# A Pipeline Functional Language for Stateful Packet Processing

Nicola Bonelli, Stefano Giordano and Gregorio Procissi
Dipartimento di Ingegneria dell'Informazione
Università di Pisa and CNIT
Via G, Caruso 16, 56122 Pisa, Italy
Email: {nicola.bonelli@for., stefano.giordano@, gregorio.procissi@}unipi.it

***Abstract*—The evolution of commodity PCs towards multi–core processing platforms equipped with high–speed network interfaces makes them reasonable and cost effective targets for the implementation of generic network functions. In addition, the availability of software accelerated I/O frameworks provides a convenient ground for running a broad variety of applications, from simple software switches to more complex network systems, with near hardware–class performance and the flexibility of a software approach.**

**Most network functions can be implemented by composing a set of elementary operations into processing pipelines to be run on top of multiple processing cores. In this framework, maintaining the flow consistency is crucial to enable stateful operations in the processing pipelines.**

**This paper presents Enif–Lang, a functional language for programming network pipelines specifically targeted at multi–core scenarios. In addition to a large set of functions for generic packet manipulation, filtering, steering and state management, the framework is built upon an abstract model that provides state aware packet splitting to prevent inter–state sharing and enable consistent stateful parallel processing on–top–of multi–core architectures.**

## I. Introduction

The rise of Software Defined Networks (SDN) and Network Function Virtualization (NFV) as the winning trends for the design and development of network functions and services has naturally switched the focus of the research community towards network virtualization and softwarization. Both paradigms propose a dramatic change in the way network operations and services are conceived and push *programmability* and *reconfigurability* as crucial keywords of a new generation of network devices. At the same time, the evolution of commodity hardware towards powerful multi–core platforms together with the wide availability of a new family of multi–gigabit network interfaces point to off–the–shelf PCs as viable alternatives to dedicated hardware devices for the development of network functions. In addition, a flourish literature about effective I/O operations has lately proposed accelerated engines such as PF_RING [1], PF_RING ZC (Zero Copy) [2], Netmap [3], DPDK [4] and PFQ [5] that allow to handle packets up to 10+ Gbps line rate.

In this scenario, the adoption of commodity PCs as traditional network nodes – such as switches and routers – as well as more complex network middleboxes implementing specific services becomes an almost natural consequence. In parallel, a broad variety of programming models and abstractions have been recently proposed for the data plane of generic network nodes.

Indeed, starting from the basic (stateless) Match–Action paradigm proposed by OpenFlow [6], several other approaches (e.g., [7], [8]) have been proposed to enable stateful processing within network devices. However, stateful operations do not straightforwardly cope with parallel processing when performance matters. The common issue is represented by the possible state sharing among flows processed by different cores. Such a condition becomes even more critical when multiple applications run concurrently on the same set of network interfaces.

This paper presents Enif–Lang (Enhanced Network processIng Functional Language), an easy, expressive and robust programming language specifically tailored to network traffic processing for multi–core PCs running Linux OS. Originally born as PFQ–Lang [9], [5], the language has evolved quite a lot lately and the current version implements a reference model that *enables and automates state persistent concurrent programming*. Like any functional language, Enif–Lang supports high-order functions (functions that take or return other functions as arguments) and currying, that turns functions that takes multiple arguments into functions that take a single argument. Moreover, the language includes conditional functions and predicates to implement a basic code control flow.

Since Enif–Lang is used to describe and specify packet processing pipelines, it plays a similar role to that of the lower level P4 [10] language and to the imperative language Pyretic [11] (that belongs to the Frenetic [12] family of network programming languages) in describing the data plane logic of an SDN network or to that of Streamline [13] to configure I/O paths to applications through the operating system. In addition, VPP [14] and eBPF [15] recently proposed alternative approaches for packet processing and data plane programming.

The rest of the paper is organized as follows. Section II presents the working scenario of Enif–Lang and sketches the main features of the language. Section III presents the language description of pipelines and elaborates upon stateful processing. Finally, Section IV presents two practical use cases

978-1-5090-6008-5/17/$31.00 

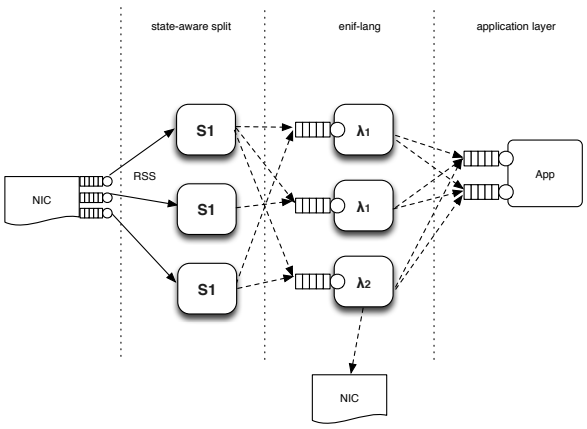


Fig. 1. The Enif–Lang abstract processing model

of usage for Enif–Lang while Section V draws conclusions and final remarks.

## II. Enif–Lang at a glance

Enif–Lang is a functional language entirely implemented as a declarative Domain Specific Language (DSL) on top of the Haskell Language and it is designed to ease the implementation of network applications by leveraging a strong type–safe system and the functional composition typical of functional programming languages.

Figure 1 shows the full Enif–Lang abstract processing model. Packets retrieved at physical network interfaces traverse a splitting layer that provides per–flow consistency within the number of processing resources and are injected into the processing engines (the $\lambda_i$ blocks in the picture). At this stage, the pipelines of computation are executed according to the formal description provided by Enif–Lang and packets are finally forwarded to the selected *endpoints*, i.e., to network cards, application threads, the OS kernel, etc..

In Enif–Lang, pipelines are formally expressed as the composition of effectful functions, called *actions*, that perform network operations on top of packets. In analogy with its predecessor PFQ–Lang [9], Enif–Lang actions are formally modelled around the concept of *monad*, a structure borrowed from the Category Theory and widely used in other languages, such as Haskell, Scala etc. In short, monads provide the theoretical support for composing effectful functions (such as those performing I/O, those that handle a state associated with a packet or a flow, and so on) while maintaining the language functionally pure. Enif–Lang actions include the most common packet processing primitives for packet forwarding, filtering, steering, logging and statistics retrieval.

At the time of writing, an open–source experimental implementation of the reference architecture exists within the PFQ framework [5] for the Linux operating system. In particular, an experimental compiler is used to produce intermediate representations of Enif–Lang programs that can be injected end executed at the kernel level. In addition, the PFQ kernel module offers a wide set of primitives implemented in the C language which are functionally composed at runtime according to the Enif–Lang source code.

The rest of the section describes the basic grammar and syntax of the language along with simple snippets for its practical usage.

### A. Functions overview

Enif–Lang is equipped with a set of built–in primitives as well as a more complete library of functions to describe generic processing pipelines. In the following, such functions are roughly divided into different categories according to their purposes.

**Predicates.** Predicates are pure functions that take an arbitrary number of arguments (possibly none) plus the current packet and return a Boolean value. Such functions are either used within the `if-then-else` statement, or passed as argument to high–order–functions to specialize their behavior.

The language implements a set of primitive predicates (that cannot be directly implemented in Enif–Lang) that compose together to implement more complex ones.

The default library includes predicates for the most common protocols. As a convention, the names of such functions are prefixed by `is_` or `has_`, when meaningful.

Examples are `is_tcp`, `is_udp`, `is_icmp`, `is_rtp`, `is_sip`, `is_gtp` or `has_port 80`, `has_addr "192.168.0.0/24"`.

**Combinators.** Enif–Lang provides a set of combinators, that is functions designed to combine predicates together. In particular, the composition of predicates is enabled by the logical `or`, `and`, `xor` and `not` functions.

**Properties.** Properties are functions designed to return a value associated with a packet. Typical examples are hash functions computed over a portion of a packet, header field extractors, state retrievals, etc. The Enif–Lang library is equipped with a wide gamma of property functions for the most common protocols (ip, tcp, udp, icmp), as well as with generic functions that extract the field value of arbitrary protocols (by specifying the offset and the size of the field). For example, properties of the IP header are: `ip_ttl`, `ip_tot_len`, `ip_id`, `ip_frag`, `ip_ttl`.

**Comparators.** Properties are meaningful only when used with comparators, namely functions that perform a comparison between a given property and a specified value. In addition to the standard operators `<`, `<=`, `>`, `>=`, `==` and `/=`, the library offers `any_bit` and `all_bit` functions to check whether some (or all) bits of a given mask are set. As an example, the expression `any_bit ip_tos 0x3f` is a valid Enif–Lang predicate that tests whether any of the DSCP bits are set in the packet.

**Filters.** Filters are effectful functions that break the pipeline processing when the packet does not match a given condition. The Enif–Lang library is equipped with a wide range of filters for the most common protocols. In a nutshell, a filter is a very simple monadic function, whose output action can be either Pass or Drop. Examples of common filters are `ip`, `tcp`,

udp, port, src_port, dst_port, addr, src_addr, dst_addr, etc.

**Monadic functions.** Monadic functions take an arbitrary number of arguments and a packet, and return a packet with an associated action. Currently, the available actions are: Pass, Drop (used by filters), Broadcast, Dispatch, Steer and DoubleSteer (used by steering functions).

Common monadic functions are `when` and `unless`, used in conditional statements as well as the family of steering functions, such as `steer_flow`, `steer_p2p`, `steer_link`, etc., used to balance the traffic among multiple endpoints with different flow consistency guarantees.

## III. Processing Pipelines

Actions can be composed together by means of the `do` notation (slightly different from that of Haskell), or through the Kleisli operator `>->`, and follow the precise rules of composition set forth in [9]. The overall expressiveness of Enif–Lang allows to build even complex pipelines through a very concise grammar. As a first example, the following simple program:

```
main = udp >-> log  >-> kernel
```

describes a stateless filter that allows UDP packets to be logged and passed back to the kernel of the operating system.

The next snippet of code, instead, provides a more complex example in which the processing pipeline takes advantage of a per–computation state.

```
http = dst_port 80
pass_to_kernel =
    when (has_state http_traffic)
         kernel
mirror_to_port =
    when (has_state other_traffic)
         (forward eth2)
process = if (not is_tcp)
            then drop
            else do pass_to_kernel
                    mirror_to_port)
http_traffic = 1
other_traffic = 2
main =  do
  if http
    then put_state http_traffic
    else put_state other_traffic
  process
```

From the language point of view, functions like `put_state`, `get_state` and the predicate `has_state` are implemented as properties and act as an implicit extra parameter for all of the pipeline functions. In the current implementation (within PFQ) such a state is eventually passed to the kernel as a mark of the socket–buffer to enable further manipulations of the packet with `netfilter`. However, the usage of the per–packet state is somewhat limited as it vanishes after the packet computation expires. As such, it cannot be used in processing pipelines that require the storage of stateful information across different packets. The solution of this issue is represented by per–flow persistent states and is described in the next section.

### A. Stateful Pipelines

Generally speaking, stateful operations in parallel architectures require an effective management of potential data sharing across multiple threads of execution to avoid race conditions. In the case of stateful processing pipelines, this would be the case of different packets belonging to the same flow but processed by different cores.

It turns out that it is possible to tackle the problem in a general and effective way by restricting the association of stateful information to packet flows only. As a consequence, in many practical applications it is to possible to partition *a–priori* the traffic and let all the computations process flows of packets in parallel and total isolation.

The central concept is here represented by a suitable definition of packet flow. Packet flows are defined through flow–keys (e.g. IP addresses, canonical 5–tuples, and so on). In the Enif–Lang context, a generic flow–key consists of the concatenation of an arbitrary number of packet header fields. Different pipelines may operate onto different types of flows, all of them specified by their own flow–keys. The bitwise intersection of all such keys represents the common flow–key and can be used upon hashing at the splitting stage (see Figure 1) to distribute packets to functional engines. Once traffic is split, the current abstraction guarantees that packets of a flow are processed in the Enif–Lang stage sequentially and on a single core. This automatically ensures the flow consistency for all the packets and their related states and prevents from data–sharing among cores. Notice that multiple Enif–Lang programs can instead run in parallel (on different cores) thanks to the immutability of packets.

However, it is worth noticing that not all configurations can be parallelized as the common flow–key might be empty. This special case can be conveniently handled by partitioning the applications in clusters of common sub–keys and by introducing shallow copies of packets to feed each of them.

From the language point of view, Enif–Lang provides a persistent per–flow state that is automatically handled by the underlying abstract processing model. A per–flow persistent state is a state shared among all packets that belong to a certain flow. Such a state information is stored in associative flow–maps, indexed by their own flow–keys.

The Enif–Lang library provides several functions for the per–flow state management. In general, all such functions take a flow map object that contains a table identifier and a flow–key definition. Furthermore, the language offers a set of predefined keys as well as utilities for building custom keys through concatenation of arbitrary header fields. In particular, the function `set_fstate` is used to set the value of the state associated with the flow of a packet. Instead, the function `get_fstate` retrieves the value of the state associated with the flow the packet belongs to. Other additional functions,

such as `incr_fstate`, `decr_fstate`, `add_fstate`, etc. are used to update the state information. It is worth noticing that Enif–Lang does not limit the number of state tables whose maximum number is instead enforced by the underlying implementation.

## IV. Use–cases

This section presents the use of Enif–Lang in two practical applications. The first application is an example of stateless processing and consists of a simple load–balancer that forwards packets to a cluster of network devices or local applications running Deep Packet Inspection. The second example, instead, provides the Enif–Lang implementation of the simple stateful firewall based on the *port knocking* scheme.

### A. Stateless processing

DPI applications typically take advantage of DNS packets to build classification trees and improve application recognition. As such, the following load–balancer broadcasts DNS packets to all DPI workers and randomly spreads the remaining packets according to `steer_p2p` stering function that preserves layer 3 symmetric flow consistency.

```
is_dns = has_port 53
main = if is_dns
       then broadcast
       else steer_p2p
```

### B. Stateful processing

In the following, the simple port knocking application is presented. The problem, already described in [7], is to create a simple firewall that permits certain flows to pass only if a known sequence of TCP packets hit predefined ports (port numbers 5123, 6234, 7345 and 8456 in the example).

The example uses a couple of tables. The first table lists the authorized flow and is implemented as an associative map based on the classic 5-tuple key (predefined as TUPLE_5). The second table, instead, is based on the 3-tuple keys (IP_SRC, IP_DST and PROTOCOL, defined as TUPLE_3) and implements the state machine for tracking the knocking sequence. Any time the expected destination port is found, the state is updated through the function `next_if` until it reaches the state value 4 which opens the firewall and the corresponding flow is authorized in the `auth_flow` table.

```
auth_flow   = flow_map 0 TUPLE_5
knock_table = flow_map 1 TUPLE_3
next_if pred =
    if pred
    then incr_fstate knock_table
    else set_fstate knock_table 0
main =
  if (get_fstate auth_flow)
  then kernel
  else case_of (get_fstate knock_table) $ with
   [ 0 ~> next_if (dst_port. == 5123) >-> drop
   , 1 ~> next_if (dst_port. == 6234) >-> drop
   , 2 ~> next_if (dst_port. == 7345) >-> drop
   , 3 ~> next_if (dst_port. == 8456) >-> drop
   , 4 ~> do
            next_if (dst_port. == 22)
            when (set_fstate knock_table. == 0)
            (set_fstate auth_flow 1 >-> kernel)
  ]
```

## V. Conclusion

The paper presents Enif–Lang a functional language for stateful processing pipelines on multi–core platforms. The language is grounded upon the theoretical framework of monads borrowed from Category Theory and provides a formal description of a generic processing machine for network traffic manipulation. In particular, the programming model allows the automatic parallelism of processing pipelines thanks to the immutability of packets, enforced by the functional paradigm, and to a suitable state aware traffic splitting across multiple computation resources. A few examples are reported in the paper to exemplify the language practical usage. The description of two simple but real applications is also given to show the expressiveness of the description of both stateless and stateful processing pipelines.


## Acknowledgment

This work has been partly supported by the EU project Behavioral Based Forwarding (BEBA, Project ID: 644122).


## References


[1] F. Fusco and L. Deri, "High speed network traffic analysis with commodity multi–core systems," in *Proc. of IMC '10*. ACM, 2010, pp. 218–224.

[2] L. Deri, "PF_RING ZC (Zero Copy)." [Online]. Available: http://www.ntop.org/products/packet-capture/pf_ring/pf_ring-zc-zero-copy/

[3] L. Rizzo, "Netmap: a novel framework for fast packet i/o," in *Proc. of USENIX ATC'2012*. USENIX Association, 2012, pp. 1–12.

[4] "DPDK." [Online]. Available: http://dpdk.org

[5] N. Bonelli, S. Giordano, and G. Procissi, "Network traffic processing with PFQ," *IEEE Journal on Selected Areas in Communications*, vol. 34, no. 6, pp. 1819–1833, June 2016.

[6] N. McKeown *et al.*, "Openflow: Enabling innovation in campus networks," *SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 2, pp. 69–74, Mar. 2008.

[7] G. Bianchi, M. Bonola, A. Capone, and C. Cascone, "Openstate: Programming platform-independent stateful openflow applications inside the switch," *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 2, pp. 44–51, Apr. 2014.

[8] "The BEBA (Behavioral Based Forwarding) H2020 EU project." [Online]. Available: http://beba-project.eu/

[9] N. Bonelli, S. Giordano, G. Procissi, and L. Abeni, "A purely functional approach to packet processing," in *Proceedings of the Tenth ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ser. ANCS '14. New York, NY, USA: ACM, 2014, pp. 219–230.

[10] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker, "P4: Programming protocol-independent packet processors," *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 3, pp. 87–95, Jul. 2014.

[11] C. Monsanto, J. Reich, N. Foster, J. Rexford, and D. Walker, "Composing software-defined networks," in *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, ser. nsdi'13. Berkeley, CA, USA: USENIX Association, 2013, pp. 1–14. [Online]. Available: http://dl.acm.org/citation.cfm?id=2482626.2482629

[12] "The Frenetic Project." [Online]. Available: http://frenetic-lang.org/

[13] W. de Bruijn, H. Bos, and H. Bal, "Application-Tailored I/O with Streamline," *ACM Trans. Comput. Syst.*, vol. 29, no. 2, pp. 6:1–6:33, May 2011. [Online]. Available: http://doi.acm.org/10.1145/1963559.1963562

[14] "VPP." [Online]. Available: https://wiki.fd.io/view/VPP

[15] "Linux Enhanced BPF (eBPF) Tracing Tools." [Online]. Available: http://www.brendangregg.com/ebpf.html