Sequence Abstractions for Flexible, Line-Rate Network Monitoring

Andrew Johnson  
Princeton University

Ryan Beckett  
Microsoft Research

Xiaoqi Chen  
Princeton University

Ratul Mahajan  
University of Washington

David Walker  
Princeton University

Abstract

We develop FLM, a high-level language that enables network operators to write programs that recognize and react to specific packet sequences. To be able to examine every packet, our compilation procedure can transform FLM programs into P4 code that can run on programmable switch ASICs. It first splits FLM programs into a state management component and a classical regular expression, then generates an efficient implementation of the regular expression using SMT-based program synthesis. Our experiments find that FLM can express 15 sequence monitoring tasks drawn from prior literature. Our compiler can convert all of these programs to run on switch hardware in a way that fits within available pipeline stages and consumes less than 15% additional header fields and instruction words when run alongside switch programs.

1 Introduction

Many network management tasks involve recognizing and reacting to a user-defined sequence of packets. Such sequence monitors can enforce security policies, prioritize traffic, mitigate attacks, ensure protocol compliance, and more. For example, they can identify and de-prioritize video flows using a fingerprint based on successive packets to improve the network for other traffic [23, 24] or verify that network clients faithfully implement protocols such as the Dynamic Host Configuration Protocol (DHCP) by observing the protocol exchange [28].

Ideal sequence monitors are i) flexible: can express a broad range of monitoring tasks; and ii) line-rate: can perform all processing directly in the data plane (hardware). Being line-rate allows sequence monitors to analyze all traffic passing through the switch, without needing the switch CPU or a remote server. Switch CPUs cannot process all packets at line rate, and using remote servers incurs high network overhead and reaction delays.

Existing sequence monitors sacrifice either flexibility or line-rate processing. Systems such as Aragog [28] can express most sequence monitoring tasks, but they run entirely in software (i.e., are not line rate). Programmable switches based on Protocol Independent Switch Architecture (PISA) [5] enable hardware-based sequence monitoring, but are programmed in languages such as P4 [4, 26] that are too low level, making it hard to express and debug sophisticated tasks [15, 29, 32]. Hybrid systems like Marple and Sonata [11, 21] run only partially on switches. Their core abstractions focus on data transformations such as filter, map, and fold operations rather than packet sequences. They are line-rate only if strong restrictions are placed on the allowed functions or if hardware could be redesigned [21].

In this paper, we present an abstraction of a packet sequence pattern that is both flexible and compiled directly to PISA-based hardware. It enables line-rate sequence monitoring without mirroring traffic to the switch CPU or a central controller. Recognizing patterns for sequence monitors directly in hardware is difficult due to stringent data access constraints in current programmable switches. PISA-based switches process packets using a series of stages. Each stage contains its own local memory and a number of arithmetic and logic units (ALUs) to perform computation. While some sequence monitors such as packet counting fit this architecture naturally, others that require tracking state across multiple packets are significantly more challenging to realize.

Our system, called FLM, allows programmers to write sequence monitors using a high-level, pattern-based language. Patterns are specified as regular expressions over packets, with the added ability to record packet parameters for later use. FLM programs can trigger local switch actions immediately upon matching a pattern, and they can monitor packets at any desired granularity (e.g., flow, host).

We convert FLM programs into imperative code using a novel core data structure, which represents a state machine maintaining a variable environment. We transform operations on this data structure into operations on PISA switch registers by carefully dividing them into variable update and transition code for a deterministic finite automaton (DFA), reflecting the pattern’s match progress. Our implementation prioritizes
line rate execution on existing network hardware like the Intel Tofino \cite{13}, for any accepted packet sequence. We have formally proven that our compilation process from patterns to pipeline stages preserves the original pattern semantics.

We evaluated FLM by encoding 15 monitoring tasks drawn from prior work \cite{10, 15, 18, 21, 24, 25, 28, 29, 31, 32}. We find that we can express all of these tasks in 10-41 lines of FLM code, which demonstrates the flexibility of FLM. We also find that we can compile all of these tasks to the Intel Tofino switch, which demonstrates FLM’s ability to provide line-rate monitoring. All of these tasks fit within the number of stages on the switch and consume less than 15% of additional metadata memory or instruction words when run alongside switch programs.

In summary, this work makes three main contributions:

- **FLM**, a language to express sequence monitors that can run at line-rate on a switch with a novel definition of a pattern syntax and semantics.

- Provably correct compilation from an FLM program to a state machine representation that runs on PISA hardware using a minimal number of stages.

- Evaluation that shows that FLM can express a wide variety of sequence monitoring tasks and compile them to existing network hardware.

## 2 Background

Our programming abstractions are broadly applicable to contexts where real-time recognition of event sequences is useful, (programmable ASICs, FPGAs, NICs, or software switches). Our implementation is designed for the PISA model. PISA architectures rely on a series of stages. To achieve a high processing rate and avoid memory access hazards such as contention, each stage has its own nearby memory region. Due to this memory layout, data can only be accessed by a single stage, and can only flow forward in the pipeline by changing the packet being processed. Within each stage, the switch is able to perform a Read-Modify-Write instruction on a value stored in its memory, where the modify step is specified by a micro-program called a Register Action. A core problem we tackled is writing a DFA transition function in such a way that it fits into one Register Action in order to be applied to packets at line rate. In this work, we consider a monitor recognizing patterns in packets passing through a single switch pipeline. A switch containing multiple pipelines operating in parallel could contain multiple monitors.

To simplify our implementation, we build on Lucid \cite{25}, an event-driven programming language for PISA switches. Lucid programs declare events (i.e., notifications of data plane packet arrival or network control signals) and corresponding handlers (i.e., code to react to events). Lucid programs are compiled to P4 code that runs on PISA switches. Both P4 and Lucid are useful intermediate languages simplify our implementation, but are not critical for FLM’s key abstractions.

## 3 Example walk through

In this section, we walk through an example monitoring task for the DHCP protocol step-by-step to show how FLM enables network programmers to more easily build flexible, line-rate packet sequence monitors.

### 3.1 DHCP Anomaly Detection

Suppose a network operator wants to verify that DHCP, which enables clients to lease IP addresses from a server, is not being misused. DHCP begins with the client broadcasting a "Discover" message, to which the server responds with an "Offer" message with available IP addresses. The client sends a "Request" message for an address, which is confirmed with an "Acknowledge" from the server. The client is then expected to use the acknowledged IP until the end of its lease.

The operator wants to ensure that clients only use their assigned IP. This monitoring task can be performed on the access switch that processes all client communication, including that with the DHCP server. Misuse would appear as a packet sequence belonging to a client, identified by its MAC (link-layer address), of some number of packets (the DHCP protocol), a DHCP "Acknowledge," and then a packet whose source IP does not match the acknowledged one. Below, we show how FLM’s core abstractions help achieve this goal.

**Events.** FLM programs are written in terms of events. For our task, we can define these two events:

```plaintext
event DHCP_Ack(int cip, int cmac);
event IP_Pkt(int sip, int smac);
```

Events are detected by parsing packets that arrive at the switch. From the DHCP_Ack packet, the system parses the DHCP message payload and extracts the client’s MAC and new IP. Other packets are parsed as generic IP_Pkts, from which the source IP (sip) and MAC (smac) are extracted.

**Patterns.** FLM patterns are regular expressions over events, including concatenation (.) and closure (*). To begin to tackle the problem DHCP misuse, a programmer might create the following pattern, which identifies the presence of a DHCP_Ack amongst any number of other IP packets:

```plaintext
IP_Pkt . DHCP_Ack . IP_Pkt*
```

**Recording parameters.** The pattern above would match any use of the DHCP server. It recognizes an event sequence, but not the event parameters. Including one character for every possible IP address in the regular expression alphabet would make it too large. Instead, FLM patterns allow for the binding
The above pattern characterizes an anomaly in the communications with a single client. In reality, an operator wants to monitor many clients. To track multiple clients, one specifies an array of patterns (in this case of size 2048), which will all be active simultaneously:

\[
\text{spec}<2048> \text{ dhcp_misuse} = \ldots
\]

Next, one must specify a mapping of events to patterns, so all clients with the same MAC are identified and applied to the same pattern. The operator provides an indexing function, which computes an array index from the values carried by each event. For DHCP_Ack events, the index is the hash of the client MAC. For IP_Pkt events (to catch the outgoing packets), the index is the hash of the source MAC.

\[
\text{IDX} = \{
\begin{align*}
| \text{DHCP}_\text{Ack} & \rightarrow \{\text{hash(cmac)}\} \\
| \text{IP}_\text{Pkt} & \rightarrow \{\text{hash(smac)}\}
\end{align*}
\}
\]

The expressions for each index calculation are user-defined. If the operator wishes to prevent hash collisions, they can implement algorithms such as probabilistic data structures or detect collisions by storing keys and siphoning overflows to a software controller, as in Sonata and Aragog [11,28].

Responses. Finally, the operator will want to react to detected misuse somehow—perhaps by blocking the client or logging the anomaly. FLM allows users to react however they choose by invoking an arbitrary Lucid subroutine.

Putting it all together. The left side of Figure 1 shows the combination of all the features above in an FLM program implementing the DHCP specification. The spec declaration generates an array of size 2048, where each index represents one copy of the FLM pattern. The IDX block identifies the flows, the DETECT block contains the pattern, and the block after "=>" contains the response.

3.2 Compilation Overview

We compile FLM programs to Lucid and use the Lucid compiler to generate P4. While Lucid frees us from defining event parsers in P4, we must still map our pattern-based programs to PISA stages. This presents two key challenges:

1. To match a pattern, the program must both update the register holding the values of the variables stored in the pattern (for example, storing a particular client IP in the assigned variable in the DHCP example) and update the register holding the position in the pattern. However, this requires too much memory and computation to fit into a single stage and register, as it must in order to keep the state of the pattern up-to-date.

2. It is not even clear how to implement an arbitrary pattern that does not contain variable bindings. For example, consider a state machine representing a pattern without
// Compute the character using the old variables
if (event.type == DHCP_Ack) {
  c = ACK;
}

// On ack, update the store value
if (event.type == IP_Pkt and sip != assigned) {
  assigned := event.cip;
}

if (event.type == IP_Pkt and sip == assigned) {
  c = IP1;
}

// Synthesized mapping f from character to value
f(c) {
  if (c == ACK) {
    return 1;
  }
  if (c == IP0) {
    return 0;
  }
  if (c == IP1) {
    return 1;
  }
}

// Synthesized mapping g from character to value
g(c) {
  if (c == ACK) {
    return 2;
  }
  if (c == IP0) {
    return 8;
  }
  if (c == IP1) {
    return 9;
  }
}

// Synthesized update function to do transition
update_state(curr, x, y) {
  if (curr + y < 3) {
    return x ⊕ 5;
  } else {
    return curr & 3;
  }
}

// Update the stored state using the f and g maps
state := update_state(state, f(c), g(c));

Figure 2: A DFA representation of the translated DHCP FLM pattern for its transition function, and preamble code to compute the input character and variable updates. All integers are 4 bits, and the update function uses addition overflow to model the transitions correctly.

any variable bindings. A simple implementation of its transitions would take the form of:

```
transition (state, input):
  if (state == s1 and input == i1):
    return s2;
  else if (state == s1 and input == i2):
    return s3;
...
```

This will contain more branches than are allowed in a single stage for most machines. However, the transition must fit into a single stage in order to read and write at line rate. The reason for this is that the transition depends on the current state of the machine, which is only available after reading a register. In PISA, registers can only be accessed once per pipeline pass, so the transition must occur at the same time as the read.

Solution overview. We solve these challenges by carefully compiling an FLM program in a series of steps. In a preprocessing step, we transform the FLM program to an intermediate representation. This step inserts explicit “transition” statements into each event handler that will be compiled away later, and keeps the FLM pattern definition as a global definition. In the first step, we break a pattern into a series of variable updates and predicate computations that computes an input character from a finite set determined by the event type and predicate evaluations. This solves the first challenge above, as we can move all of the variable storage and predicate computation into earlier stages. In the second step, we transform a pattern without bindings into a classical regular expression over the input characters from the first step, and synthesize an implementation of the corresponding DFA in a single stage. Condensing the transition function into a single stage solves problem 2, but is difficult as it requires searching through all possible state numberings and bit-wise ALU operations for one that satisfies all transitions. We offload this hard work to an automated SMT solver.

Preprocessing on DHCP. We translate the DHCP FLM program into an intermediate representation that allows for more control of the inputs to an underlying state machine. The core data structure is re<size>, which defines an array of state machines with size indices that match an FLM pattern. The pattern is copied from the high-level program. To interact with it, the expression transition(name, idx, ev) applies the event ev to the state machine at index idx, and evaluates to a boolean indicating whether or not the sequence of events applied to it so far matches the FLM pattern name. For the DHCP example, in the handler of each event, we compute the index using the expressions provided in the high-level program (hashing the MAC address). Then, we add transition(dhcp_misuse, idx, this), where this represents the event for the current handler. If that returns true, we run the user-defined response code. The right side of Figure 1 shows the intermediate response code. The right side of Figure 1 shows the intermediate representation with explicit transition statements in event handlers.

Step 1 for DHCP. The next two steps compile the remaining FLM pattern and transition statements from the inter-
mediate representation into simple assignments and register operations. We will refer to the pseudocode in Figure 2, which represents the implementation of one transition statement. First, we separate out the variable bindings. For the DHCP example, it is enough to store the parameter of the first DHCP_Ack event in the variable assigned. Furthermore, we wish to compute an input character c from a finite alphabet by evaluating the predicates in the FLM pattern. This alphabet is composed of all of the event types of the original, followed by bit strings representing the values of the predicates. Because IP_Pkt appears with a predicate, it is expanded to the letters IP_Pkt1 and IP_Pkt0. DHCP_Ack does not appear with one, so it stays as is. These translations are shown in Lines 1–9 of Figure 2.

Step 2 for DHCP. In this step, we will translate the FLM pattern into a classical regex over the alphabet described above, and then implement its transition in a single stage. To translate the pattern, events that appear with a predicate become unions of any event with the same type where that predicate is true; similarly, events without a predicate are unions of any event of their respective type. Other constructs such as concatenation and closure remain as they are. For this example, the translated alphabet is the set \{DHCP_Ack, IP_Pkt0, IP_Pkt1\} and the classical regex is:

\[(\text{IP}_\text{Pkt}0 + \text{IP}_\text{Pkt}1)^* \text{DHCP}_\text{Ack} \text{ IP}_\text{Pkt}1^* \text{IP}_\text{Pkt}0\]

Next, we translate this classical regex into a DFA and synthesize its implementation in a single pipeline stage. This is required as a naive implementation of the transitions would not fit in the limited computation available in one stage. Instead, we search through all of the state numberings and bit-wise operations to find ALU operations that complete all transitions correctly. On the right side of Figure 2, we show a picture of the DFA representing the above regex. To implement it, we take advantage of the fact that a single register read-modify-write action can take up to two arguments computed in prior stages. Given a DFA, we search for the following:

- A mapping from states to integers, as shown by numbers preceding each state in Figure 2 (e.g. Start is numbered 0).
- Two mappings \(f\) and \(g\) from the alphabet to integers that will be used as inputs to the read-modify-write instruction. These are shown on lines 11 and 15 of Figure 2. Because they can be computed in earlier stages, we can use lookup tables to implement them, which are not available when updating the DFA state.
- A read-modify-write instruction that implements the transition function of the DFA using operations available on the switch and results from the \(f\) and \(g\) mappings. This is shown on line 19 of Figure 2.

The code in Figure 2 is laid out on the switch to compute \(c\), \(f\), and \(g\). In Figure 1, the re definition is replaced with a register definition, and each transition statement is replaced with the code in Figure 2: it first reads and updates the variables at the current index, then computes the input character and its corresponding mapping values, and finally applies the transition function to the state register with those values. It outputs whether the result represents an accepting state in the DFA (in this example, result was 5 for "Acc").

4 FLM Language Definitions

In this section, we describe the FLM language, provide its regular-expression-like syntax, and define the language’s semantics over packet traces. In section 6, we prove that our compiler translations are correct: the low-level switch program correctly implements the high-level pattern semantics.

4.1 FLM Language

The FLM language is a wrapper to provide access to the expressive FLM patterns. An FLM program consists of:

1. A name and size, written spec\(<\> myname = ... where \(i\) is the number of replicated state machines.
2. An IDX \{...\} block which determines which index to use for each event.
3. Optionally, a DATA \{...\} block that declares one or more registers to be used for a stateful response to a sequence match (for example, counting matches).
4. A DETECT\{pat\} = \{response\} block, where \(pat\) is an FLM pattern and \(response\) is Lucid code indicating what to do when the pattern is recognized.

Finally, to recognize properties such as liveliness or timeouts, such as detecting half-open TCP queries [21], we provide a special event called maintenance. This event is guaranteed to eventually visit every state machine in a spec, so it acts as a final event to match a pattern that might never observe any more packets arriving. More about maintenance events is included in the Appendix.

4.2 Syntax and Semantics of Patterns

In each FLM program, there is a finite set \(A\) of event types, such as DHCP_Ack and IP_Pkt. An event is a pair of an event type \(a \in A\) and an integer \(z\), written \(a(z)\). While this only includes events with a single parameter, it generalizes easily to any number of parameters. The top of Figure 3 shows the syntax of an FLM pattern. We allow predicates over the parameters of events \(a(p)\) to denote events of type \(a\) whose parameter satisfies \(p\). We also allow binding parameters in events \(a(@y;p)\) for use in predicates.

Patterns (and their contained predicates) are evaluated under an environment. An environment \(E\) is a mapping from variables to integers, with its domain denoted by \(\text{Dom}(E)\).
we use lambda notation to define predicates. For example, the empty environment is denoted by "\( \emptyset \)" and the empty string by "\( \varepsilon \)". The union and intersection of the sets are classical regular expressions; for example, "\( a(\@y;p).r_1 \)" is a binding event, and "\( r_1 \cdot r_2 \)" represents concatenation, "\( r_1 \)" represents closure, "\( r_1 + r_2 \)" represents union, and "\( r_1 \& r_2 \)" represents intersection.

### Denotational Semantics

Denote the set of strings with "\( \mathcal{S} \)." The empty set is denoted by "\( \emptyset \)." The empty string is denoted by "\( \varepsilon \)." The set of strings that are contained in a pattern and their derivatives is computed using the standard semantics of lambda calculus. For example, if the given integer is at least 10, we use the standard semantics of lambda functions. Finally, an FLM pattern represents an environment that may contain one or more free variables.

In this section (except for the HCIF pattern, for continuity), we use lambda notation to define predicates. For example, \( \lambda x.(x \geq 10) \) is a predicate that returns true if the given integer is at least 10. We use the standard semantics of lambda functions. Finally, an FLM pattern is closed under an environment if all of its free variables appear in \( E \).

On the bottom of Figure 3, we show the semantics of an FLM pattern. Each FLM pattern defines a set of strings of events that belong to its language. A binding has a scope for its variables. Predicates within the scope can use the variable. For example, \( \{a(\@y;\lambda x.true),b(\lambda x.(x == y))\} \) is the set of any event of type \( a \) (the predicate is always true) followed by one of type \( b \) with the same parameter (e.g. \( a(\@y).\lambda x.(x == y) \)). Constructors of FLM patterns are defined similarly to those of classical regular expressions; \( [r + s]_E \) and \( [r \& s]_E \) represent union and intersection of the sets \( [r]_E \) and \( [s]_E \), respectively, and \( [r^*]_E \) represents zero or more copies of \( [r]_E \).

<table>
<thead>
<tr>
<th>FLM pattern</th>
<th>( R )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( p )</td>
<td>predicate</td>
</tr>
</tbody>
</table>

\[ r ::= \emptyset \quad \varepsilon \quad a(p) \quad a(\@y;p).r_1 \quad r_1 \cdot r_2 \quad (r_1)^+ \quad r_1 + r_2 \quad r_1 \& r_2 \]

**Figure 3:** Technical cheat sheet. Definitions for FLM patterns and their derivatives.

### 4.3 FLM Intermediate Representation

The FLM intermediate representation simplifies the higher-level language features to leave just the patterns. It includes two new features not present in Lucid:

1. \( \text{re<}@y@> \) pat is a statement that defines an array of \( i \) finite state machines named \( \text{myname} \), each recognizing the FLM pattern \( \text{pat} \).

2. \( \text{transition} \langle \text{myname, idx, ev} \rangle \) is an expression that applies a transition with the event \( \text{ev} \) to the state machine at index \( \text{idx} \) of \( \text{myname} \). It evaluates to \( \text{true} \) if the state machine is in an accepting state (the pattern has been recognized), and \( \text{false} \) otherwise.

An FLM program is transformed into the definition of a state machine with the same pattern, size, and name. Then, at the beginning of each event handler, the compiler adds the following code:

\[
\text{if } \langle \text{transition}(\text{myname, idx, this}) \rangle \{ \text{response}; \}
\]

\( \text{myname} \) is the name of the state machine, \( \text{this} \) is the event transitioned with (the current event for the handler), \( \text{idx} \) is computed using the \( \text{IDX} \) block, and \( \text{response} \) is the user-defined response.

### 5 From FLM patterns to Regular Expressions

We showed in section 3 how to build a DFA and some preambles for the DHCP example. Here, we describe more generally how to translate an FLM pattern into a regular expression, which we translate to a DFA in section 7. Due to hardware restrictions, we cannot complete all of the actions necessary to store variables, evaluate predicates, and transition pattern state machines in one stage, so our plan is to carefully separate those operations into a series of stages. First, we lift variable bindings out of patterns, then remove predicates to reduce the problem to implementing a finite state machine over a finite alphabet where events are paired with bits representing the predicates in a pattern.

#### 5.1 Lifting out variable bindings

A binding FLM pattern has the form \( b(\@y;p).r \). These may occur deep within a pattern, posing a problem for implementing the variable bindings in a pipeline stage before the pattern state. In order to place bindings in an earlier stage, binding occurrences must depend only on the incoming event and environment, not the state of the pattern. We move the bindings to the top-level of a pattern while preserving its semantics by introducing a new form of patterns:

\[ b(@y) \triangleright r \]
Intuitively, this construct binds the first occurrence of an event with type \( b \)'s value to the variable \( y \), and then proceeds with matching \( r \). The key aspect of the \( \triangleright \) syntax is that it separates the binding out from the rest of the pattern. Our goal is to move these bindings all the way to the top-level using **rewrite rules**, to get a pattern that is written as a series of bindings, followed by a pattern without any variable changes at all. We show how this works on the DHCP example:

\[
\begin{align*}
\text{IP}_\text{Pkt}^* \\
\text{DHCP}_\text{Ack}(\#\text{int assigned} = \text{cip}) \triangleright \\
\text{IP}_\text{Pkt}(\text{sip} == \text{assigned})^* \\
\text{IP}_\text{Pkt}(\text{sip} != \text{assigned})
\end{align*}
\]

The first rule converts an "@" binding into a "\( \triangleright \)" one in place.

\[
\begin{align*}
\text{IP}_\text{Pkt}^* \\
\text{DHCP}_\text{Ack}(\#\text{int assigned} = \text{cip}) \triangleright \\
\text{DHCP}_\text{Ack} \\
\text{IP}_\text{Pkt}(\text{sip} == \text{assigned})^* \\
\text{IP}_\text{Pkt}(\text{sip} != \text{assigned})
\end{align*}
\]

The second rule moves the binding up by one level.

\[
\begin{align*}
\text{DHCP}_\text{Ack}(\#\text{int assigned} = \text{cip}) \triangleright \\
\text{IP}_\text{Pkt}^* \\
\text{DHCP}_\text{Ack} \\
\text{IP}_\text{Pkt}(\text{sip} == \text{assigned})^* \\
\text{IP}_\text{Pkt}(\text{sip} != \text{assigned})
\end{align*}
\]

This pattern has the same semantics as the original, but is in **prefix form**: a binding followed by a binding-free pattern.

**Definition 1.** An FLM pattern \( s \) is binding-free if it contains neither \( \triangleright \) nor \( @ \).

In the last DHCP example above, the binding-free pattern is the portion after the \( \triangleright \).

**Definition 2.** An FLM pattern \( r \) is in prefix form if it is written as \( B \triangleright s \), where \( B \) is a series of bindings \( (b_1(\#y_1) \triangleright b_2(\#y_2) \ldots) \), and \( s \) is binding-free.

The last version of the DHCP pattern above is in prefix form. FLM patterns in prefix form cannot contain \( @ \) bindings, as they are all converted to the \( \triangleright \) syntax. The semantics of the \( \triangleright \) operator is the union of two sets. The first covers cases when the binding is required. In this case, the binding \( b(\#y) \triangleright r \) should bind the value of the first occurrence of event \( b \) to the variable \( y \). The second covers cases when the variable \( y \) is not used to match the pattern. For example, the pattern:

\[
b(\#y) \triangleright (a(\lambda x.\text{true}) + (b(\lambda x.\text{true}).a(\lambda x.x == y)))
\]

is meant to define either any string with a single event of type \( a \), or a sequence of an event of type \( b \) followed by \( a \) where their parameters match. However, in the case of a single \( a \) event, the binding is not needed. For this case, we quantify over all possible values for \( y \) when defining it, which ensures the value of \( y \) does not matter for matching.

**Definition 3.**

\[
\begin{align*}
[b(\#y) \triangleright r]_E = \{ w_1 b(\#z).w_2 \in [r]_{E,y,z-c} \mid b \not\in w_1 \} \\
\cup \{ w \mid b \not\in w \text{ and } \forall z,w \in [r]_{E,y,z-c} \}
\end{align*}
\]

In section 6, we show that if the rewrite rules can transform a regular expression into a new one that is in prefix form, it can always be implemented in a pipeline. A full list of rewrite rules is contained in the Appendix. For all of these rules, we show that if \( r \) is rewritten to \( r' \), then for all environments \( E \) under which \( r \) is closed, \([r]_E = [r']_E\).

**Unimplementable patterns** The rewrite rules are not complete. Some patterns cannot be easily rewritten into prefix form to be recognized in a pipeline. For example, the following pattern fails to reach prefix form:

\[a(\lambda x.\text{true}).a(\lambda y).a(\lambda x.x == y)\]

This is meant to define a sequence of three events of type \( a \), where the parameters of the second and third events match. This is semantically well defined, but it cannot be implemented easily because the variable updates happen before accessing the pattern state. When an \( a \) event arrives, the decision to record \( y \) must be made without knowledge of previous events. Our rewrite rules would reject this pattern. This could arise in some compound monitors: for example, extending the DHCP example to bind the unassigned \( \text{sip} \) and check that it is subsequently used as part of another pattern, such as one for a generic TCP handshake. This would fail because the binding would not be with the first \text{IP}_\text{Pkt}. We included some explicit rejected patterns in the Appendix.

### 5.2 Translating events

Now, we will focus on recognizing a binding-free pattern by translating it into a DFA over a new alphabet.

**Alphabet.** The alphabet is formed by combining all of the event types of the pattern with all possible combinations of values of the predicates. As shown in subsection 3.2, the alphabet for the DHCP example is \{\text{DHCP}_\text{Ack}, \text{IP}_\text{Pkt0}, \text{IP}_\text{Pkt1}\}. \text{IP}_\text{Pkt} appears with a bit representing the value of the associated predicate, and \text{DHCP}_\text{Ack} remains as it is because it does not appear with a predicate.

In general, the alphabet for the translation of an FLM pattern \( r \) is defined as follows, where \( \text{bin}(n) \) is the binary representation of a natural \( n \), \( P(a) \) is the list of all of the predicates in \( r \) for event type \( a \), and \( A \) is all of the event types:

\[
\{ a.\text{bin}(n) \mid a \in A \land n < 2^{\text{len}(P(a))} \}
\]

**Events.** We need to take a single concrete event \( a(\#z) \) and output a character in the new alphabet. To do this, we define the letter translation \( T_l \), which keeps the event type and appends each predicate’s evaluation under the given environment. In our example, consider an \text{IP}_\text{Pkt} event where \text{sip} = 10.0.0.1. If the environment contains the mapping from \text{assigned} to 10.0.0.1, then the translated letter is \text{IP}_\text{Pkt1}. Otherwise, it is \text{IP}_\text{Pkt0}. 


5.3 Eliminating predicates from patterns

Now, we translate a binding-free pattern into a classic regular expression over a finite alphabet by eliminating all remaining predicates. The translation, called $T_e$, maps events with predicates to unions of characters with the same event and the corresponding predicate being true. The formal definition is in the Appendix. For example, IP_Pkt(sip == assigned) is translated to the pattern IP_Pkt1. Because IP_Pkt alone specifies no predicates, it is translated to the pattern (IP_Pkt0 + IP_Pkt1). Union, intersection, concatenation, and closure all just apply the translation recursively. The full translation of the DHCP pattern is:

\[
(IP_{Pkt0} + IP_{Pkt1})^* \cdot DHCP_{\text{Ack}} \cdot IP_{Pkt1}^* \cdot IP_{Pkt0}
\]

6 Translation Correctness

In this section, we present the theorem of correctness for our translations. Intuitively, this means that a translated string of events is accepted by a translated regular expression exactly when the original string of events is in the language of the original pattern, assuming that the variable updates are performed correctly. We first introduce the concept of derivatives, which formalize what should happen when a single event is processed. Then, we state our main theorem, which relates a series of derivatives to processing using our translations.

6.1 FLM Pattern Derivatives

Derivatives of FLM patterns formalize what happens when one event arrives. We will define one for binding-free FLM patterns and one for lists of bindings. A derivative of a pattern, $D_w$, is taken with an event and an environment. It outputs a new pattern, which represents the remaining pattern to be matched. We illustrate this by example with the DHCP pattern matching a string of events on the left of Figure 4. The last pattern is $\varepsilon$, which means that the original DHCP pattern accepts the string of events. The full derivative rules used are shown on the right of Figure 4. This also shows the binding derivative, which takes a series of bindings, an event, and an environment, and outputs a new binding and environment with the correct variable updates.

Formally, we show that the outputted pattern of a derivative contains all the strings that would form word in the language of the input pattern when concatenated to the input event:

**Theorem 1.** For all $a,z,s,E$ where $s$ is binding-free and closed under $E$:

\[
[D_w(a(z);s;E)]_E = \{ w | a(z), w \in \llbracket s \rrbracket_E \}
\]

6.2 Correctness Theorem

Our translations are correct if, after we translate an FLM pattern into a DFA with $T_e$, and feed it a string of events translated with $T_i$, the DFA accepts only when the original string of events is in the language of the original pattern.

We show that the relation between a pattern $s$ and its translated DFA via $T_e$ is preserved when transitioning the DFA using characters translated with $T_i$. In particular, taking the FLM pattern derivative of $s$ with an event and then translating it to a DFA is the same as transitioning $T_e(s)$ with a translated event. At the end of a string of events, we test whether the translated DFA is accepting, which is equivalent to string being in the language of the original pattern.

To state this formally, we define the translation of a word (a string of events), which repeatedly applies $T_i$ and $D_{\text{bind}}$ to transform the word into a string of finite-alphabet characters, given a list of bindings, an initial environment, and a list of predicates. The translated example word from Figure 4 would be $\text{DHCP}_{\text{Ack}} \cdot IP_{Pkt1} \cdot IP_{Pkt0}$.

**Definition 5.** The word translation

\[
T_w(a_1(z_1) \ldots a_n(z_n), B, E, P) = T_i(a_1(z_1), E, P) \cdot T_w(a_2(z_2) \ldots a_n(z_n), B', E'), P)
\]

Where $B', E' = D_{\text{bind}}(a(z); B; E)$

The word translation of $\varepsilon$ is $\varepsilon$. The rewrite rules described in subsection 5.1 preserve the semantics of patterns. They also preserve a property we call implementability, which means that the derivative of a pattern with events that are not binding is semantically equivalent no matter the assignments to unbound variables. This property holds if input patterns bind variables using the first occurrence of an event type, and always use variables after they are bound. We show an unimplementable pattern in section 5. For the technical definition, see the Appendix. Finally, we have our correctness theorem, which states we can check whether a translated word is in the language of a translated regular expression to determine if a word matches a pattern. $\text{Asgn}(B)$ simply assigns 0 to each variable of $B$, so that there are never undefined variables. The proof is by induction on the length of a word and is contained in the Appendix.

**Theorem 2.** For any word $w$, binding list $B$, pattern $s$, environment $E$, and predicates $P$, if $B \triangleright s$ is in prefix form, closed under $E$, and implementable, then:

\[
T_w(w, B, (E, \text{Asgn}(B)), P) \in L(T_e(s, P)) \iff w \in \llbracket B \triangleright s \rrbracket_E
\]
Let \( \{=, \neq, <, >\} \) denote membership in the set of characters. We show how to synthesize code to perform the classical regex matching algorithm. Acceptance is indicated by the empty string, \( \varepsilon \).

The previous section reduced the problem of matching FLM patterns to matching specially constructed regular expressions, but it is still not clear how to do this at line rate. In this section, we show how to synthesize code to perform the classical regex matching algorithm. Acceptance is indicated by the empty string, \( \varepsilon \).

Figure 4: The progress of matching a string with the DHCP example. The left column contains the binding-free patterns, and the right tracks the environment. The arrows indicate the pattern derivative with the incoming event and current environment. Acceptance is indicated by the empty string, \( \varepsilon \).

Figure 5: A syntax template for a single register action to be synthesized. Blue-bubbled brackets represent choices between the expressions in the brackets. Red-bubbled brackets represent choices between the operators in the brackets. Each \( c_i \) represents a constant chosen by the synthesizer.

7 DFA Synthesis

The previous section reduced the problem of matching FLM patterns to matching specially constructed regular expressions, but it is still not clear how to do this at line rate. In this section, we show how to synthesize code to perform the classical regex matching algorithm in order to use theorem 2. We use SMT-based synthesis to fit a transition function into at most four register actions, the maximum allowed on the Tofino.

7.1 Synthesis goal

To implement a DFA’s transition function within the allowed register actions, the synthesizer will attempt to cleverly assign numbers to the DFA states and alphabet while generating a short function composed of a fixed number of simple instructions like bitwise operations, arithmetic operations, and conditional branches. The function will calculate the next state given the current state and input event without the need to enumerate DFA transitions.

We take as input a DFA and a bitvector length \( l \). A DFA is a five-tuple \( (Q, \Sigma, \delta, q_0, F) \), where: \( Q = \{q_0, q_1, \ldots\} \) is a finite set of states with initial state \( q_0 \), \( \Sigma = \{\sigma_0, \sigma_1, \ldots\} \) is a finite set of alphabet symbols, \( \delta : Q \times \Sigma \rightarrow Q \) is the transition function, and \( F \subseteq Q \) is the set of accepting states. For the DHCP example from subsection 3.1, the DFA has \( Q = \{\text{Start}, \text{Acked}, \text{Acc}, \text{Rej}\} \), \( \Sigma = \{\text{Ack}, \text{IP0}, \text{IP1}\} \), \( F = \{\text{Acc}\} \), and \( \delta \) as shown in Figure 2.

We output a function to implement the state machine’s transition on the Tofino or any other hardware whose memory update is at least as expressive. Specifically, we output:

- A mapping \( R \) from \( Q \) to \( \{0, \ldots, 2^l - 1\} \) that uniquely numbers the states, and by convention we fix \( R(q_0) = 0 \).
- Two mappings \( f \) and \( g \) from \( \Sigma \) to \( \{0, \ldots, 2^l - 1\} \). These are passed as arguments to the register actions.
We identified 15 such tasks from prior work, and implemented the FLM compiler atop the Lucid framework \[19,25\] using approximately 1500 lines of OCaml available on GitHub\(^1\). We implemented DFA synthesis code in OCaml using the z3 SMT solver \[30\]. It first transforms each FLM pattern into prefix form and translates it to a classical regular expression. Then, it converts the pattern into a DFA and runs syntax-guided synthesis to generate the corresponding mappings and memops, expressed as an intermediate representation Lucid program. This program is subsequently compiled using the existing Lucid framework’s backend and vendor-provided P4 compiler (bf-p4c)\(^2\) to generate the final data plane program binary. We use ocamlc 4.14.0, z3 4.11.2, and bf-p4c 9.13.0.

### 8 Implementation

We implement the FLM compiler atop the Lucid framework \[19,25\] using approximately 1500 lines of OCaml available on GitHub\(^1\). We implemented DFA synthesis code in the compiler in OCaml using the z3 SMT solver \[30\]. It first transforms each FLM pattern into prefix form and translates it to a classical regular expression. Then, it converts the pattern into a DFA and runs syntax-guided synthesis to generate the corresponding mappings and memops, expressed as an intermediate representation Lucid program. This program is subsequently compiled using the existing Lucid framework’s backend and vendor-provided P4 compiler (bf-p4c)\(^2\) to generate the final data plane program binary. We use ocamlc 4.14.0, z3 4.11.2, and bf-p4c 9.13.0.

### 9 Evaluation

We evaluate FLM by using it to implement a diverse set of sequence monitoring tasks of interest to network operators. We identified 15 such tasks from prior work, and implemented them alongside a Lucid program that used the same events. All of the monitored patterns are listed in the Appendix.

Figure 6a shows these programs and summarizes our results, including the lines of code needed to express the examples, the synthesized DFA complexity and synthesis time, and the stages used. We are able to express each of the 15 tasks in the FLM language, pointing to its flexibility. The table shows lines of code as a proxy for ease of use. We see that FLM programs are short and all tasks are expressed in a few 10s of lines. In contrast, when translated to Lucid these programs are 5-10x bigger, which is a proxy for implementation effort of expressing these tasks directly in P4.

Our compiler is able to compile each of the programs to the Intel Tofino, which demonstrate the line-rate monitoring capabilities of FLM.

#### 9.1 Compilation time

Figure 6b shows the compilation time for each program on an AWS EC2 t3.medium server with 4GB memory and 2 vCPU (unlimited burst). We break the total compilation time into the frontend (Lucid compiling) and backend (P4 compiling). Note that the Lucid time includes the synthesis time from Figure 6a. We see that most programs need little time (seconds) to complete our compilation steps. Although the DFA synthesis step depends on the complexity of the pattern and is theoretically NP-Hard, all tasks finish in a few minutes.

Programs decorated with FLM have slower compilation time for both the Lucid compiler backend and the vendor P4 compiler. This is caused by the additional statements added to the IR and the resulting overhead for optimizing the pipeline layout, and mostly depends on the complexity of the original program. The complexity of the pattern (aside from additional variables) does not affect the backend’s and vendor compiler’s compiling time. All implementations take roughly the same time to compile once synthesized.

#### 9.2 Hardware Resource Utilization

Figure 6c shows the hardware resource usage of the Tofino binaries for each program. The three most relevant metrics for FLM are the number of pipeline stages used (Figure 6a), the percentage of metadata fields (PHV) allocated, and the percentage of instruction words (VLIW) allocated.

Depending on the complexity of pre-processing involved in translating events (packets) to letters in the pattern, FLM compiles programs into 6-10 stages, all comfortably fitting within the Intel Tofino v1 (12 stages). The DFA itself always uses a single stage, regardless of the complexity of the pattern. The additional stages take care of the Lucid event handling and control flow, as well as predicate computation. Because there are data dependencies between stages arising from control flow, these stages are not always “full;” additional unrelated programs can fit alongside them without using more stages, as the preprocessing steps are parallelized with existing logic.

Meanwhile, FLM programs use reasonable resources: adding FLM to an existing Lucid program only consumes 1-15% additional PHV and VLIW, much of which is for setup (parsing and pre-processing). Interestingly, the resource usage for some tasks goes down with added FLM monitoring due to the additional stage usage.

---

1. https://github.com/PrincetonUniversity/lucid/tree/SpecRegex
2. bf-p4c: Backward Flow P4 Compiler
(a) The results from compilation of our 15 example monitors. The gray
rows show the base programs. The following white rows show one
or more monitors that we added to the base programs. For the base
programs, we show the lines of code of the source program (Lucid),
the intermediate representation before compiling to P4 (IR), and the
resulting P4 program (P4), as well as the number of stages used. For
the monitoring tasks, we show the additional lines of code and stages
added by FLM as well as the size of the DFA in terms of states (Q)
and the alphabet (Σ) as well as the time required to compile FLM to Lucid.

9.3 DFA Synthesis

We measured the size of each DFA in our examples, noted in the "DFA"
column of Figure 6a as a pair \((Q, |Σ|)\) to show the number of states and the size of the alphabet. These varied from small automata with only a few states and symbols to ones with dozens of states and alphabet characters. The step of synthesizing an implementation, included in the "FLM Time", was relatively quick for all of them. In prior work [6], we further evaluated the difficulty of state machine synthesis.

We found that it was possible to generate implementations for state machines representing simple networking tasks such as tracking a TCP handshake. We also tested combinations of state machines via union (tracking multiple at once) or concatenation (tracking one after another) via video fingerprinting examples. While union was usually easier than concatenation, there were many examples of the same size DFA (as measured by states and alphabet) where some worked and some did not. There are other qualities of DFA complexity that show up only in some fingerprinting patterns. For example, states with many incoming or outgoing edges place many constraints on the same variables and make a solution more difficult to find.

Finally, we tried five different synthesis templates of varying complexity, ranging from simple assignment to using Tofino-specific tricks such as representing states with two integers in a paired array and updating both simultaneously. We also varied the choices of bitwise operators available in each template. We found that the template shown in Figure 5 was the best balance between expressiveness (ability to represent a wide variety of DFAs) and synthesis time.

10 Related Work

The syntax for FLM programs is inspired by Aragog [28], a system that focused on recognizing issues in distributed systems by specifying regex-like patterns. The patterns were checked by a global verifier that had specific events forwarded to it by all the systems. Aragog operates entirely in the control plane, whereas our work focuses on recognizing packet sequences appearing on a single data plane switch at line rate.

Many works use data plane switches to recognize regular expressions appearing in packets for the purposes of content inspection. Some early work appeared before P4 was released [20, 22]. More recent work has built on further hardware advances, and includes frameworks specialized for matching strings in a packet [14, 27]. DBVal [17] focuses on verifying the data plane execution of a single packet. DeepMatch [12] focuses on searching for regular expressions within the payload of packets, and developed some techniques to hold state between payloads for a single flow. Our work differs in that it focuses on patterns of sequences of packets, where all the computation happens in a single stage used once
per packet, rather than a series of stages that can search for patterns within packet content. We also allow more expressive patterns with the binding of event values and predicates.

Our patterns draw inspiration from previous work on parametric verification [3], as well as studies of various forms of automata for wide-ranging applications. Timed automata [2] can record the timing of events and place time constraints on transitions. Symbolic automata [8, 9] permit a very wide variety of predicates on transitions without memory other than their state, while register automata [16] can record characters in registers, and check only equality. Recently, symbolic register automata [7] were proposed, which combines the two.

Our preliminary paper [6] presented an algorithm for synthesizing DFA transitions using SMT solvers. FLM extends that core with a new pattern-based language and compiler that separates bindings and predicates from classical regular expressions which are turned into DFAs.

11 Discussion

Limitations. One limitation for recognizing FLM patterns, and the reason for the rewrite rules, is the difficulty of correctly updating variables under the constraints on PISA switches. Other hardware or computation models might provide an easier path to computing the environment derivative (on a general purpose CPU with unlimited memory, it could just be computed directly), which would increase the number of FLM patterns that are recognizable. Still, we show in our evaluations that the subset of implementable patterns is expressive enough for many applications. An interesting research question for the future is to characterize which FLM patterns can be implemented under which computation models, and how best to do so while minimizing resource use.

Another limitation comes from the amount of computation available in a single stage, which dictates the size of implementable DFAs. Very long or complicated patterns could translate to DFAs that are too large to fit into one read-modify-write action. For more details, see the paragraph on synthesis hardness above. Our work is only platform-specific in that the template defined in Figure 5 is tailored to compilation on the Intel Tofino. Other hardware that has different computation available in a single stage would likely permit a more or less expressive template. A solution to this that applies for some DFAs is to carefully use more than one stage to implement it. For preliminary details about this approach, see the Appendix. As hardware improves, we hope to see both more computation available within a stage and more stages, alleviating this restriction from two angles.

Scalability and flexibility. In our examples, we used a variety of indexing functions to represent individual state machines, including per-flow, per-port, and per-MAC. However, in some networks, there is not enough memory available on current hardware to store this many values, risking index collisions. Other works [21] have dealt with this problem using complex data structures, sampling, or grouping flows to be considered together. While FLM does not employ any of these by default, it is flexible enough that a programmer could implement any of these techniques to compute an index before applying transitions to the FLM state machine array. Furthermore, if the high-level language is too restricting, a programmer can write code directly in the FLM intermediate representation, allowing them to intersperse arbitrary code for how the index is computed, which event is used to transition the state machine, and where in the control flow to transition.

Monitoring scope. We built our FLM compiler to target the Intel Tofino, but the core ideas are not reliant on any particular piece of hardware. The main requirement for an FLM program is a single pipeline with atomic updates and persistent memory that is updated per-packet. This applies to any switch implementing the PISA architecture.

We did not solve the problem of recognizing patterns in a distributed system of switches, instead focusing on how to properly compile to a single pipeline. A switch with multiple pipelines would implement multiple FLM monitors independently. For most monitoring tasks, properly configuring how ports map to pipelines (essentially slicing the index space across monitors) should preserve the monitor’s reliability by sending packets intended for the same state machine through the same pipeline. We consider distributed monitoring an interesting task for future research.

12 Conclusion

We introduce FLM, a programming language that uses new abstractions to recognize and react to user-defined packet and event sequences at a switch. We develop a compilation procedure that transforms FLM programs into a series of match-action tables and register update functions, using a combination of rewrite rules and SMT-based program synthesis. Our evaluation using 15 sequence monitoring finds that FLM is flexible and supports line-rate processing on current networking hardware. This work raises no ethical concerns.

Acknowledgments

We would like to thank our shepherd, Macro Chiesa, and all of the anonymous reviewers for their insightful comments. This material is based upon work supported by the National Science Foundation Grants CNS-2007073, FMiTF-2219862, and FMiTF-2219863. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation.
References


13 Appendix

13.1 Translating patterns

The full definition of translating patterns uses $BigOr$, which translates an individual event and predicate.

**Definition 6.** $BigOr(b(p), P) = \bigvee\{\{b \circ \text{bin}(i) \mid i < 2^{\text{len}(P)} \text{ and the bit representing } p \text{ is } 1\}\}$

$BigOr(b(p_1), [p_1, p_2]) = (b10 + b11)$, as above. Similarly, $BigOr(b(p_2), [p_1, p_2]) = (b01 + b11)$. The other pattern translations are defined recursively. All of the constructors ($+, \&$, $*$) stay the same, as they have similar semantics for classical regular expressions and patterns, and we translate all of the event patterns with $BigOr$.

**Definition 7.** The FLM pattern translation $T_{re}(r, P)$:

\[
\begin{align*}
T_{re}(\varepsilon, P) &= \varepsilon \\
T_{re}(\emptyset, P) &= \emptyset \\
T_{re}(a(P_1), P) &= BigOr(a(p_1), P) \\
T_{re}(s_1, s_2) &= T_{re}(s_1, P) \cdot T_{re}(s_2, P) \\
T_{re}(s_1 + s_2) &= T_{re}(s_1, P) + T_{re}(s_2, P) \\
T_{re}(s_1 \& s_2) &= T_{re}(s_1, P) \& T_{re}(s_2, P) \\
T_{re}(s^r) &= (T_{re}(s, P))^r
\end{align*}
\]

13.2 Implementability

The following is a list of the rewrite rules we use:

- $b(@y; p).r \xrightarrow{rw} b(@y) \triangleright b(p).r$. This rule always applies, and is how we introduce the $\triangleright$ operator. It splits a binding into two parts, and removes it from the expression.

- $r.(b(@y) \triangleright s) \xrightarrow{rw} b(@y) \triangleright (r,s)$. This rule only applies if $b$ and $y$ do not appear in $r$. This ensures that no word in $r$ contains an event $b$, mirroring the semantics of $\triangleright$.

- $(b(@y_1) \triangleright s_1) + (b(@y_2) \triangleright s_2) \xrightarrow{rw} b(@y_1) \triangleright (s_1 + [y_1/y_2]s_2)$. Here $[y_1/y_2]s_2$ denotes the capture-avoiding substitution of $y_1$ for $y_2$ in $s_2$. This rule only applies when $s_2$ contains no references to $y_1$. An equivalent rule applies for $\&$.

- $(a(@y_1) \triangleright s_1) + (b(@y_2) \triangleright s_2) \xrightarrow{rw} a(@y_1) \triangleright b(@y_2) \triangleright (s_1 + s_2)$. This rule applies if $s_1$ contains no occurrences of $y_2, s_2$ contains no occurrences of $y_1$, and $a \neq b$. An equivalent rule applies for $\&$.

- $(b(@y) \triangleright s_1) + s_2 \xrightarrow{rw} b(@y) \triangleright (s_1 + s_2)$. This rule applies if $s_2$ contains no occurrences of $y$. An equivalent rule applies for $\&$.

These contain side conditions that ensure rewritten expressions have the same semantics after rewriting. For example, we check that variables are not contained in out-of-scope expressions before hoisting bindings to an outer scope. However, they also preserve a key property of FLM patterns that can be written without the new binding form that uses $\triangleright$. We call this property implementability. First, we define a shorthand for an environment defined over the variables in a list of bindings.

**Definition 8.** An environment $G$ is compatible with a binding list $B$ if $\text{Dom}(G) = \text{Range}(B)$. That is, $G$ contains one assignment for each variable in $B$. We will use the metavariable $G$ for compatible environments to differentiate from general environments denoted by $E$.

For example, the environment $(y_1 \leftarrow 1, y_2 \leftarrow 15)$ is compatible with $b_1(@y_1) \triangleright b_2(@y_2)$. We will write $\text{Asgn0}(B)$ to mean the environment that assigns every variable in $B$ to 0. Next, we define implementability, which intuitively means that the pattern derivatives are not affected by assignments to variables in a binding list for event types that do not appear in the bindings.

**Definition 9.** An FLM pattern in prefix form $B \triangleright s$ is implementable if and only if for any event type $a \notin B$, any integer $z$, any environment $E$ where $B \triangleright s$ is closed under $E$, and any two environments $G_1$ and $G_2$ which are compatible with $B$:

\[
[D_{re}(a(z); s; E, G_1)]_{E, G_2} = [D_{re}(a(z); s; E, G_2)]_{E, G_2}
\]
All of the patterns we introduce in the main paper are implementable. An example pattern which does not have this property is:

\[ b(\langle y \rangle) \triangleright a(\lambda x. (x == y)). b(\lambda x. \text{true}) \]

This pattern is semantically well-defined: it is the set of events of type \( b \) followed by \( b \) with equal parameters. However, the value of \( y \) is not known when the event \( a \) appears, and so it cannot be implemented at line-rate (in general, this type of pattern would require some look-back capability). This fails the implementability property because the assignment to \( y \) will change the result of \( D_n \) for events of type \( a \). We show that our rewrite rules preserve this property: if \( r \xrightarrow{\text{rewrite}} r' \) and \( r \) is implementable, then so is \( r' \).

Now, we define a theorem that captures the semantic meaning of both the binding and pattern derivatives. Intuitively, it is similar to theorem 1, but uses an environment produced by \( D_{\text{bind}} \) instead of a constant one.

**Theorem 3.** \( \forall B, s, a, z, E : \text{if } B \triangleright s \text{ is in prefix form, closed under } E, \text{ and implementable, then:} \)

\[
[D_{\text{re}}(a(z);s;E',\text{Asgn}(B'))]_{E',\text{Asgn}(B')} = \{ w|a(z).w \in [B \triangleright s]_E \}
\]

Where \( B', E' = D_{\text{bind}}(a(z);B;E) \)

As an example, consider the one from section 5, where \( p_1 = \lambda x. x \geq 10 \) and \( p_2 = \lambda x. x == y \):

\[ b(\langle y \rangle) \triangleright (b(p_1).b(p_2)) \]

Starting with an empty environment and an event \( b(12) \), \( D_{\text{bind}}(b(12);b(\langle y \rangle);:) = (:(y \leftarrow 12)) \). Taking the pattern derivative using the new environment:

\[ D_{\text{re}}(b(12);b(\lambda x. x \geq 10).b(p_1);y \leftarrow 12) = b(p_2) \]

The semantics of this remaining pattern when \( y \leftarrow 12 \) contains just \( b(12) \), which is correct according to the theorem: the only string starting with \( b(12) \) in \([b(p_1).b(p_2)]_{y=12}\) is \( b(12)b(12) \).

### 13.3 Example patterns

Below is a full list of the example patterns that were used for evaluation. They range from simple checks to more complicated patterns about high-level protocols.

**A1** Cuckoo Firewall [25]

```
(ip_pkt(@int saved_src=src, @int saved_dst=dst)
 .(((cuckoo_insert(fst_src==saved_src) & cuckoo_insert(fst_dst==saved_dst))*)
   .((cuckoo_insert(!((fst_src==saved_src)) || cuckoo_insert(!((fst_dst==saved_dst))))
     ||((cuckoo_insert(fst_src==saved_src) && cuckoo_insert(fst_dst==saved_dst))
     .(cuckoo_insert(fst_src==saved_src) && cuckoo_insert(fst_dst==saved_dst))
     .(cuckoo_insert(fst_src==saved_src) && cuckoo_insert(fst_dst==saved_dst))))))
```

This checks whether the cuckoo firewall insertion algorithm is working properly.

**B1** Stateful FW Timeout [25]

```
ip_pkt (@int start_time = Sys.time(); ip#tos == TOS_TRUSTED)
 .(ip_pkt(ip#tos == TOS_TRUSTED)
  || ((ip_pkt((ip#tos == TOS_TRUSTED))
    && ip_pkt(Sys.time() - start_time < 10000)))*
  .((ip_pkt((ip#tos == TOS_TRUSTED))
    && ip_pkt((Sys.time() - start_time < 10000))))
```

This is a general specification of firewall correctness. It checks that packets from inside (TOS_TRUSTED) are allowed out, and that return packets are not allowed back in past the timeout threshold.

**C1** SipHash [29]

```
/iptcp_to_server_syn
 .siphash_intermediate
 .siphash_intermediate
 .siphash_intermediate
 .siphash_intermediate
```

This checks whether or not the siphash implementation is completing the proper number of hashing rounds.

**D1 Chain replication 1 [31]**

```c
write(@int saved_seq=seq)
write(seq<saved_seq)
```

This checks whether there are write events with sequence numbers out of order.

**D2 Chain replication 2 [31]**

```c
write()
.write()
.ack()
```

This checks whether there are two write events to the same index before the first one is ACKed.

**E1 DHCP Anomaly (section 3)**

```c
IP_Pkt*
.DHCP_Ack(@int assigned = cip)
.(IP_Pkt(sip == assigned))'*
.IP_Pkt(sip != assigned)
```

This is the example from the paper.

**E2 Fingerprint ([24])**

```c
```

This represents one example video fingerprint. S1 - S8 denote different packet sizes. The fingerprint is 8 packets in sequence, with the denoted sizes.

**E3 Port Knocking (len=4) [10]**

```c
ip_in(dport==1234)
.ip_in(dport==5678)
.ip_in(dport==9012)
.ip_in(dport==3456)
```

This pattern represents an example port knocking sequence. dport is the destination port of a packet.

**E4 DNS TTL Change Count [21]**

```c
DNS_packet_fwd(@int<<32>> fst_ttl = ttl)
.(DNS_packet_fwd(fst_ttl != ttl))
```

This checks whether the ttl of packets changes when using DNS by recording the first in a flow and checking the second. The intent is to count the number of TTL changes.

**E5 DNS Tunneling [21]**
This checks that, upon receiving a DNS response, the receiver goes on to contact the requested IP. Tunneling is suspected if the receiver of a DNS response never uses the information.

E6 SwiSh Local View [32]

\[(S_1 | S_2 | S_3) * \]
\[.((S_2 . S_1) | | (S_3 . S_2) | | (S_1 . S_3))\]

This denotes any of the switches in a chain of 3 SwiShMem switches forwarding an update to the wrong neighbor.

E7 NetChain [15]

\[\text{NetChainUpdate}(\text{\texttt{v=version}})\]
\[.\text{NetChainUpdate}(\text{\texttt{version < v}})\]

This shows an algorithm running improperly, as detected by having an earlier version after a newer one.

E8 Paxos Recovery [28]

\[\text{Paxos}(\text{\texttt{\texttt{a=l1}},\text{\texttt{c=l2}},\text{\texttt{e=l3}}; \texttt{ty==RECOVER}}).\]
\[(\text{\texttt{Paxos(l1<a)}} \&\& \text{\texttt{Paxos(ty==RECOVER)}}) | | (\text{\texttt{Paxos(l2<c)}} \&\& \text{\texttt{Paxos(ty==RECOVER)}}) | | (\text{\texttt{Paxos(l3<e)}} \&\& \text{\texttt{Paxos(ty==RECOVER)}})\].
\[\text{Paxos(ty==RECOVERED)}\]

This shows the sequence of exchanges of a Paxos recovery.

E9 ATP sequence [18]

\[\text{ATP_add}(\text{\texttt{x = cnt}})\]
\[.\text{ATP_add}(\text{\texttt{cnt=x+1}})\]
\[.\text{ATP_add}(\text{\texttt{cnt=x+2}})\]
\[.\text{ATP_add}(\text{\texttt{cnt=x+3}})\]

This shows a sequence of 4 consecutive add events with an increasing count variable.

E10 ATP JobID [18]

\[\text{ATP_add}(\text{\texttt{saved_jobid=jobid}})\]
\[.\text{ATP_add}(\text{\texttt{jobid=saved_jobid}})\]
\[.\text{ATP_add}(\text{\texttt{jobid=saved_jobid}})\]
\[.\text{ATP_add}(\text{\texttt{jobid=saved_jobid}})\]
\[.\text{ATP_add}(\text{\texttt{jobid=saved_jobid}})\]
\[.\text{ATP_add}(\text{\texttt{jobid=saved_jobid}})\]
\[.\text{ATP_add}(\text{\texttt{jobid=saved_jobid}})\]
\[.\text{ATP_add}(\text{\texttt{jobid=saved_jobid}})\]

This shows a sequence of 24 consecutive add events with an increasing jobid variable.
This shows a sequence of 16 consecutive add events with the same job ID.

Unimplementable patterns Below are a few patterns that are rejected:

R1 DHCP Example Reject

\[
\text{IP}_\text{Pkt}^* \\
\text{DHCP}_\text{Ack}(\text{int } \text{assigned} = \text{cip}) \\
(\text{IP}_\text{Pkt}(\text{sip} == \text{assigned}))^* \\
\text{IP}_\text{Pkt}(\text{int } \text{unassigned} = \text{sip}; \text{sip} \neq \text{assigned}) \\
\text{IP}_\text{Pkt}(\text{sip} == \text{unassigned}) \\
\text{IP}_\text{Pkt}(\text{sip} == \text{unassigned})
\]

This pattern is rejected because the binding of the variable \text{unassigned} will not be on the first occurrence of the event \text{IP}_\text{Pkt}. A similar limitation applies to any compound, related patterns. It is meant to represent a device that uses the same wrong IP three times.

R2 ATP Sequence Version 2

\[
(\text{ATP}_\text{add}(\text{int } x = \text{cnt}; \text{cnt} == x+1))^*
\]

This pattern has two problems. First, the variable \text{x} is used before it is defined (the predicate \text{cnt} == \text{x+1} does not have the variable \text{x} in scope). Second, bindings can never occur under a closure. It is meant to loosely specify a similar pattern to E9, but of any length.

R3 Chain Replication Reject

\[
\text{write}(\text{int } \text{saved_seq}=\text{seq}) \\
\text{write}(\text{int } \text{saved_seq2}=\text{seq}; \text{seq} > \text{saved_seq}) \\
\text{write}(\text{seq}<\text{saved_seq2})
\]

This pattern is rejected for a reason similar to R1: the binding of variable \text{saved_seq2} will not occur on the first \text{write} event (even though the first \text{write} event does bind a variable). This does not apply when binding two variables with the same event (see E5).

13.4 Extensions

In this section, we go over various extensions to FLM to improve its expressiveness and usability.

13.4.1 Maintenance events

Some patterns, such as those that wait for a timeout, can be difficult to express as a sequence. Consider a sequence to check that received requests are replied to with decisions in a timely manner:

\[
\text{request}(\text{t1} = \text{Sys.time()}) \\
\text{.request}^* \\
\text{.decision}(\text{Sys.time()} - \text{t1} > \text{Threshold})
\]

Here, we are ignoring the indexing and response code to focus on the pattern. This pattern seems reasonable to detect late decisions, but what if a decision never comes? Intuitively, that should be a violation as well, but if there is never another packet for this flow, one will never be reported. The problem is that examples such as timeouts are examining a liveness property. Our solution to this is to add \text{maintenance events}, which do not represent incoming network events. Instead, they are guaranteed to visit every index of an array of FLM patterns eventually. To fix the above example, we can write the following, using \text{maintenance} for the new events.
maintenance*
  .request(@t1 = Sys.time())
  .(maintenance(Sys.time() - t1 <= Threshold) + request)∗
  .(decision(Sys.time() - t1 > Threshold) +
    (maintenance(Sys.time() - t1 > Threshold)))

The placement of maintenance events will not interfere with detecting late decisions, but it will now also detect decisions that
never arrive because the maintenance event will eventually come along and match the last disjunction. These can be implemented
easily with a packet that repeatedly circulates through the array one index at a time, perhaps with a delay to reduce overhead.

13.4.2 Longer patterns

Some sequence monitors are too complicated to be expressed using the original syntax, causing the compiler to reject them. This
could be for one of two reasons:

1. The pattern is unable to be rewritten into prefix form, usually because the programmer desires to bind variables using the
   same event type in two places. This would violate the rewrite rule for concatenation.

2. The pattern is able to be rewritten successfully, but an implementation of the transition function for the translated DFA
cannot be found. The synthesis algorithm might either time out, or return "unsat."

The solution to both of these is to pay more stages for the ability to implement a pattern. To do so, we introduce unambiguous
concatenation, which allows a pattern to be split into two parts that can be implemented sequentially.

**Unambiguous concatenation**  We say that two FLM patterns in prefix form are unambiguously concatenated with a semantic
condition that allows us to split it across stages. First, we define the **prefixes** of a pattern, which are all the prefixes of any accepted
word:

**Definition 10.** The prefixes of an FLM pattern r, denoted prefix(r), is the set:

\{ u \mid \exists E, v \text{ such that } u.v \in \llbracket r \rrbracket_E \}

Next, we define the **continuations** of a pattern, which are all the words which can be appended to an accepted word to get
another accepted word:

**Definition 11.** The Continuations of an FLM pattern r, denoted continuation(r) is the set:

\{ v \mid \exists E, u, w \text{ such that } u \in \llbracket r \rrbracket_E \text{ and } u.v.w \in \llbracket r \rrbracket_E \}

As a simple example with the pattern $a.b^*$, the prefixes and continuations can be defined with the languages of the following
patterns:

prefix($a.b^*$) = $\varepsilon + a.b^*$
continuation($a.b^*$) = $b^*$

Finally, we say that two patterns $r_1$ and $r_2$ are unambiguously concatenated, denoted $r_1!!r_2$, if the prefixes of $r_1$ and the
continuations of $r_2$ only intresect with $\varepsilon$, or more formally:

$r_1!!r_2 \iff (\text{prefix}(r_1) \cap \text{continuation}(r_2)) \setminus \{\varepsilon\} = \emptyset$

Using the above example, $a.b^*!!a.b^*$ is a valid unambiguous concatenation. $\varepsilon$ is excluded because it is always both a prefix and
continuation of any non-empty language.

**Implementation**  The unambiguous concatenation condition lets us compile a single pattern into multiple stages, which will
either allow a programmer to reuse events for variable bindings or compile a larger pattern to switch actions. In principle, there
could be many patterns connected with !!. We assume that this is at the top level, and call each concatenated pattern a **section**.
We compile a series of sections in three steps, after compiling each section individually:

1. For each DFA except the last section, add one new state, called "done." Add a self-loop for every character to "done." For
each transition from any accepting state to the rejecting state, replace it with a transition to "done."
2. For each section’s code except the first, add a clause to only run it if the previous section’s state was "done."

3. In each section’s DFA, if any later sections are not nullable, change all of its accepting states to non-accepting ones. The transition statement returns whether or not the last section run ended in an accepting state.

Step one allows us to track when each section has finished matching characters. This is the step where the unambiguous concatenation condition is important. The transitions from accepting states to non-reject states correspond to the continuations of a section, while the transitions from the initial state correspond to its prefixes. The condition guarantees that the prefixes of a later section do not coincide with the continuations of an earlier one, so we never miss a transition.

Step two ensures that we are running the patterns in sequence, not in parallel. Note that if there are new variables to be bound, they are only bound after previous sections are "done."

Step three ensures that the series of sections only accepts when the current string matches the unambiguously concatenated pattern. We leave accepting states if all later states are nullable because otherwise, we would miss some strings that only match the earlier sections.

13.5 Proofs of Theorems

Theorem 4. "Derivatives commute":
∀ a, z, s, E, P where s is binding free, both s and each predicate in P is closed under E, and preds(s) ⊆ P:

\[ D_{cl} \left( T_{l}(a(z), E, P); T_{r}(s, P) \right) = T_{r}(D_{re}(a(z); s; E), P) \]

Proof. By induction on the structure of s

Theorem 5. For all a, z, s, E where s is binding-free and s is closed under E:

\[ \llbracket D_{re}(a(z); s; E) \rrbracket_E = \{ w | a(z).w \in \llbracket s \rrbracket_E \} \]

Proof. Proof by induction on the structure of s. Base cases:

1. \( \emptyset \):

\[ \llbracket D_{re}(a(z); \emptyset; E) \rrbracket_E = \llbracket \emptyset \rrbracket_E = \emptyset = \{ w | a(z).w \in \emptyset \} \]

2. \( \varepsilon \):

\[ \llbracket D_{re}(a(z); \varepsilon; E) \rrbracket_E = \llbracket \emptyset \rrbracket_E = \emptyset = \{ w | a(z).w \in \{ \varepsilon \} \} \]

3. \( a(p) : a(z) \in \llbracket a(p) \rrbracket_E \) iff \( \llbracket p(z) \rrbracket_E \). So, \( \{ w | a(z).w \in \llbracket a(z) \rrbracket_E \} = \{ \varepsilon \} \) if \( \llbracket p(z) \rrbracket_E \) and \( \emptyset \) else, which is the derivative.

Induction cases:

1. \( r + s \):

\[ \llbracket D_{re}(a(z); r + s; E) \rrbracket_E = \llbracket D_{re}(a(z); r; E) \rrbracket_E \cup \llbracket D_{re}(a(z); s; E) \rrbracket_E \]

By induction:

\[ = \{ w | a(z).w \in \llbracket r \rrbracket_E \} \cup \{ w | a(z).w \in \llbracket s \rrbracket_E \} \]

\[ = \{ w | a(z).w \in \llbracket r \rrbracket_E \cup \llbracket s \rrbracket_E \} \]

\[ = \{ w | a(z).w \in \llbracket r + s \rrbracket_E \} \]

2. \( r & s \): Very similar to above, substituting \& and \( \cap \) for + and \( \cup \).

3. \( s^* \):

The following together show that \( \llbracket D_{re}(a(z); s^*; E) \rrbracket_E = \{ w | a(z).w \in \llbracket s^* \rrbracket_E \} \)
(a) \( \{ w|a(z).w \in [s']_E \} \subseteq [D_{re}(a(z);s';E)]_E: \)
If \( a(z).w \in [s']_E \), then \( a(z).w \in [s']_E \) for some smallest natural \( i \). \( i \) cannot be 0. Since \( a(z).w \in [s']_E = [s.s'-1]_E \), \( a(z).w = w_1.w_2 \) s.t. \( w_1 \in [s]_E \wedge w_2 \in [s'-1]_E \). \( s_1 \) cannot be \( \epsilon \), otherwise \( i \) could be smaller. So, \( w_1 = a(z).w'_1 \) and \( w = w'_1.w_2 \).

By induction,
\[
[D_{re}(a(z);s';E)]_E = \{ w|a(z).w \in [s]_E \}
\]
So, \( w'_1 \in [D_{re}(a(z);s';E)]_E \) and \( w \in [D_{re}(a(z);s';E).s']_E = [D_{re}(a(z);s';E)]_E \)

(b) \( [D_{re}(a(z);s';E)]_E \subseteq \{ w|a(z).w \in [s']_E \}: \)
If \( w \in [D_{re}(a(z);s';E)]_E \), then \( w = s_1.s_2 \), where:
\[
s_1 \in [D_{re}(a(z):s';E)]_E \text{ and } s_2 \in [s']_E
\]

By induction, \( a(z).s_1 \in [s]_E \). So, \( s_1.s_2 \in [s']_E \)

4. \( rs: \)
The following together show that \( [D_{re}(a(z);r.s;E)]_E = \{ w|a(z).w \in [r.s]_E \} \)

(a) \( \{ w|a(z).w \in [r.s]_E \} \subseteq [D_{re}(a(z);r.s;E)]_E: \)
If a word \( a(z).w \in [r.s]_E \), then by definition \( a(z).w = w_1.w_2 \) s.t. \( w_1 \in [r]_E \wedge w_2 \in [s]_E \). By definition,
\[
[D_{re}(a(z);r.s;E)]_E = [D_{re}(a(z);r.E).s]_E \cup [v(r).D_{re}(a(z);s;E)]_E
\]
There are two cases
i. \( w_1 = a(z).w'_1 \). Then by induction, \( w'_1 \in [D_{re}(a(z);r.E)]_E \), so \( w'_1.w_2 \in [D_{re}(a(z);r.E)]_E \)
ii. \( w_1 = \epsilon \), and \( w_2 = a(z).w'_2 \). Since \( w_1 = \epsilon \in [r]_E \), \( v(r) = \epsilon \). By induction, \( w'_2 \in [D_{re}(a(z);s;E)]_E \), so \( w \in \) \( [v(r).D_{re}(a(z);s;E)]_E \).

This includes all possibilities for any word in \( \{ w|a(z).w \in [r.s]_E \} \).

(b) \( [D_{re}(a(z);r.s;E)]_E \subseteq \{ w|a(z).w \in [r.s]_E \}: \)
If \( w \in [D_{re}(a(z);r.s;E)]_E \), then either:

i. \( w \in [D_{re}(a(z);r.E).s]_E \). By induction, \( w = w'_1.w_2 \) such that \( a(z).w'_1 \in [r]_E \) and \( w_2 \in [s]_E \). So, \( a(z).w \in [r.s]_E \).

ii. \( w \in [v(r).D_{re}(a(z);s;E)]_E \). Then either \( v(r) = \epsilon \) and there are no such \( w \), or \( v(r) = \epsilon \) and by induction, \( a(z).w \in [s]_E \). Since \( r \) is nullable, \( [s]_E \subseteq [r.s]_E \).

This includes all possibilities for any word in \( [D_{re}(a(z);r.s;E)]_E \).

Lemma 1. For any \( B \triangleright b_1(@y_1) \triangleright b_2(@y_2) \triangleright B' \triangleright s \) in prefix form and any \( E \) it is closed under:
\[
[B \triangleright b_1(@y_1) \triangleright b_2(@y_2) \triangleright B' \triangleright s]_E = [B \triangleright b_2(@y_2) \triangleright b_1(@y_1) \triangleright B' \triangleright s]_E
\]

Proof. By induction on the length of \( B \) preceding the two to be exchanged. In the base case, expand the definitions twice. In the induction case, just exchange the smaller list.

Lemma 2. If \( b(@y) \triangleright B \triangleright s \) is implementable, then so is \( B \triangleright s \).

Proof. By expanding the semantics to get some value for \( y \), and then using impl property on the compatible environments with \( B \).

Lemma 3. If \( B \triangleright s \) is implementable, then for any \( a \notin B,z,E \) where \( B \triangleright s \) is closed, and compatible environment \( G, B \triangleright D_{re}(a(z);s;(E,G)) \) is implementable as well.

Proof. By induction on the structure of \( s \).

Theorem 6. Restatement of Theorem 3: For all \( a,z,B,s,E \) where \( B \triangleright s \) is in prefix form (\( B \) is a non-redundant list of bindings and \( s \) is binding-free) and implementable:
\[
[D_{re}(a(z);s';E',Asgn0(B'))]_{E',Asgn0(B')} = \{ w'|a(z).w' \in [B \triangleright s]_E \}
\]
where \( B',E' = D_{bind}(a(z),B,E) \)
Proof. By induction on the number of bindings in \(B\). When \(B\) has no bindings, just use 1. When \(B\) has a binding with event \(b\), proceed by cases on whether \(a \in B\).

1. If \(a \in B\), then by 1, we can assume without loss of generality that it is the first binding in \(B\) (i.e. \(B = a(\@y) \triangleright B' \triangleright s\)). Also, since \(B\) is not redundant, \(D_{\text{bind}}(a(z); B'; E, y \leftarrow z) = (B', E, y \leftarrow z)\). Then \(D_{\text{bind}}(a(z); B; E) = (B', (E, y \leftarrow z))\). Next, we can expand the definition of \([B \triangleright s]_E\):

\[
\[a(\@y) \triangleright B' \triangleright s\]_E = \{w_1.a(z), w_2 \in [B' \triangleright s]_{E, y \leftarrow z} \text{ and } a \not\in w_1\} \cup \{w|\forall z'.w \in [B' \triangleright s]_{E, y \leftarrow z} \text{ and } a \not\in w\}
\]

There are clearly no \(a(z), w \in S_2\), since they cannot contain the event \(a\). Furthermore, if any word \(a(z).w = w_1.a(z).w_2 \in S_1, w_1\) must be \(\varepsilon\). So:

\[
\{w'|a(z), w' \in [a(\@y) \triangleright B' \triangleright s]_E\} = \{w'|a(z), w' \in S_1\}
\]

We now want to show that this set is equal to \([B' \triangleright D_{\text{re}}(a(z); s; E, y \leftarrow z, \text{Asgn0}(B'))]_{E, y \leftarrow z, \text{Asgn0}(B')}\), which is true directly by the induction hypothesis.

\[
[B' \triangleright D_{\text{re}}(a(z); s; E, y \leftarrow z, \text{Asgn0}(B'))]_{E, y \leftarrow z, \text{Asgn0}(B')} = \{w'|a(z), w' \in [B' \triangleright s]_{E, y \leftarrow z}\} \quad \text{By induction}
\]

2. If \(a \not\in B\), then \(B = b(\@y) \triangleright B' \triangleright s\). \(D_{\text{bind}}(a(z); B; E) = (B, E)\), since \(a\) doesn’t appear in the bindings. So, we can expand the definition of \([B \triangleright s]_E\):

\[
[b(\@y) \triangleleft B' \triangleleft s]_E = \{w_1.b(z), w_2 \in [B' \triangleright s]_{E, y \leftarrow z} \text{ and } b \not\in w_1\} \cup \{w|\forall z'.w \in [B' \triangleright s]_{E, y \leftarrow z} \text{ and } b \not\in w\}
\]

Next, we can expand the definition of \([b(\@y) \triangleleft B' \triangleleft s]_{E, \text{Asgn0}(b(\@y) \triangleright B')}\):

\[
[b(\@y) \triangleright B' \triangleleft D_{\text{re}}(a(z); s; \text{E, y \leftarrow 0, Asgn0}(B'))]_{E, \text{Asgn0}(b(\@y) \triangleright B')} = \{w|a(z), w' \in [B' \triangleright D_{\text{re}}(a(z); s; E, y \leftarrow 0, \text{Asgn0}(B'))]_{E, y \leftarrow z, \text{Asgn0}(B')} \text{ and } b \not\in w_1\}
\]

Because \(b(\@y) \triangleright B' \triangleright s\) is implementable, so is \(B' \triangleright D_{\text{re}}(a(z); s; E, y \leftarrow 0, \text{Asgn0}(B'))\) by using 2 and 3. Using this, we have the following in both \(S_3\) and \(S_4\):

\[
[B' \triangleright D_{\text{re}}(a(z); s; E, y \leftarrow 0, \text{Asgn0}(B'))]_{E, y \leftarrow z, \text{Asgn0}(B')} = [b(\@y) \triangleright s]_{E, y \leftarrow z, \text{Asgn0}(B')} = \{w|a(z), w \in [B' \triangleright s]_{E, y \leftarrow z}\}
\]

This lets us use the induction hypothesis on \(S_3\) and \(S_4\), since \(B' \triangleright D_{\text{re}}(a(z); s; E, y \leftarrow 0, \text{Asgn0}(B'))\) is implementable by 2 and 3:

\[
[B' \triangleright D_{\text{re}}(a(z); s; E, y \leftarrow z, \text{Asgn0}(B'))]_{E, y \leftarrow z, \text{Asgn0}(B')} = \{w|a(z), w \in [B' \triangleright s]_{E, y \leftarrow z}\}
\]

Plugging this into the definitions above gives the following:

\[
[b(\@y) \triangleright B' \triangleright D_{\text{re}}(a(z); s; \text{E, y \leftarrow 0, Asgn0}(b(\@y) \triangleright B'))]_{E, \text{Asgn0}(b(\@y) \triangleright B')} = \{w'|w_1, b(z), w_2 \in [B' \triangleright D_{\text{re}}(a(z); s; E, y \leftarrow 0, \text{Asgn0}(B'))]_{E, y \leftarrow z, \text{Asgn0}(B')} \text{ and } b \not\in w_1\}
\]

\[
[S_3'] \cup \{w'|\forall z'.a(z), w' \in [B' \triangleright s]_{E, y \leftarrow z} \text{ and } b \not\in w\}
\]

This concludes the proof, since these two sets are the same as \(S_1\) and \(S_2\), except for the requirement that the words start with \(a(z)\). Finally, it is clear that \(b\) is not in a word \(a(z).w\) if \(b\) is not in \(w\).
Lemma 4. For all $a$, $z$, $E$, and $B$:

$$D_{bind}(a(z); B; E, \text{Asgn0}(B)) = B', (E', \text{Asgn0}(B'))$$

Where $B', E' = D_{bind}(a(z); B; E)$

Proof. Observe that the initial values in $E$ do not matter for $D_{bind}$. If $a \in B$, then afterwards it is not in $B$, and each variable that was paired with $a$ now has a value in $E'$. These values would have overwritten the 0 added from $\text{Asgn0}(B)$. The values of all other variables do not change. If $a \notin B$, then neither $B$ nor $E$ change.

Lemma 5. For all $a$, $z$, $E$, $B$, $s$ where $B \triangleright s$ is closed under $E$:

$$\llbracket B \triangleright s \rrbracket_E = \llbracket B \triangleright s \rrbracket_{E, \text{Asgn0}(B)}$$

Proof. By induction on the length of $B$.

Base case: When $B$ is empty, $\text{Asgn0}(B)$ adds no variables, so they are the same.

Induction step: $B = b(\@y) \triangleright B'$:

$$\llbracket B \triangleright s \rrbracket_{E, \text{Asgn0}(B)} = \{w_1.b(z).w_2 \in \llbracket B' \triangleright s \rrbracket_{E, \text{Asgn0}(B), y \leftarrow z} \mid b \notin w_1 \} \cup \{w \mid b \notin w \text{ and } \forall z, w \in \llbracket B' \triangleright s \rrbracket_{E, \text{Asgn0}(B), y \leftarrow z} \}\}$$

By induction:

$$\llbracket B \triangleright s \rrbracket_E = \llbracket B \triangleright s \rrbracket_E$$

Theorem 7. Restatement of Theorem 2: For any word $w$, bindings $B$, environment $E$, and predicates $P$, if an FLM pattern $B \triangleright s$ is in prefix form, $B \triangleright s$ is closed under $E$, and $B \triangleright s$ is implementable, then: $T_w(w, B, (E, \text{Asgn0}(B)), P) \in L(T_{re}(s, P))$ if and only if $w \in \llbracket B \triangleright s \rrbracket_E$.

Proof. By induction on the length of $w$.

For the base case, $T_{re}$ does not change the length of accepted words. So, $\varepsilon \in T_{re}(s, P) \iff \varepsilon \in \llbracket B \triangleright s \rrbracket_E$ ($\varepsilon$ is the only word of length 0).

For the induction step: $w = a(z).w'$; $s' = D_{re}(a(z); s; E, \text{Asgn0}(B))$; and $B'; E' = D_{re}(a(z); B; E)$.

Starting with:

$$T_w(a(z).w', B, (E, \text{Asgn0}(B)), P) \in L(T_{re}(s, P))$$

By the definition of $T_w$ and applying 4, This is equivalent to:

$$T_l(a(z), (E', \text{Asgn0}(B'))), P). T_w(w', B', (E', \text{Asgn0}(B')), P) \in L(T_{re}(s, P))$$

Where $B', (E', \text{Asgn0}(B')) = D_{bind}(a(z), B, (E, \text{Asgn0}(B)))$. By the definition of the classic derivative, this is equivalent to:

$$T_w(w', B', (E', \text{Asgn0}(B')), P) \in L(D_{class}(T_l(a(z), (E', \text{Asgn0}(B')), P); T_{re}(s, P)))$$

By 4, this is equivalent to:

$$w' \in \llbracket B' \triangleright s \rrbracket_{E, \text{Asgn0}(B')}$$

By 6, this is true if and only if:

$$a(z).w' \in \llbracket B \triangleright s \rrbracket_{E, \text{Asgn0}(B)}$$

Finally, we can use 5 to get the result:

$$w \in \llbracket B \triangleright s \rrbracket_E$$
Lemma 6. If \( r \xrightarrow{\text{rw}} r' \) and \( r \) is implementable, then for all \( E \) such that \( r \) and \( r' \) are closed under \( E \), \( \llbracket r \rrbracket_E = \llbracket r' \rrbracket_E \).

Proof. By cases on which rewrite rule is used.

1. \( r = b(@y; p).s \xrightarrow{\text{rw}} b(@y) \triangleright b(p).s \)

\[
\llbracket b(@y) \triangleright b(p).s \rrbracket_E = \{ w_1.b(z).w_2 \in \llbracket b(p).s \rrbracket_{E,y \rightarrow \cdot} | b \not\in w_1 \} \cup \{ w | \forall z.w \in \llbracket b(p).s \rrbracket_{E,y \rightarrow \cdot} \land b \not\in w \}
\]

The right-hand set is empty, since any word in it must start with a \( b \) event. Similarly, \( w_1 \) must be \( \varepsilon \) for any word in the left-hand set.

\[
\llbracket b(@y).s \rrbracket_E = \{ b(z).w_2 \in \llbracket b(p).s \rrbracket_{E,y \rightarrow \cdot} \} = \bigcup_{z \in \Sigma} \{ \llbracket b(z) \rrbracket_{p(z)} \rrbracket_E \circ [r]_{E,y \rightarrow \cdot} \}
\]

\[
= \llbracket b(@y; p).s \rrbracket_E
\]

2. \( r = B_1 \triangleright (s_1. (b(@y) \triangleright s_2) \xrightarrow{\text{rw}} (B_1 \triangleright b(@y) \triangleright s_1.s_2) \)

In this case, we know that \( b(@y) \triangleright s_2 \) and \( B_1 \triangleright s_1 \) are implementable, that \( b \not\in B_1 \triangleright s_1 \), and that \( y \not\in B_1 \). We can now show the two are equivalent by induction on the length of \( B_1 \). In the base case, we use the fact that \( y \) is out of scope in \( s_1 \):

\[
\llbracket s_1. (b(@y) \triangleright s_2) \rrbracket_E = \{ w_1.w_2 | w_1 \in [s_1]_E \land w_2 \in \llbracket (b(@y) \triangleright s_2) \rrbracket_E \}
\]

\[
= \{ w_1.w_2 | (\forall z.w_1 \in [s_1]_E \land w_2 \in \llbracket (b(@y) \triangleright s_2) \rrbracket_E \}
\]

\[
= \{ w_1.w_2 | (\forall z.w_1 \in [s_1]_{E,y \rightarrow \cdot} \land w_2 \in \llbracket (b(@y) \triangleright s_2) \rrbracket_E \}
\]

\[
= \{ w_1.w_2 | (\forall z.w_1 \in [s_1]_{E,y \rightarrow \cdot}) \land (w_2 \in \{ w_1.b(z).w_2 \in \llbracket s_2 \rrbracket_{E,y \rightarrow \cdot} | b \not\in w' \} \cup \{ w | \forall z.w \in \llbracket s_2 \rrbracket_{E,y \rightarrow \cdot} \land b \not\in w \}) \}
\]

Because we are quantifying over \( z \) for \( w_1 \), it is clear that any \( w_1.w_2 \) in this set must also be in \( \llbracket b(@y) \triangleright s_1.s_2 \rrbracket_E \), depending on which set \( w_2 \) is in:

\[
\llbracket b(@y) \triangleright s_1.s_2 \rrbracket_E = \{ w_1.w_2 | w_1 \in [s_1]_E \land w_2 \in \llbracket (b(@y) \triangleright s_2) \rrbracket_E \}
\]

If \( w_2 \) is in the right-hand set, then \( w_2'' = w_1.w_1' \) and \( w_2'' = w_2' \). Otherwise, \( w = w_1'.w \). The same expansions show membership the other way; if a word \( w''_1.b(z).w''_2 \in \llbracket s_1.s_2 \rrbracket_{E,y \rightarrow \cdot} \), then a prefix of \( w''_1 \in [s_1]_{E,y \rightarrow \cdot} = [s_1]_E \), and the rest is in the right-hand set above. Otherwise, a prefix of \( w'' \) is in \( s_1 \) and the rest is in the left-hand set above.

The induction case follows directly from expanding the definition:

\[
\llbracket B_1 @(y_1) \triangleright B'_1 \triangleright (s_1. (b(@y) \triangleright s_2)) \rrbracket_E
\]

\[
= \{ w_1.b(z).w_2 \in \llbracket B'_1 \triangleright (s_1. (b(@y) \triangleright s_2)) \rrbracket_{E,y \rightarrow \cdot} | b \not\in w_1 \} \cup \{ w | \forall z.w \in \llbracket B'_1 \triangleright (s_1. (b(@y) \triangleright s_2)) \rrbracket_{E,y \rightarrow \cdot} \land b \not\in w \}
\]

Using induction:

\[
\llbracket B'_1 \triangleright (s_1.s_2) \rrbracket_{E,y \rightarrow \cdot} = \{ w_1.b(z).w_2 \in \llbracket B'_1 \triangleright b(@y) \triangleright (s_1.s_2) \rrbracket_{E,y \rightarrow \cdot} | b \not\in w_1 \} \cup \{ w | \forall z.w \in \llbracket B'_1 \triangleright b(@y) \triangleright (s_1.s_2) \rrbracket_{E,y \rightarrow \cdot} \land b \not\in w \}
\]

3. \( (b(@y_1) \triangleright s_1) + (b(@y_2) \triangleright s_2) \xrightarrow{\text{rw}} b(y_1) \triangleright (s_1 + [y_1/y_2]s_2) \)

Here, we assume both \( b(@y_1) \triangleright s_1 \) and \( b(@y_2) \triangleright s_2 \) are implementable. The proof follows directly from expanding the two \( \triangleright \) expressions, and substituting the variables in \( s_2 \).

\[
\llbracket b(@y_1) \triangleright (s_1 + [y_1/y_2]s_2) \rrbracket_E
\]

\[
= \{ w_1.b(z).w_2 \in [s_1 + [y_1/y_2]s_2]_{E,y \rightarrow \cdot} | b \not\in w_1 \} \cup \{ w | \forall z.w \in [s_1 + [y_1/y_2]s_2]_{E,y \rightarrow \cdot} \land b \not\in w \}
\]

\[
= \{ w_1.b(z).w_2 \in [s_1]_{E,y \rightarrow \cdot} \cup [y_1/y_2]s_2]_{E,y \rightarrow \cdot} | b \not\in w_1 \} \cup \{ w | \forall z.w \in [s_1]_{E,y \rightarrow \cdot} \cup [y_1/y_2]s_2]_{E,y \rightarrow \cdot} \land b \not\in w \}
\]

\[
= \{ w_1.b(z).w_2 \in [s_1]_{E,y \rightarrow \cdot} \cup [y_1/y_2]s_2]_{E,y \rightarrow \cdot} | b \not\in w_1 \} \cup \{ w | \forall z.w \in [s_1]_{E,y \rightarrow \cdot} \cup [y_1/y_2]s_2]_{E,y \rightarrow \cdot} \land b \not\in w \}
\]

\[
= \llbracket b(@y_1) \triangleright s_1 \rrbracket + \llbracket b(@y_2) \triangleright s_2 \rrbracket_E
\]
We assume that \( b(\@y_1) \triangleright s_1 \) and \( c(\@y_2) \triangleright s_2 \) are implementable. The proof follows directly from expanding the definitions of the two \( \triangleright \) expressions.

\[
\begin{align*}
\text{We assume that } b(\@y_1) \triangleright s_1 \text{ and } c(\@y_2) \triangleright s_2 \text{ are implementable. The proof follows directly from expanding the definitions of the two } \triangleright \text{ expressions.}
\end{align*}
\]
3. Finally, we can apply the induction hypothesis for $s_1$ and $s_2$ and roll back up the definitions to get the desired result:

$$\begin{align*}
&= \left[ D_{re}(a(z); s_1; ((E, y \leftarrow z_2), G'_2)) \right]_{(E, y \leftarrow z_2), G'_2} \circ [s_2]_{(E, y \leftarrow z_2), G'_2} \cup [v(s)]_{(E, G'_2), y \leftarrow z_2} \circ \left[ D_{re}(a(z); s_2; ((E, G'_2), y \leftarrow z_1)) \right]_{(E, G'_2), y \leftarrow z_2}
\end{align*}$$

On the left, we can apply the induction hypothesis directly. On the right, we undo the substitution and use it, then roll back up the definition.

$$\begin{align*}
&= \left[ D_{re}(a(z); s_1; E, y_1 \leftarrow z_1) \right]_{E, y_1 \leftarrow z_2} + D_{re}(a(z); [y_1/y_2]s_2; E, y_1 \leftarrow z_1) \right]_{E, y_1 \leftarrow z_2} \\
&= \left[ D_{re}(a(z); s_1; E, y_1 \leftarrow z_1) \right]_{E, y_1 \leftarrow z_2} \cup D_{re}(a(z); [y_1/y_2]s_2; E, y_1 \leftarrow z_1) \right]_{E, y_1 \leftarrow z_2}
\end{align*}$$

4. We assume that $b(\@y_1) \triangleright s_1$ and $c(\@y_2) \triangleright s_2$ are implementable, and expand definitions, writing $G_1$ as $y_1 \leftarrow z_1$ and $G_2$ as $(y_1 \leftarrow z_2)$:

$$\begin{align*}
&= \left[ D_{re}(a(z); s_1 + [y_1/y_2]s_2; E, y_1 \leftarrow z_1) \right]_{E, y_1 \leftarrow z_2} + D_{re}(a(z); [y_1/y_2]s_2; E, y_1 \leftarrow z_1) \right]_{E, y_1 \leftarrow z_2} \\
&= \left[ D_{re}(a(z); s_1 + [y_1/y_2]s_2; E, y_1 \leftarrow z_1) \right]_{E, y_1 \leftarrow z_2} \cup D_{re}(a(z); [y_1/y_2]s_2; E, y_1 \leftarrow z_1) \right]_{E, y_1 \leftarrow z_2}
\end{align*}$$

On the left, we can apply the induction hypothesis directly. On the right, we undo the substitution and use it, then roll back up the definition.

Finally, we can apply the induction hypothesis for the other variables, using the implementability property. Then, we roll back up the definitions.

$$\begin{align*}
&= \left[ D_{re}(a(z); s_1 + [y_1/y_2]s_2; E, y_1 \leftarrow z_1, y_2 \leftarrow z_2) \right]_{E, y_1 \leftarrow z_2} \cup \left[ D_{re}(a(z); s_2; E, y_1 \leftarrow z_1, y_2 \leftarrow z_2) \right]_{E, y_1 \leftarrow z_2} \\
&= \left[ D_{re}(a(z); s_1 + s_2; E, y_1 \leftarrow z_1, y_2 \leftarrow z_2) \right]_{E, y_1 \leftarrow z_2} \cup \left[ D_{re}(a(z); s_2; E, y_1 \leftarrow z_1, y_2 \leftarrow z_2) \right]_{E, y_1 \leftarrow z_2}
\end{align*}$$

5. This proof is very similar to the above, because the binding-free $s_2$ is always implementable.

$$\begin{align*}
&= \left[ D_{re}(a(z); s_1 + s_2; E, y \leftarrow z_1) \right]_{E, y \leftarrow z_2} + D_{re}(a(z); [y_1/y_2]s_2; E, y \leftarrow z_1) \right]_{E, y \leftarrow z_2} \\
&= \left[ D_{re}(a(z); s_1 + s_2; E, y \leftarrow z_1) \right]_{E, y \leftarrow z_2} \cup D_{re}(a(z); s_2; E, y \leftarrow z_1) \right]_{E, y \leftarrow z_2}$$
\( y \) is out of scope in \( s_2 \), so we can replace its value with anything without changing the derivative.

\[
= \{D_{re}(a(z); s_1; E, y \leftarrow z_1)\}_{E, y \leftarrow z_2} \cup \{D_{re}(a(z); s_2; E, y \leftarrow z_2)\}_{E, y \leftarrow z_2}
\]

Now, we apply the induction hypothesis for \( s_1 \), and get the require result:

\[
= \{D_{re}(a(z); s_1; E, y \leftarrow z_2)\}_{E, y \leftarrow z_2} \cup \{D_{re}(a(z); s_2; E, y \leftarrow z_2)\}_{E, y \leftarrow z_2}
\]
\[
= \{D_{re}(a(z); s_1 + s_2; E, y \leftarrow z_2)\}_{E, y \leftarrow z_2}
\]

The \& forms of rules 3, 4, and 5 above have very similar proofs, replacing + and \( \cup \) with \& and \( \cap \).