures of networks is Software-Defined Networks (SDNs) using a controller appliance to control the set of switches on the network. The controlling process includes installing or uninstalling packet-processing rules on flow tables of switches.
This paper presents a high-level imperative network programming language, called *ImNet*, to facilitate writing efficient, yet simple, programs executed by controller to manage switches. *ImNet* is simply-structured, expressive, compositional, and imperative. This paper also introduces an operational semantics to *ImNet*. Detailed examples of programs (with their operational semantics) constructed in *ImNet* are illustrated in the paper as well.

Key–Words: Network programming languages, controller-switch architecture, operational semantics, syntax, *ImNet*.

1 Introduction

A network is a group of appliances connected to exchange data. Among these appliances are switches forwarding data depending on MAC addresses, routers forwarding data depending on IP addresses, and firewalls taking care of forbidden data. The network appliances are connected using a model that efficiently allows forwarding, storing, ignoring, tagging, and providing statistics about data moving in the network. Some of the network appliances, like routers [21, 16], are special in their functionality as they have some control over the network. This enables routers to compute and determine routes of data in the network. Of course different networks have different characteristics and abilities.

In 2011, the Open Networking Foundation [33], suggested removing the control owned by different network appliances and adding, instead, a general-purpose appliance, controller, to program different network appliances and querying data flowing in the network. The impact of this simple suggestion is huge; giant networks do not need special-purpose, complex, expensive switches any more. In such networks, cheap programmable switches can be used and programmed to configure and optimize networks via writing programs [20] running on controllers.

Software-Defined Networks (SDNs) [10] are networks established using the controller-switch architecture. A precise implementation of this architecture is OpenFlow [2] used to achieve various network-wide applications such as monitoring data flow, balancing switch load, network management, controlling appliances access, detection of service absence, host mobility, and forwarding data center. Therefore SDNs caused the appearance of network programming languages [18, 19, 17, 11].

This paper presents *ImNet*, an imperative high-level network programming language. *ImNet* expresses commands enabling controllers to program other network appliances including switches. *ImNet* has a clear and simply-structured syntax based on classical concepts of imperative programming that allows building rich and robust network application in a natural way. *ImNet* can be realized as a generalization of Frenetic [24] which is a functional network programming language. This is clear by the fact

*Corresponding author.

that the core of programs written in *ImNet* and Frenetic is based on a query result in the form of stream of values (packets, switches IDs, etc.). Commands for treating packets in *ImNet* include constructing and installing (adding to flow tables of switches) switch rules. *ImNet* supports building simple programs to express complex dynamic functionalities like load balancing and authentication. *ImNet* programs can also analyze packets and historical traffic patterns.

Motivation

The motivation of this paper is the lack of a simple syntax for an imperative network programming language. Yet, a stronger motivation is that most existing network programming languages are not supported theoretically (using operational semantics, type systems, program logics like FloydHoare logic,etc.).

Contributions

Contributions of this paper are the following.

1. A new simply-structured syntax for an imperative network programming language; *ImNet*.
2. An operational semantics (in the form of states and inference rules) for constructs of *ImNet*.
3. Two detailed examples of programs constructed in *ImNet* with their precise operation semantics.

Organization

The rest of this paper is organized as following. Section 2 presents the syntax and semantics of *ImNet*. The proposed semantics is operational and hence consists of states and inference rules presented in Section 2. Two detailed examples of programmes built in *ImNet* are presented in Section 3. This section also explains how the two examples can be assigned precise semantics using our proposed operational semantics. Section 4 reviews related work and gives directions for future work. Section 5 concludes the paper.

2 Semantics

This section presents the syntax and semantics of *ImNet*, a high-level programming language for SDN networks using switch-controller architecture. Figure 1 shows the syntax of *ImNet*. Figures 2 and 3 present the semantics of *ImNet* constructs. The proposed semantics is operational and its states are defined in the following definition.

Definition 1

1. $t \in$ *Types* $= \{$*int, Switch IDs, Packet, (Switch IDs, int, bool)*$\} \cup \{(t_1, t_2) \mid t_1, t_2 \in$ *Types*$\}$.
2. $v \in$ *Values* $=$ *Natural numbers* $\cup$ *Switch IDs* $\cup$ *Packets* $\cup$ *Switch IDs* $\times$ *Natural numbers* $\times$ *Boolean values* $\cup \{(v_1, v_2) \mid v_1, v_2 \in$ *Values*$\}$. *The expression* $v : t$ *denotes that the type of the value* v *is* t.
3. $ev \in$ *Events* $= \{(v_1, v_2, \ldots, v_n) \mid \exists t(\forall i\; v_i : t)\}$.
4. *Actions*$= \{$*sendcontroller, sendall, sendout, change(h,v)* $\}$.
5. $r \in$ *Rules* $=$ *Patterns* $\times$ *Actions*.
6. $rl \in$ *Rule-lists* $= \{[r_1, r_2, \ldots, r_n] \mid r_i \in$ *Rules*$\}$.
7. $ir \in$ *Intial-rule-assignment* $=$ *Switch IDs* $\times$ *Rules*.
8. $\sigma \in$ *Swich-states* $=$ *Flow-tables* $=$ *Switch IDs* $\rightarrow$ *Rule-lists*.
9. $\gamma \in$ *Variable-states* $=$ *Var* $\rightarrow$ *Events*$\cup$*Rule-lists*.
10. $s \in$ *States* $=$ *Swich-states* $\times$ *Variable-states* $\times$ *Rule-lists*.

A program in *ImNet* is a sequence of queries followed by a statement. The result of each query is an event which is a finite sequence of values. The event concept is also used in Frenetic. However an event in Frenetic is an infinite sequence of values. A value is an integer, a switch ID, a packet, a triple of a switch ID, an integer, and a Boolean value, or a pair of two values. Each value has a type of the set *Types*. In this paper, we focus on the details of statements as this is the most interesting part in a network programming language.

Possible actions taken by a certain switch on a certain packet are *sendcontroller*, *sendall*, *sendout*, or *change(h,v)*. The action *sendcontroller* sends a packet to the controller to take care of it. The action *sendall* sends the packet to all other switches. The action *sendout* sends the packet out of the switch through a certain port. The action *change(h,v)* modifies the header field h of the packet to the new value v.

A rule in our semantics is a pair of *pattern* and *action* where *pattern* is a form that concretely describes a set of packets and *action* is the action to be taken on elements of this set of packets. Rules are stored in tables (called *flow tables*) of switches. *Intial-rule-assignment* represents an initial assignment of rules to flow tables of switches.

ISBN: 978-960-474-363-6

$$
\begin{array}{rcl}
 & & x \in \text{lVar} \qquad\qquad Q \in \text{Queries} \\
et \in \text{Eventrans} & ::= & \text{Lift}(x, \lambda t.f(t)) \mid \text{ApplyLft}(x, \lambda t.f(t)) \mid \text{ApplyRit}(x, \lambda t.f(t)) \mid \\
 & & \text{Merge}(x_1, x_2) \mid \text{MixFst}(A, x_2, x_3) \mid \text{MixSnd}(A, x_2, x_3) \mid \\
 & & \text{Filter}(x, \lambda.f(t)) \mid \text{Once}(x) \mid \text{MakForwRule}(x) \mid \text{MakeRule}(x) \\
S \in \text{Stmts} & ::= & x := et \mid S_1; S_2 \mid \text{AddRules}(x) \mid \text{Register} \mid \text{Send}(x) \\
D \in \text{Defs} & ::= & \epsilon \mid x := Q \mid DD. \\
p \in \text{Progs} & ::= & D \gg S.
\end{array}
$$

Figure 1: ImNet Syntax.

A state in the proposed operational semantics is a triple (σ, γ, ir). In this triple γ captures the current state of the program variables and hence is a map from the set of variables to the set of events and rule lists. This is so because in *ImNet* variables may contain events or rule lists. The symbol σ captures the current state of flow tables of switches and hence is a map from switches *IDs* to rule lists. Finally, ir is an initial assignment of rules assigned to switches but have not been registered yet (have not been added to γ yet).

There are five type of statements in *ImNet*. The assignment statement $x := ef$ assigns the result of an event transformer (et) to the variable x. The statement *AddRules(x)* adds the switch rules stored in x to the reservoir of initially assigned rules. These are rules that are assigned to switches but are not added to flow tables yet. The statement *Register* makes the initial assignments permeant by adding them to flow tables of switches. The statement *Send(x)* sends specific packets to be treated in a certain way at certain switches. To keep a record of actions takes on packets on different switches we assume a map called *history* from the set of switches IDs to the set of lists of pairs of packets and taken actions. This map is used in the Rule (Send^s). Operational semantics of these statements are given in Figure 3. Judgement of inference rules in this figure have the form $S : (\sigma, \gamma, ir) \rightarrow (\sigma', \gamma', ir')$. This judgement reads as following. If the execution of S in the state (σ, γ, ir) ends then the execution reaches the state (σ', γ', ir').

Inference rules in Figure 3 use that in Figure 2 to get the semantics of the other important construct of *ImNet* which is event transformers (et). Judgements of Figure 2 have the form $et : \gamma \rightarrow u$ meaning that the semantics of the transformer et in the variable state γ is u. The event transformer $\text{Lift}(x, \lambda t.f(t))$ applies the map $\lambda t.f(t)$ to values of the event in x (Rule (Lift^s)). The event transformer $\text{Filter}(x, \lambda.f(t))$ filters the event in x using the map $\lambda t.f(t)$ (Rule (Filter^s)). From a given set of actions A and two events x_1 and x_2 the event transformers $\text{MixFst}(A, x_1, x_2)$ and $\text{MixSnd}(A, x_1, x_2)$ create lists of rules (Rules (Mix_1^s) and (Mix_2^s)).

3 Controller Programs

This section presents two examples of programs constructed using the syntax of *ImNet* (Figure 1). The first example constructs rules based on information stored in the variable x and then installs the established rules to flow tables of switches stored in z. This program has the following statements.

$$y = \text{MakeRule}(x);$$

$$z = \text{Lift}(z, \lambda t.(t, y));$$

$$\text{AddRules}(z);$$

$$Register;$$

The first statement of the program makes a rule for each value of the event stored in x. Then the second statement assigns these rules to switch IDs in the event stored in z. The third statement stores the rule assignment of z in ir as an initial rule assignment. The last statement of the program adds the established rules to the flow tables of switches. Figure 4 shows the operational semantics of this program using the semantics of the previous section.

The second example constructs forwarding rules based on source IPs of arriving packets and then installs the established rules to flow tables of switch IDs stored in z. This program has the following statements.

$$y = \text{SourceIps};$$

$$y = \text{ApplyLft}(y, \lambda t.(t, \text{port}(t)));$$

$$y = \text{Lift}(y, \lambda t.(t, \text{switch}(t, z));$$

$$y = \text{MakForwRule}(y);$$

$$\text{AddRules}(y);$$

$$Register;$$

ISBN: 978-960-474-363-6

$$\frac{v_i : t \qquad \gamma(x) = (v_1, v_2, \ldots, v_n)}{\text{Lift}(x, \lambda t.f(t)) : \gamma \to (f(v_1), f(v_2), \ldots, f(v_n))} \ (\text{Lift}^s)$$

$$\frac{\gamma(x_1) = (v_1, v_2, \ldots, v_n) \qquad \gamma(x_2) = (w_1, w_2, \ldots, w_n)}{\text{Merge}(x_1, x_2) : \gamma \to ((v_1, w_1), (v_2, w_2), \ldots, (v_n, w_n))} \ (\text{Merge}^s)$$

$$\frac{\gamma(x) = (v_1, v_2, \ldots, v_n) \qquad A = \{i \mid f(v_i) = \text{true}\}}{\text{Filter}(x, \lambda.f(t)) : \gamma \to (\ldots, v_i, \ldots \mid i \in A)} \ (\text{Filter}^s)$$

$$\frac{v_i : t \qquad \gamma(x) = ((v_1, v_1'), (v_2, v_2'), \ldots, (v_n, v_n'))}{\text{ApplyLft}(x, \lambda t.f(t)) : \gamma \to ((f(v_1), v_1'), (f(v_2), v_2'), \ldots, (f(v_n), v_n'))} \ (\text{App}_1^s)$$

$$\frac{v_i' : t \qquad \gamma(x) = ((v_1, v_1'), (v_2, v_2'), \ldots, (v_n, v_n'))}{\text{ApplyRit}(x, \lambda t.f(t)) : \gamma \to ((v_1, f(v_1')), (v_2, f(v_2')), \ldots, (v_n, f(v_n')))} \ (\text{App}_2^s)$$

$$\frac{\text{type}(x) \in \text{Types}}{\text{Once}(x) : \gamma \to (\underbrace{x, x, \ldots, x}_{n\_times})} \ (\text{Once}^s)$$

$$\frac{\gamma(x_1) = (v_1^1, v_2^1, \ldots, v_n^1) \quad \gamma(x_2) = (v_1^2, v_2^2, \ldots, v_n^2) \quad A_1 = A \cup \{v_1^1\} \quad \forall i > 1. A_i = A_{i-1} \cup \{v_i^1\}}{\text{MixFst}(A, x_1, x_2) : \gamma \to ((A_1, v_1^2), (A_2, v_2^2), \ldots, (A_n, v_n^2))} \ (\text{Mix}_1^s)$$

$$\frac{\gamma(x_1) = (v_1^1, v_2^1, \ldots, v_n^1) \quad \gamma(x_2) = (v_1^2, v_2^2, \ldots, v_n^2) \quad A_1 = A \cup \{v_1^2\} \quad \forall i > 1. A_i = A_{i-1} \cup \{v_i^2\}}{\text{MixSnd}(A, x_1, x_2) : \gamma \to ((v_1^1, A_1), (v_2^1, A_2), \ldots, (v_n^1, A_n))} \ (\text{Mix}_2^s)$$

$$\frac{\gamma(x) = ((v_1^1, a_1, v_1^3), (v_2^1, a_2, v_2^2), \ldots, (v_n^1, a_n, v_n^2))}{\text{MakForwRule}(x) : \gamma \to [(v_1^1, (v_1^3, sendout(v_1^2))), (v_2^1, (v_2^3, sendout(v_2^2))), \ldots, (v_n^1, (v_n^3, sendout(v_n^2)))]} \ (\text{MFR}^s)$$

$$\frac{\gamma(x) = ((v_1^1, a_1, v_1^2), (v_2^1, a_2, v_2^2), \ldots, (v_n^1, a_n, v_n^2))}{\text{MakeRule}(x) : \gamma \to [(v_1^1, a_1(v_2^1)), (v_1^2, a_2(v_2^2)), \ldots, (v_n^i, a_n(v_2^n))]} \ (\text{MkRl}^s)$$

Figure 2: Operational semantics for event functions of ImNet

The first statement of the program assumes a function *SourceIps* that returns source IPs of arriving packets and stores them in the form of an event in y. The second statement transfers event of y into event of pairs of IPs and port numbers through which packets will be forwarded. The third statement augments values of event in y with switch IDs from the event stored in z. The fourth statement makes a forward rule for each value of the event stored in y. Then the fifth statement stores the rule assignment of y in ir as an initial rule assignment. The last statement of the program adds the established rules to the flow tables of switches. Figure 5 shows the operational semantics of this program using the semantics of the previous section.

4 Related and Future Work

This section presents work most related to that presented in the current paper.

One of the early attempts to develop software-defined networking (SDN) is NOX [9] based on ideas from [8] and 4D [7]. On the switch-level, NOX uses explicit and callbacks rules for packet-processing. Examples of applications that benefitted from NOX are load balancer [6] and the work in [4, 5]. Many directions for improving platforms of programming networks include Maestro [2] and Onix [3], which uses distribution and parallelization to provide better performance and scalability.

A famous programming language for networks is Frenetic [24, 25] which has two main components. The first component is a collection of operators that are source-level. The operators aim at establishing and treating streams of network traffic. These operators also are built on concepts of functional programming (FP) and query languages of declarative database. Moreover the operators support a modular design, a cost control, a race-free semantics, a single-tier programming, and a declarative design. The second component of Frenetic is a run-time system. This system facilitates all of the actions of adding and re-

ISBN: 978-960-474-363-6

$$\frac{et : \gamma \to u}{x := et : (\sigma, \gamma, ir) \to (\sigma, \gamma[x \mapsto u], ir)} \text{(Assgn}^s\text{)}$$

$$\frac{S_1 : (\sigma, \gamma, ir) \to (\sigma'', \gamma'', ir'') \qquad S_2 : (\sigma'', \gamma'', ir'') \to (\sigma', \gamma', ir')}{S_1; S_2 : (\sigma, \gamma, ir) \to (\sigma', \gamma', ir')} \text{(seq}^s\text{)}$$

$$\frac{\gamma(x) \in \text{Intial-rule-assignment}}{\text{AddRules}(x) : (\sigma, \gamma, ir) \to (\sigma, \gamma, ir \cup \gamma(x))} \text{(Addrl}^s\text{)}$$

$$\frac{}{\text{Register} : (\sigma, \gamma, ir) \to (\sigma \cup ir, \gamma, \emptyset)} \text{(Reg}^s\text{)}$$

$$\frac{\gamma(x) = ((v_1^1, v_1^2, v_1^3), (v_2^1, v_2^2, v_2^3), \ldots, (v_n^1, v_n^2, v_n^3)) \qquad \forall i.(v_2^i, v_3^i) \in \text{history}(v_1^i)}{\text{Send}(x) : (\sigma, \gamma, ir) \to (\sigma, \gamma, ir)} \text{(Send}^s\text{)}$$

Figure 3: Operational semantics statements of ImNet

moving low-level rules to and from flow tables of switches. One advantage of *ImNet*, the language presented in this paper, over Frenetic is that *ImNet* is imperative. Therefore *ImNet* paves the way to the appearance of other types of network programming languages such as object-oriented network programming langues and context-oriented network programming languages.

Other examples to program network components though high-level languages are NDLog and NetCore [1]. NetCore provides an integrated view of the whole network. NDLog is designed in an explicitly distributed fashion.

As an extension of Datalog, NDLog [22, 23] was presented to determine and code protocols of routing [21], overlay networks, and concepts like hash tables of distributed systems. *ImNet* (presented in this paper), Frenetic, and NDLog can be classified as high-level network programming languages. While NDLog main focus is overlay networks and routing protocols, Frenetic (in a functional way) and *ImNet* (in an imperative way) focus on implementing packet processing such as modifying header fields. Therefore *ImNet* equips a network programmer with a modular view of the network which is not provided by NDLog and Frenetic. This is supported by the fact that a program in NDLog is a single query that is calculated on each router of the network.

The switch component [30] of networks can be programmed via many interfaces such as OpenFlow platform. Examples of other platforms include Shangri-La [31] and FPL-3E [32], RouteBricks [29], Click modular router [26], Snortran [27] and Bro [28]. The idea in Shangri-La [31] and FPL-3E [32] is to produce certain hardware for packet-processing from high-level programs that achieves packet-processing. In RouteBricks [29], stock machines are used to improve performance of program switches. As a modular approach, the platform of Click modular router [26], enables programming network components. This system focuses on software switches in the form of Linux kernel code. For the sake of intrusions detection and preserving network security, Snortran [27] and Bro [28] enable coding monitoring strategies and robust packet-filtering. One advantage of *ImNet*, the language presented in this paper, over all the related work is that *ImNet* overcomes the disadvantage of most similar languages of focusing on controlling a single device.

There are many interning directions for future work. One such direction is develop methods for static analysis of network programming languages. Obviously associating these analyses with correctness proofs, in the spirit of [12, 13, 15, 14], will have many network applications.

5 Conclusion

Software-Defined Networks (SDNs) is a recent architectures of networks in which a controller device programs other network devices (specially switches) via a sequence of installing and uninstalling rules to memories of these devices.

In this paper, we presented a high-level imperative network programming language, called *ImNet*, to facilitate the job of controller through efficient, yet simple, programs. *ImNet* has the advantages of simplicity, expressivity, propositionally, and being imperative. The paper also introduced a concrete operational semantics to meanings of *ImNet* constructs. Detailed examples of using *ImNet* and the operational semantics were also illustrated in the paper.

$(\emptyset, \{z \mapsto \{id_1, id_2\}, x \mapsto \{((srcport(80), sendall, \_), (inport(1), sendcontroller, \_))\}, [])$
$y =$ **MakeRule**(x);
$(\emptyset, \{z \mapsto \{id_1, id_2\}, x \mapsto \{((srcport(80), sendall, \_), (inport(1), sendcontroller, \_))\},$
$y \mapsto \{(srcport(80), [sendall]), (inport(1), [sendcontroller])\}\}, \emptyset)$
$z =$ **Lift**$(z, \lambda t.(t, y))$;
$(\emptyset, \{z \mapsto \{(id_1, \gamma(y)), (id_2, \gamma(y))\},$
$x \mapsto \{((srcport(80), sendall, \_), (inport(1), sendcontroller, \_))\},$
$y \mapsto \{(srcport(80), [sendall]), (inport(1), [sendcontroller])\}\}, \emptyset)$
AddRules(z);
$(\emptyset, \{z \mapsto \{(id_1, \gamma(y)), (id_2, \gamma(y))\},$
$x \mapsto \{((srcport(80), sendall, \_), (inport(1), sendcontroller, \_))\},$
$y \mapsto \{(srcport(80), [sendall]), (inport(1), [sendcontroller])\}\}, \{(id_1, \gamma(y)), (id_2, \gamma(y))\})$
Register;
$(\{(id_1, \gamma(y)), (id_2, \gamma(y))\}, \{z \mapsto \{(id_1, \gamma(y)), (id_2, \gamma(y))\},$
$x \mapsto \{((srcport(80), sendall, \_), (inport(1), sendcontroller, \_))\},$
$y \mapsto \{(srcport(80), [sendall]), (inport(1), [sendcontroller])\}\}, \emptyset)$

Figure 4: Example 1; an operational semantics of a program written in ImNet

$(\emptyset, \{z \mapsto \{id_1, id_2\}\}, [])$
$y =$ **SourceIps**;
$(\emptyset, \{z \mapsto \{id_1, id_2\},$
$y \mapsto \{(ip_1, pk_1), (ip_2, pk_2)\}\}, \emptyset)$
$y =$ **ApplyLft**$(y, \lambda t.(t,$ **port**$(t)))$;
$(\emptyset, \{z \mapsto \{id_1, id_2\},$
$y \mapsto \{(pr_1, pk_1), (pr_2, pk_2)\}\}, \emptyset)$
$y =$ **Lift**$(y, \lambda t.(t,$ **switch**$(t, z))$;
$(\emptyset, \{z \mapsto \{id_1, id_2\},$
$y \mapsto \{(id_1, pr_1, pk_1), (id_2, pr_2, pk_2)\}\}, \emptyset)$
$y =$ **MakForwRule**(y);
$(\emptyset, \{z \mapsto \{id_1, id_2\},$
$y \mapsto \{(id_1, (pk_1, sendout(pr_1)), (id_2, (pk_2, sendout(pr_2)))\}\}, \emptyset)$
AddRules(y);
$(\emptyset, \{z \mapsto \{id_1, id_2\},$
$y \mapsto \{(id_1, (pk_1, sendout(pr_1)), (id_2, (pk_2, sendout(pr_2)))\}\},$
$\{(id_1, (pk_1, sendout(pr_1)), (id_2, (pk_2, sendout(pr_2)))\})$
Register;
$(\{(id_1, (pk_1, sendout(pr_1)), (id_2, (pk_2, sendout(pr_2)))\}, \{z \mapsto \{id_1, id_2\},$
$y \mapsto \{(id_1, (pk_1, sendout(pr_1)), (id_2, (pk_2, sendout(pr_2)))\}\}, \emptyset)$

Figure 5: Example 2; an operational semantics of a program written in ImNet

ISBN: 978-960-474-363-6

References:

[1] B. Loo, J. Hellerstein, I. Stoica, and R. Ramakrishnan. Declarative routing: Extensible routing with declarative queries. *SIGCOMM*, 2005, pp.289-300.

[2] Z. Cai, A. Cox, and T. Ng. Maestro. A system for scalable OpenFlow control. *Technical Report TR10-08, Rice University*, Dec 2010.

[3] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama, and S. Shenker. Onix: A distributed control platform for large-scale production networks. *OSDI*, Oct 2010.

[4] N. Handigol, S. Seetharaman, M. Flajslik, N. McKeown, and R. Johari. Plug-n-Serve. Load-balancing web traffic using OpenFlow. *Demo at ACM SIGCOMM*, Aug 2009.

[5] B. Heller, S. Seetharaman, P. Mahadevan, Y. Yiakoumis, P. Sharma, S. Banerjee, and N. McKeown. ElasticTree: Saving energy in data center networks. *NSDI*, Apr 2010.

[6] R. Wang, D. Butnariu, and J. Rexford. OpenFlow-based server load balancing gone wild. *Hot-ICE*, Mar 2011.

[7] A. Greenberg, G. Hjalmtysson, D. Maltz, A. Myers, J. Rexford, G. Xie, H. Yan, J. Zhan, and H. Zhang. A clean slate 4D approach to network control and management. *SIGCOMM CCR* 35, October 2005, pp.41-54.

[8] M. Casado, M. Freedman, J. Pettit, J. Luo, N. Gude, N. McKeown, and S. Shenker. Rethinking enterprise network control. *Trans. on Networking.* 17(4), Aug 2009.

[9] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker. NOX: Towards an operating system for networks. *SIGCOMM CCR* 38(3), 2008.

[10] N. Foster, A. Guha, M. Reitblatt, A. Story, M. Freedman, N. Katta, C. Monsanto, J. Reich, J. Rexford, C. Schlesinger, D. Walker, R. Harrison. Languages for software-defined networks. *IEEE Communications Magazine* 51(2), 2013, pp. 128–134.

[11] T. Bain, P. Campbell, J. Karlsson. Modeling growth and dynamics of neural networks via message passing in Erlang: neural models have a natural home in message passing functional programming languages. *Erlang Workshop*, 2011, pp. 94-97.

[12] M. El-Zawawy. Flow sensitive-insensitive pointer analysis based memory safety for multithreaded programs. In: Murgante, B., Gervasi, O., Iglesias, A., Taniar, D., Apduhan, B.O. (eds.) *ICCSA*, Part V. LNCS, vol. 6786, Springer, Heidelberg (2011), pp. 355-369.

[13] M. El-Zawawy. Probabilistic pointer analysis for multithreaded programs. *ScienceAsia* 37(4), 2011, pp. 344-354.

[14] M. El-Zawawy. Detection of Probabilistic Dangling References in Multi-core Programs Using Proof-Supported Tools. *ICCSA* 2013, pp. 516–530.

[15] M. El-Zawawy. Frequent Statement and Dereference Elimination for Distributed Programs. *ICCSA*, 2013, pp. 82–97.

[16] T. Suzuki, K. Pinte, T. Cutsem, W. De Meuter, A. Yonezawa. Programming language support for routing in pervasive networks. *PerCom Workshops*, 2011, pp. 226–232.

[17] A. Elsts, L. Selavo. A user-centric approach to wireless sensor network programming languages. *SESENA* 2012, pp. 29–30.

[18] C. Monsanto, N. Foster, R. Harrison, D. Walker. A compiler and run-time system for network programming languages. *POPL* 2012, pp. 217–230.

[19] S. Hong, Y. Joung. Meso: an object-oriented programming language for building strongly-typed internet-based network applications. *SAC* 2013, pp.1579–1586.

[20] J. Rexford. Programming languages for programmable networks. *POPL* 2012, pp. 215–216.

[21] H. Arneson, C. Langbort. A linear programming approach to routing control in networks of constrained linear positive systems. *Automatica* 48(5), 2012, pp. 800-807.

[22] B. Loo, T. Condie, J. Hellerstein, P. Maniatis, T. Roscoe, and I. Stoica. Implementing declarative overlays. *SIGOPS* 39(5), 2005, pp 75-90.

[23] B. Loo, J. Hellerstein, I. Stoica, and R. Ramakrishnan. Declarative routing: Extensible routing with declarative queries. *SIGCOMM*, 2005, pp. 289-300.

[24] N. Foster, R. Harrison, M. Meola, M. Freedman, J. Rexford, and D. Walker. Frenetic: A high-level langauge for OpenFlow networks. *PRESTO*, Nov 2010.

[25] N. Foster,R. Harrison, M. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker. Frenetic: A Network Programming Language.*the 16th ACM SIGPLAN international conference on Functional programming*,2011 pp. 279–291.

[26] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. Kaashoek. The Click modular router. *ACM Transactions on Computer Systems* 18(3), Aug 2000, pp 263-297.

[27] S. Egorov and G. Savchuk. SNORTRAN: An Optimizing Compiler for Snort Rules. *Fidelis Security Systems*, 2002.

[28] V. Paxson. Bro: A system for detecting network intruders in realtime. *Computer Networks* 31(2324), Dec 1999, pp. 2435-2463.

[29] M. Dobrescu, N. Egi, K. Argyraki, B. Chun, K. Fall, G. Iannaccone, A. Knies, M. Manesh, and S. Ratnasamy. RouteBricks: Exploiting parallelism to scale software routers. *SOSP*, Oct 2009.

[30] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. Openflow: Enabling innovation in campus networks. *SIGCOMM CCR* 38(2), 2008, pp. 69-74.

[31] M. Chen, X. Li, R. Lian, J. Lin, L. Liu, T. Liu, and R. Ju. Shangri-la: Achieving high performance from compiled network applications while enabling ease of programming. *PLDI*, Jun 2005, pp 224-236.

[32] M. Cristea, C. Zissulescu, E. Deprettere, and H. Bos. FPL-3E: Towards language support for reconfigurable packet processing. *SAMOS*, Jul 2005, pp 201-212.

[33] The Open Networking Foundation, Mar 2011. See http:// www.opennetworkingfoundation.org/