On Network Policy Composition and the Maple SDN Controller

Lewen Yu Advised by Yang Richard Yang

Abstract

Software-Defined Networking promises a simple conceptual programming model to configure a complex network. In reality, the difficulty in managing flow tables and producing correct forwarding rules scales exponentially with the size of the network. Maple[2] is a programming system that enables SDN developers to write a single algorithmic policy \( f \) without worrying about such low-level details. As policies themselves become more complicated, it becomes imperative to offer composition semantics for modularity. We explore a framework to efficiently implement policy composition in Maple.

1 Background

Definition 1. A packet is a general unit of data transmitted between endpoints. It can be thought of as a collection of metadata and a payload. The collection of metadata, often referred to as the packet header, enables a variety of features, such as error correction and host addressing. The payload is a stream of serialized bits containing data from some upper-layer that needs to be transmitted to some other host.

Definition 2. A host is a computer connected to a network, or some endpoint on the network topology graph. A host offers and requests information and services to and from other hosts on a network.

Definition 3. A switch is a device that links segments of a network and other hosts or network devices. It receives messages from any device connected to it and transmits the message only to the intended recipient device.
1.1 Software-defined networking

Software-Defined Networking (SDN) is a recent development in computer networking. Among SDN’s chief tenets are decoupling the control and the data plane of the network, centralizing network control logic, and enabling a simple programming model for a complicated network. SDN offers promises of quicker infrastructure deployment, a high degree of flexibility and programmability of the network, and novel ways to write applications.

Definition 4. The control plane is characterized as the system with a high-level view of the network topology and decides where traffic in a network should be sent.

Definition 5. The data plane is characterized as the system that forwards traffic to some destination.

1.2 OpenFlow

OpenFlow[5] is a set of open specifications that has enabled major progress within the SDN research community. In OpenFlow, the flow table, filled with flow entries, serves as the abstraction for the data plane. The controller is able to install flow entries in switches through the OpenFlow protocol, and switches will forward to the controller any packets that don’t match entries in its flow tables. The separation of central control and distributed data forwarding intends to produce a more flexible way to control and program network traffic.

Definition 6. A controller is some central entity that can make authoritative decisions as to where packets go.

Definition 7. A flow table is a collection of flow entries in a switch. A flow table, and thereby, the switch can be dynamically reprogrammed through the addition and removal of flow entries.

Definition 8. A flow entry is a collection of fields, the most important of which are the match, priority, and action or instruction fields. The match specifies bits and other identifiers to match against packet headers. Matching packets will execute its corresponding actions or instructions. If a packet does not match any flow entries, it will be redirected to the controller.
1.3 Maple

Maple[2] is a programming system that enables the SDN developer to write a single algorithmic policy $f$ that simply defines the core logic of the controller. By only writing $f$, the SDN developer does not have to become mired in the difficult process of writing correct forwarding rules and pushing them to switches. Maple executes the algorithm in a tracing runtime and gathers information about decisions made by $f$ to generate and maintain a trace tree. Finally, Maple can compile the trace tree into a correct and optimized forwarding table, and automatically distribute it to switches.

Definition 9. A trace is an ordered list of decisions made by some policy and its result, given a particular set of inputs.

Definition 10. A trace tree is a tree generated from a set of linear traces from the same policy. It is a compact representation of the logic and can be used to compile forwarding rules for switches.

1.4 A Motivating Example

Currently, Maple specifies the use of a single algorithmic policy $f$. However, in certain cases, it is helpful to think of multiple policies as composable units $f_1, f_2, f_3, ... f_n$ that can be mixed, matched, and merged.

In the general case, Maple provides language-dependent composition semantics. For example, let us consider an SDN developer, Bob, who is in charge of Yale University’s network services. Bob has several different types of policies that he wants to deploy on the university network. For instance: he has a security policy that drops packets coming from a blacklisted host, a simple routing policy that forwards to the next hop, and a traffic monitoring policy that simply counts the number of packets with a particular attribute:
--- $f_1$: security ---

```python
blacklistIPs = [
    123.123.123.123,
    199.10.40.8,
    5.0.133.7
]

if (packet.isIPv4 and
    packet.ipsrc in blacklistIPs)
    return drop
```

--- $f_2$: routing ---

```python
path = dijkstra(packet.ethsrc,
    packet.ethdst)
return path
```

--- $f_3$: monitor ---

```python
# counts the number of SIP packets
if (packet.isUDP and
    (packet.tpDstPort == 5060 or
     packet.tpDstPort == 5061))
    return increment-count
```

Where $f_1$ and $f_2$ returns a path that will be resolved to a sequence of next-hop ports for switches and $f_3$ returns an action that instructs the switch to increment a hardware counter on the ingress switch. All switch-level rules will be generated and pushed out by Maple.

Bob wants to deploy all three policies to the campus-wide network. Ideally, Bob would keep $f_1$, $f_2$, and $f_3$ as distinct logical modules to be reused and interchanged, and make use of built-in composition semantics to specify how the three policies will be implemented in the network. In fact, we will introduce such composition extensions to Maple later, but for now, Bob manages to refactor his three policies into a single policy $f'$ by simply appending them:
$f'$: combined policy

```python
blacklistIPs = [
    123.123.123.123,
    199.10.40.8,
    5.0.133.7
]

monitor = false

if (packet.isIPv4)
    # first apply security policy (f1)
    if (packet.ipsrc in blacklistIPs)
        return drop

    # do check for monitor policy (f3)
    # since a UDP packet must be
    # contained an IPv4 packet
    if (packet.isUDP and
        (packet.tpDstPort == 5060 or
         packet.tpDstPort == 5061))
        monitor = true

    # if not dropped, continue to
    # apply routing policy (f2)...
    path = dijkstra(packet.ethsrc,
                    packet.ethdst)

    # finally, merge results with
    # monitor policy (f3)
    if (monitor)
        return [path, increment-count]
    else
        return path
```

Where the return result if a packet is a UDP packet (line 17) is a list of actions to be executed at the switch. In this case, the controller will install rules for next-hop forwarding on all switches that form the shortest path and increment the counter only at the ingress switch, thereby maintaining
correctness for the monitor policy \( (f_3) \).

We note that because \( f_1 \), \( f_2 \), and \( f_3 \) are orthogonal, the order in which the controller evaluates each policy in \( f' \) is irrelevant so long as the correctly merged result is returned. We also note that, in \( f' \), blacklisted packets are not monitored.

Though this refactored policy is correct and compact, there is one highly undesirable side-effect—a non-trivial increase in the size of the resulting trace tree and, thus, the size of the generated ruleset.

We can see that this is extremely wasteful because whether the packet is an UDP packet or a TCP packet is completely orthogonal to its source and destination host addresses, but they are still recorded as dependencies to the host address lookup further down in the trace tree.

For a network with \( n \) hosts, there will be at most \( n^2 \) flow table rules generated by the routing policy \( (f_2) \). Naively composing it with security \( (f_1) \) and monitor \( (f_3) \) will roughly quadruple the total number of flow table rules pushed to switches! Additional policies that read header fields orthogonal to those already read will, in the best case, increase the number of flow table rules by a factor of \( 2^m \), where \( m \) is the number of new fields read.

In the rest of this paper, we lay the foundation for a much more space-efficient method for policy composition in Maple by using multiple flow tables as specified in OpenFlow 1.1 and above, and we describe our efforts in implementing a naive composition framework in Maple on OpenFlow 1.0.

## 2 Composition Semantics

First, we must clarify and define what we mean by composition. In particular, based on work by Monsanto, Foster et al. [3, 4, 6], we consider two meaningful kinds of policy composition, which we specify with the following operators:

1. **ApplySequential**: apply policies \( f_1, f_2, \ldots, f_n \) to a packet \( pkt \) in order so that actions from later policies will override the same type of actions from earlier policies. That is, the accumulated actions from **ApplySequential** has set semantics.

2. **ApplyAll**: apply all policies \( f_1, f_2, \ldots, f_n \) to a packet \( pkt \) as if each policy had its own copy of packet \( pkt \).

Let’s consider a simple example for each operator. Suppose Bob the Yale SDN programmer wanted to drop all packets from the blacklist but
route all non-blacklisted packets (that is, a combination of $f_1$ and $f_2$), then the correct composition operator would be \texttt{APPLYSEQUENTIAL}. To ensure correctness, Bob would have to specify the policy for dropping blacklisted packets ($f_1$) after the policy for routing all packets ($f_2$), as follows: \texttt{APPLYSEQUENTIAL}($f_2, f_1$). This ensures that the routed packets with a blacklisted source address are dropped.

Now suppose that Bob still wanted to drop packets from the blacklist ($f_1$), but now he also wanted to keep a record of all UDP packets passing through the network ($f_3$), including those that are blacklisted. In this case, the correct operator would be \texttt{APPLYALL}. This operator would in principle apply both $f_1$ and $f_3$, each with a separate instance of the packet, ensuring that even if a packet is dropped, the switch dropping the packet still increments its counter.

Because both composition operators provide useful and meaningful semantics, it is important that Maple supports both of them.

Additionally, we must specify semantics for composing routing policies. In particular, should some packet be routed along the same path in $n$ routing policies that are to be applied simultaneously (that is, applied using the \texttt{APPLYALL} operator), then we would expect exactly $n$ copies of that packet to arrive at the destination switch. This expectation is applied for both unicast and multicast routing policies.

Likewise, should a routing policy $p_r$ be composed with some other non-routing policy $p_{nr}$ using the \texttt{APPLYALL} operator, then we declare that $p_r$ should be applied to all switches along the path or tree specified by the routing policy $p_r$, in addition to the routing behavior of $p_r$ itself.

## 3 Efficient Policy Composition

Since OpenFlow 1.1, it is possible to specify multiple FIB tables in the switches[1]. This suggests a very natural solution to policy composition—build one FIB table for each policy! Instead of naively merging multiple traces into one much longer trace, the composition operator would return separate traces. Accordingly, Maple would simultaneously maintain multiple trace trees, with each tree corresponding to a single policy. Maple would compile each trace tree into its own flow table, while specifying the order to walk the flow tables and the semantics to accumulate actions from each matching entry. Below, we describe one possible implementation using OpenFlow 1.1
semantics and prove its correctness and compactness.

3.1 Summary of Relevant Changes since OpenFlow 1.1

- **Pipeline processing**: OpenFlow 1.1 introduces pipeline processing, where each OpenFlow switch contains multiple flow tables. As a packet moves down this pipeline, its corresponding metadata and action set accumulates. After the very last flow table, the entire action set is executed.

- **Instructions**: Instead of specifying actions in forwarding entries, OpenFlow 1.1 forwarding entries now specify instructions instead. Instructions can be thought of as meta-actions. In particular, the Apply-Actions instruction immediately applies a list of actions to the current packet, whereas the Write-Actions instruction merges a list of actions into the action set to be executed further down in the pipeline. We note here that the semantics for the Write-Actions instruction is particularly suitable for implementing the \texttt{APPLYSEQUENTIAL} operator, and likewise, that the Apply-Actions instruction can be used to implement the \texttt{APPLYALL} operator. The Goto-Table instruction can be used to specify a monotonic order in which to traverse multiple flow tables.

3.2 Implementation Framework

On the policy side, \texttt{APPLYALL} and \texttt{APPLYSEQUENTIAL} executes each policy separately, and return separate (trace, result) pairs for each policy. The order in the (trace, result) pairs are appended is fixed and deterministic, so that the Maple controller can easily deduce the order to build and compile the trace trees.

The core Maple controller augments the trace tree from a linear trace by simply walking the trace tree and extending its branches where necessary and writing commands at the leaves. This algorithm, called \texttt{AUGMENTTT} (not reproduced in this paper), must be slightly altered to enable the maintenance of multiple trace trees. In particular, some commands must be pruned to produce correct routing behavior for \texttt{APPLYALL}.

As an example, consider two separate routing policies \(g\) and \(h\), that happen to produce the same path \([p_0, p_1, p_2, \ldots, p_n]\) specifying a sequence of out-
ports in switches on the path for some packet \textit{pkt}. If Maple were to naively construct the two trace trees without pruning the commands of the second trace tree, then the compiled flow tables would specify that packet \textit{pkt} be forwarded out of the same port twice per switch on the path. Though the expected behavior is that exactly two copies of \textit{pkt} will arrive at the destination switch, but the forwarding tables compiled from the unpruned trees would produce $2^n$ \textit{pkts} at the destination.

To fix this, we specify the following pruning behavior when composing multiple routing policies simultaneously:

- **PruneUnicast:** If the resulting command from a policy trace is a unicast, then if applicable, prune all links in the unicast path that also exist in routing commands from all prior traces for this particular match. We shall refer to such links as redundant links. It follows that we must fix an order for traces returned by policies and trace trees maintained by Maple. There is one exception: if all links in the unicast path are redundant, then it follows that the entire path has been specified in some previous trace. Therefore, only the final link in the path shall be retained to reproduce the intended behavior of the same packet being sent twice along the same path.

- **PruneMulticast:** If the resulting command from a policy trace is a multicast, then if applicable, prune all links in the multicast tree that also exist in routing commands from all prior traces on this particular match. As with pruning unicast commands, we do not prune any links going to some edge switch if the path going to that edge switch in the multicast tree already exists in the union of prior routing commands.

By pruning all redundant links and all but the last link in an overlapping path, the packet is duplicated either right before its path diverges from another packet travelling through the same path or at the last possible moment before it arrives at its destination. This design has the benefit of minimizing congestion and improving fault-tolerance simply because the reproduced packets travel through fewer links and therefore have lower chances of being dropped.

The \texttt{Prune} algorithm, which follows the behavior prescribed above, is used in tandem with \texttt{AugmentTT} to prune commands from policy traces before they are appended to the leaves of trace trees. We only prune if the semantics is under the \texttt{ApplyAll} operator.
Algorithm 1 PRUNE(cmd, prevcmds)
1: if cmd is unicast(links) then
2:    routeUnion(graph) ← prevcmds.allRoutes
3:    prunedLinks ← PRUNEUNICAST(graph, links)
4:    cmd ← unicast(prunedLinks)
5: else if cmd is multicast(tree) then
6:    routeUnion(graph) ← prevcmds.allRoutes
7:    prunedTree ← PRUNEMULTICAST(graph, tree)
8:    cmd ← multicast(prunedTree)
9: end if
10: return cmd

Algorithm 2 PRUNEUNICAST(graph, links)
1: if graph contains all l ∈ links then
2:    prevLinks : lastLink ] ← links
3:    prunedLinks ← unicast([ lastLink ])
4:    return prunedLinks
5: else
6:    prunedLinks ← unicast([ l, l ∈ links, l /∈ graph ])
7:    return prunedLinks
8: end if

Algorithm 3 PRUNEMULTICAST(graph, links)
1: ( source, links, dests ) ← tree
2: prunedLinks ←
3: for dst ∈ dests do
4:    links ← path from source to dst
5:    prunedLinks ← prunedLinks ∪ PRUNEUNICAST(graph, links)
6: end for
7: prunedTree ← tree(source, prunedLinks, dests)
8: return prunedTree

Where prevcmds is a temporary record used to collect policy results from the traces for the current match. Consequently, allRoutes is a field in prevcmds containing a union of routing policy results.

After the trace trees are correctly augmented, Maple will proceed to generate and distribute rules to switches. This process is handled by an al-
algorithm called BuildFT (not reproduced in this paper), which recursively walks the trace tree and compiles a list of (priority, match, action) triples representing the corresponding FIB entry on a switch. Because the definition of flow entries are slightly different in OpenFlow 1.1, and because composition requires multiple flow tables, we present an update to the BuildFT algorithm: BuildFT’:

Algorithm 4 BuildFT’(t, match, list, op, nextTable)
1: if \( type_t = L \) then
2:   if nextTable exists then
3:     instr ← \{ apply(action_t, op), goto(nextTable) \}
4:     list.append(match, instr)
5:   else
6:     instr ← \{ apply(action_t, op) \}
7:     list.append(match, instr)
8:   end if
9: else if \( type_t = \Omega \) then
10:   instr ← \{ apply(ToController, op) \}
11:   list.append(match, instr)
12: else if \( type_t = T \) then
13:   instr_+ ← \{ apply(action_t, op) \}
14:   m = match.append(attr_t, instr_+)
15:   BuildFT’(t_+, m, list, nextTable)
16:   instr_− ← \{ apply(ToController, op) \}
17:   list.append(m, instr_−)
18:   BuildFT’(t_−, match, list, nextTable)
19: else if \( type_t = V \) then
20:   for \( v \in \text{dom}(\text{subtrees}_t) \) do
21:     BuildFT’(subtrees_v, match.append(attr_t, v), list, nextTable)
22:   end for
23: end if

BuildFT’ will generate a list of flow entries intended to occupy a single flow table for a particular trace tree. BuildFTs (not shown) will simply call BuildFT’ in sequence for each trace tree maintained by Maple to generate the complete forwarding entries to be prioritized and pushed to switches.

apply (omitted) will produce a corresponding OpenFlow instruction for the command specified in every trace tree leaf. In particular, it will emit the
Apply-Actions instruction if the semantics is under the APPLYALL operator \((op)\), and it will emit the Write-Actions instruction if the semantics is under the APPLYSEQUENTIAL operator. Additionally, it will emit the Goto-Table instruction if it is processing all but the last tree in the sequence.

### 3.3 Correctness

**Theorem 1.** The Prune algorithm will not cause more nor fewer than the expected number of packets to arrive at the destination.

**Proof.** Suppose that at a destination is expecting \(n\) packets, we receive \(> n\) packets. Then it must follow that the algorithm left some unnecessary redundant link \(l_u\) while pruning. This cannot be the case, because by construction, all redundant links are removed with the exception of the final links in paths that are completely redundant. If such links are removed, then we would be receiving \(\leq n\) packets. This is a contradiction. We will receive no more than \(n\) packets.

Now suppose that some link \(l_i\) that has been pruned causes the destination to receive one fewer packet than expected. By construction, this cannot be true because all removed links must be redundant by definition, so the packet will pass through link \(l_i\) at least once. The only case where removing a redundant link will reduce the number of packets received to be fewer than expected is at the final link of a completely redundant path. Again, by construction, this link will not be removed so this cannot be true.

Therefore, pruned links will not cause a deviation in the number of packets actually received from the number of packets expected. 

### 3.4 Analysis

The PruneUnicast algorithm runs in time \(O(n^2)\) for an \(O(n^2)\) set membership lookup algorithm, where \(n\) is the number of links. The PruneMulticast algorithm runs in time \(O(n^3)\) since it depends on PruneUnicast, where \(n\) is the largest number of links in the tree. Consequently, the Prune algorithm runs in time \(O(n^3)\). However, by using this, we store \(m\) trees with linear space requirement \(O(n)\), where \(n\) is the size of the number of nodes in the largest tree and \(m\) is the number of policies.
4 Software-based composition

Because many switches today support OpenFlow 1.0, we wish to extend the same composition operators APPLYSEQUENTIAL and APPLYALL for all hardware. Below we describe a naive software composition implementation that results in a single trace tree in Maple. This a graceful fallback option for hardware that do not support the multiple table workflow. As a result, it will experience the exponential explosion in the size of the trace tree and generated rules.

4.1 Implementation

There are multiple ways to implement this graceful fallback option. One way is to implement the merging functionality in Maple, which makes for a uniform APPLYSEQUENTIAL and APPLYALL specifications from the policy side. The alternative way, which is what we implemented, would shift the merge logic to the developer side. The advantage of such a design is that it requires no change to the Maple core codebase. However, it must be re-written for each programming language Maple supports.

In this approach, we offer alternative definitions for APPLYALL and APPLYSEQUENTIAL whereby they return one trace and one result, merged from executing multiple user policies and peeking at their traces.

Algorithm 5 APPLYALL\((\text{policies})\)

1: fix an order such that \(\text{policies} \rightarrow \{p_0...p_n\}\)
2: for \(i \leftarrow 0, n\) do
3: \((\text{trace}_i, \text{action}_i) \leftarrow \text{APPLYPOLICY}(p_i)\)
4: end for
5: \(\text{mergedTrace} \leftarrow []\)
6: \(\text{mergedActions} \leftarrow {}\)
7: for \(i \leftarrow 0, n\) do
8: \(\text{mergedTrace}.\text{append}(\text{trace}_i)\)
9: \(\text{mergedActions} \leftarrow \text{mergeActions}(\text{mergedActions}, \text{action}_i)\)
10: end for

Where \(\text{mergeActions}\) (omitted) correctly prunes all routing actions before the merge. This produces the correct behavior given overlapping paths or trees from routing policies.
Algorithm 6 \textbf{APPLYSEQUENTIAL}(\textit{policies})

1: fix an order such that \textit{policies} $\rightarrow \{p_0...p_n\}$
2: for $i \leftarrow 0, n$ do
3: \hspace{1em} $(\text{trace}_i, \text{action}_i) \leftarrow \text{APPLYPOLICY}(p_i)$
4: end for
5: mergedTrace $\leftarrow []$
6: mergedActions $\leftarrow \{\}$
7: for $i \leftarrow 0, n$ do
8: \hspace{1em} mergedTrace.append(\text{trace}_i)
9: \hspace{1em} mergedActions $\leftarrow$ mergedActions $\cup$ \text{action}_i
10: end for

4.2 Correctness

\textbf{Theorem 2.} For the \textbf{APPLYALL} operator, assuming that the set of policies being applied do not change each other’s behavior in some direct or indirect manner, then sequentially appending traces for such policies, even if each policy is being executed in a separate thread, does not produce nondeterminism or unexpected behavior.

\textit{Proof.} By straightforward reasoning from the fact that the set of policies are non-interacting, sequentially executing the policies will produce the same result as interleaving their execution in any arbitrary manner. Thus, it follows that the complete trace of some policy followed by the complete trace of another policy will not produce any nondeterminism.

Finally, because we fix a deterministic order to execute and append the traces, the Maple controller will produce a well-formed trace tree that can be correctly augmented. \hfill $\Box$

5 Conclusion

In this paper, we explore a framework for efficiently implementing composition operators in Maple. We believe composition can have many uses in SDNs, though it is especially useful as a tool for building complex policies without the complexity in constituent modules. Future work includes implementing efficient composition operators using OpenFlow 1.1. Further investigation into higher-order and freeform composition is encouraged. For example, let $g = \text{APPLYALL}(f_1, f_2)$ be a policy. Then let $h = \text{APPLYSEQUENTIAL}(g, f_3)$
Being able to recursively compose policies could result in a powerful and expressive form of policy specification.

5.1 Acknowledgements

I wish to thank Richard Yang for introducing me to computer networks and graciously advising me throughout this project, and whose advice and patient guidance drove this to be where it is today. I would like to thank Andreas Voellmy, whose great advice helped define the core of this project. I would also like to thank Michael “Fitz” Nowlan and Junchang Wang, for their incredible patience and expertise—and providing me with the best programming-filled semester I’ve had Yale. Cheers!
References


