ImNet: An Imperative Network Programming Language

Mohamed A. El-Zawawy1,2,*
1College of Computer and Information Sciences
Al Imam Mohammad Ibn Saud Islamic University (IMSIU)
Riyadh
Kingdom of Saudi Arabia
2Department 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 Freneric 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. \( t \in \text{Types} = \{\text{int}, \text{Switch IDs, Packet}, \{\text{Switch IDs}, \text{int}, \text{bool}\}\} \cup \{(t_1, t_2) \mid t_1, t_2 \in \text{Types}\} \).

2. \( v \in \text{Values} = \text{Natural numbers} \cup \text{Switch IDs} \cup \text{Packets} \cup \text{Switch IDs} \times \text{Natural numbers} \times \text{Boolean values} \cup \{(v_1, v_2) \mid v_1, v_2 \in \text{Values}\} \).
The expression \( v : t \) denotes that the type of the value \( v \) is \( t \).

3. \( ev \in \text{Events} = \{(v_1, v_2, \ldots, v_n) \mid \exists \forall v_i : t\}\).

4. \( \text{Actions} = \{\text{sendcontroller, sendall, sendout, change}(h, v)\} \).

5. \( r \in \text{Rules} = \text{Patterns} \times \text{Actions} \).

6. \( rl \in \text{Rule-lists} = \{[r_1, r_2, \ldots, r_n] \mid r_i \in \text{Rules}\} \).

7. \( ir \in \text{Initial-rule-assignment} = \text{Switch IDs} \times \text{Rules} \).

8. \( \sigma \in \text{Switch-states} = \text{Flow-tables} = \text{Switch IDs} \rightarrow \text{Rule-lists} \).

9. \( \gamma \in \text{Variable-states} = \text{Var} \rightarrow \text{Events} \cup \text{Rule-lists} \).

10. \( s \in \text{States} = \text{Switch-states} \times \text{Variable-states} \times \text{Rule-lists} \).

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 Freneric. However an event in Freneric is an infinite sequence of values. Each value has a type of the set \( \text{Types} \). 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. Initial-rule-assignment represents an initial assignment of rules to flow tables of switches.
A state in the proposed operational semantics is a triple \((\sigma, \gamma, ir)\). In this triple \(\gamma\) 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 \(\sigma\) 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 \(\gamma\) 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 permutable 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 (\(Send^*\)). 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 \((\sigma, \gamma, ir)\) ends then the execution reaches the state \((\sigma', \gamma', ir')\).

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 \(\gamma\) is \(u\). The event transformer \(Lift(x, \lambda.f(t))\) applies the map \(\lambda.f(t)\) to values of the event in \(x\) (Rule (\(Lift^*\))). The event transformer \(Filter(x, \lambda.f(t))\) filters the event in \(x\) using the map \(\lambda.f(t)\) (Rule (\(Filter^*\))). From a given set of actions \(A\) and two events \(x_1\) and \(x_2\) the event transformers \(MixFst(A, x_1, x_2)\) and \(MixSnd(A, x_1, x_2)\) create lists of rules (\(Rules (Mix_1^*)\) and \(Mix_2^*\)).

### 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.

\[
y = \text{MakeRule}(x);
\]
\[
z = \text{Lift}(z, \lambda.t.(t, y));
\]
\[
\text{AddRules}(z);
\]
\[
\text{Register};
\]

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.

\[
y = \text{SourceIps};
\]
\[
y = \text{ApplyLft}(y, \lambda.t.(t, \text{port}(t)));
\]
\[
y = \text{Lift}(y, \lambda.t.(t, \text{switch}(t, z)));
\]
\[
y = \text{MakForwRule}(y);
\]
\[
\text{AddRules}(y);
\]
\[
\text{Register};
\]
The first statement of the program assumes a function `SourceCps` 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 \( z \). 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-
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 languages 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 interesting 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.
Figure 4: Example 1; an operational semantics of a program written in ImNet

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


[22] B. Loo, J. Rexford, C. Schlesinger, D. Walker, R. Harri-


