Sequence Abstractions for Flexible, Line-Rate Network Monitoring

Abstract
We develop a high-level language called FLM that enables network operators to write programs that recognize and react to specific packet sequences. To be able to examine every packet, such programs must run in the switch hardware. 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, and it then generates an efficient implementation of the regular expression using SMT-based program synthesis. Our experiments find that FLM is highly expressive: It can easily express 15 diverse sequence monitoring tasks drawn from the literature. Our compiler can convert all of these programs to run on switch hardware in a way that easily fit within available pipeline stages and consume less than 10–25% of header fields and instruction words.

1 Introduction
Many network management tasks involve recognizing and reacting to 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 video flows using a fingerprint based on successive packets and then de-prioritize them to improve experience for other traffic [25]. Or, they can verify that network clients faithfully implement protocols such as the Dynamic Host Configuration Protocol (DHCP) by observing the protocol exchange [29].

Ideal sequence monitors are i) flexible: can easily 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 [29] 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) [6] enable hardware-based sequence monitoring, but are programmed in languages such as P4 [5, 27] that are too low level, making it hard to express and debug sophisticated tasks [16, 30, 33]. Hybrid systems like Marple and Sonata [12, 22] 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 [22].

In this paper, we present an abstraction of a packet sequence pattern that is both flexible and is 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 with its own local memory and a small 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 specify sophisticated 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 [14], for any accepted packet sequence. We have formally proven that our compilation process from patterns to pipeline stages preserves the original pattern semantics.

We evaluate FLM by encoding 15 diverse monitoring tasks drawn from prior work [11, 16, 19, 22, 25, 26, 29, 30, 32, 33]. 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 easily fit within the number of stages on the switch and consume less than 10–25% of metadata memory or instruction words.

In summary, this work makes three main contributions:
• FLM, a language to easily and concisely 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 abstractions and implementation are broadly applicable to architectures like PISA that rely on a pipeline of stages. Each stage has a set of series of registers and match-action tables, and the data between stages is isolated (i.e., registers from prior or future stages cannot be accessed).

To simplify our implementation, we build on Lucid [26], an event-oriented 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.

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 containing available IP addresses. The client sends a "Request" message for one of the addresses, which the server responds with a "DHCP_Ack" message by writing:

```
DHCP_Ack(int cip); // Identifier
```

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

```
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. The DHCP_Ack event is detected by checking for a packet’s corresponding header flag. From this packet, the operator parses the DHCP message payload and extracts the client’s MAC and its newly assigned IP. Any other packet is simply left as a generic IP_Pkt, 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:

```
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 the parameters as part of the regular expression alphabet would clearly make it too large. To remedy this, an FLM pattern allows for the binding of parameters to recognize patterns over very large alphabets (e.g., all IP addresses). The operator can record the value of the client’s assigned IP in the DHCP_Ack message by writing:

```
DHCP_Ack(int assigned = cip)
```

This will match any DHCP_Ack packet, and record the value of its cip parameter in the new variable assigned. To check whether future packets from the client are using this recorded IP, the operator can add a predicate over the parameters of an IP_Pkt by writing:

```
IP_Pkt(sip == assigned)
```

This pattern will match any IP_Pkt event whose sip parameter equals assigned. In our application, we also need the negation (IP_Pkt(sip != assigned)). Hence, using binding and predicates, the operator can now construct the following FLM pattern to detect DHCP misuse from a client:

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

Arrays of patterns. 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:

```
spec<2048> dhcp_misuse = ...
```
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.

\[
\text{IDX} = \{
    \text{DHCP\_Ack} \rightarrow \{\text{hash(cmac)}\},
    \text{IP\_Pkt} \rightarrow \{\text{hash(smac)}\}
\}
\]

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 [12, 29].

**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. A programmer may also use the `reset` function to set all state for the current index back to 0 and prepare it for reuse.

**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 and resets the pattern.

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

\[
\text{transition (state, input):}
\]
\[
    \text{if (state == s1 and input = i1):}
    \quad \text{return s2;}
    \]
\[
    \text{else if (state == s1 and input = i2):}
    \quad \text{return s3;}
    \]
\[
\]

This already contains more branches than are allowed in a single stage. 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...
The core data structure is an index using the expressions provided in the high-level DHCP example, in the handler of each event, we compute the name applied to it so far matches the FLM pattern. The pattern remains the same, and so is just boolean indicating whether or not the sequence of events to the state machine at index expression transition copied from the high-level program. To interact with it, the FLM pattern definition as a global definition. In the first step, we break a pattern into an intermediate representation that allows for more control of the inputs to an underlying state machine. The core data structure is an \texttt{index(size)}, which defines an array of state machines with size indices that match an FLM pattern. The pattern remains the same, and so is just copied from the high-level program. To interact with it, the expression \texttt{transition(name, idx, ev)} applies the event \texttt{ev} to the state machine at index \texttt{idx}, and evaluates to a boolean indicating whether or not the sequence of events applied to it so far matches the FLM pattern \texttt{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 correct MAC address). Then, we add \texttt{transition(dhcp_misuse, idx, thin)}, where the keyword \texttt{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 representation with explicit transition statements in event handlers.

**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 an \texttt{index(size)}, which defines an array of state machines with size indices that match an FLM pattern. The pattern remains the same, and so is just copied from the high-level program. To interact with it, the expression \texttt{transition(name, idx, ev)} applies the event \texttt{ev} to the state machine at index \texttt{idx}, and evaluates to a boolean indicating whether or not the sequence of events applied to it so far matches the FLM pattern \texttt{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 correct MAC address). Then, we add \texttt{transition(dhcp_misuse, idx, thin)}, where the keyword \texttt{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 representation with explicit transition statements in event handlers.

**Step 1 for DHCP.** The next two steps compile the remaining FLM pattern and \texttt{transition} statements from the intermediate representation into simple assignments and register operations. We will refer to the pseudocode in Figure 2, which represents the implementation of one \texttt{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 a register called \texttt{assigned}. Furthermore, we wish to compute an input character \texttt{c} from a finite alphabet by evaluating the predicates in the FLM pattern. This alphabet is composed of all of the event \texttt{types} of the original, followed by bit strings representing the values of the predicates. Because \texttt{IP\_Pkt} appears with a predicate, it is expanded to the letters \texttt{IP\_Pkt1} and \texttt{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 \texttt{unions} of any event with the same type where that

---

```
1  // Compute the character using the old variables
2  if (event.type == DHCP\_Ack) {
3    c = ACK;
4  // On ack, update the store value
5    assigned := event.cip;
6  if (event.type == IP\_Pkt and sip != assigned) {
7    c = IP0;
8  if (event.type == IP\_Pkt and sip == assigned) {
9    c = IP1;
10  // Synthesized mapping f from character to value
11  f(c){ if (c == ACK) {return 12;}
12    if (c == IP0) {return 0;}
13    if (c == IP1) {return 12;}}
14  // Synthesized mapping g from character to value
15  g(c){ if (c == ACK) {return 2;}
16    if (c == IP0) {return 8;}
17    if (c == IP1) {return 9;}}
18  // Synthesized update function to do transition
19  update\_state(curr, x, y) {
20    if(curr + y < 3){
21      return x \oplus 5;}
22    else{
23      return curr & 3;}}
24  // Update the stored state using the f and g maps
25  state := update\_state(state, f(c), g(c));
```

Figure 2: A DFA representation of the translated DHCP FLM pattern the memop 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.
we search through all of the state numberings and bit-wise op-
high-level pattern semantics.
level compiled P4 switch program correctly implements the
our compiler translations are correct – namely that the low-
mantics over packet traces. Later, in section 6, we prove that
In this section, we describe the FLM language, provide its

transition function to the state register with those values. It
current index, then computes the input character and
the code in Figure 2: it first reads and updates the variables
definition, and each

• A read-modify-write instruction that implements the transi-
• 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
different instructions available on the
• A mapping from states to integer representations, as shown
by the numbers preceding each state in Figure 2 (for exam-
ple, Start is represented by 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
different instructions available on the
• A read-modify-write instruction that implements the transi-
tion function of the DFA using operations available on the
switch and results from the f and g mappings. This function
is shown on line 19 of Figure 2.
The code in Figure 2 is laid out on the switch to compute e, 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 resulting integer represents an accepting
state in the DFA (in this example, whether the register action
returned 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 se-
manics over packet traces. Later, in section 6, we prove that
our compiler translations are correct – namely that the low-
level compiled P4 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<i> 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
do when the pattern is recognized.

Finally, to recognize properties such as liveness or timeouts,
such as detecting half-open TCP queries [22], we provide
a special event type called maintenance. This event type 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. An
event is a pair of an event type a ∈ 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)) and binding
the parameters in events (a(@y;p)) to be used in predicates.
Patterns (and their contained predicates) are evaluated under
a particular environment.

An environment E is a mapping from variables to integers,
with its domain denoted by Dom(E). The empty environment
is denoted by ".". An example environment is (y ← 3, t ← 12),
with domain {y,t}. A predicate p is a function from integer
to boolean that may contain one or more free variables, and is
closed under an environment E if the free variables of p are
contained in Dom(E). The evaluation of a predicate p applied
to an integer z under an environment E is denoted by \[ p(z) \in E \],
and exists only if p is closed under E.

In our examples, we use lambda notation to define predi-
cates. For example, \( \lambda x. 1 \) is a predicate that returns true
when the given integer is less than 10. We use the standard semantics
of lambda functions. Finally, an FLM pattern r is closed under
an environment E if all of its free variables appear in E.

On the left 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. For example, the
semantics of \( a(\lambda x. x \geq 10) \) under an empty environment, den-
noted \( \{ a(\lambda x. x \geq 10) \} \), is all events \( a(z) \) such that \( z \geq 10 \).
4.3 FLM Intermediate Representation

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

1. \texttt{reset(myname, idx)} is a statement that resets the state machine at index \texttt{idx} to its initial values.

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:

\begin{verbatim}
if (transition(myname, idx, this)) {
   response;
}
\end{verbatim}

myname is the name of the state machine, this is the event transitioned with (the current event for the handler), and idx is computed using the IDX block, and 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 pream-ble code for the DHCP example. Here, we describe more generally how to translate an FLM pattern into a related regular expression, which we will 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 all 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 whether the predicates in a pattern are true. In the following sections, we will use this simple running example pattern:

\[ b(@y; p_1).b(p_2) \]

Where \( p_1 = \lambda x.(x \geq 10) \) and \( p_2 = \lambda x.(x == y) \). This represents all strings of two events with type \( b \), whose parameters are both at least 10 and equal.

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. In this section, we show how to move the bindings up to the top-level of the pattern while preserving its semantics. To do so, we introduce a new form of a pattern:

\[ b(@y) \triangleright r \]

Intuitively, this construct binds the first occurrence of an event with type \( b \)'s value to the variable \( y \), and then proceeds with matching \( r \). We can rewrite the above example using this syntax as follows:

\[ b(@y) \triangleright (b(p_1).b(p_2)) \]

This pattern has the same semantics as \( b(@y;p_1).b(p_2) \), as the value of the first event of type \( b \) is bound to \( y \), and must match \( p_1 \). In general, the semantics of the \( \triangleright \) operator is:

\[
\langle b(@y) \triangleright r \rangle_E = \{ w_1, b(z).w_2 \in \langle r \rangle_E, y \triangleright z \mid b \not\in w_1 \}
\]

\[
\cup \{ w \mid b \not\in w \text{ and } \forall z, w \in \langle r \rangle_E, y \triangleright z \}
\]

The first set follows the intuition exactly: if \( b \) is a word, the first event of type \( b \) has its value bound to \( y \). The second set covers the case when no event of type \( b \) appears. In that case, we require that \( y \) not matter when matching \( r \), and so use a universal quantifier over all integers for its value.

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. In section 6, we show that if the following rewrite rules can transform your regular expression into a new one that meets a simple syntactic condition, it can always be implemented in a pipeline.

First, we give a name to the type of patterns we are targeting:

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

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

Note that FLM patterns in prefix form cannot contain @ bindings. We will convert all of the @ bindings into the new \( \triangleright \) syntax. To implement an FLM pattern already in prefix form, we simply check (at the beginning, before touching the underlying state for the pattern \( s \)) whether an incoming event matches any of the bindings, and record its value. In the next stage, we compute the predicate values using the new environment and implement the matching of the binding-free pattern \( s \). 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 \&.

For all of these rules, we show that if \( r \xrightarrow{rw} r' \), then for all environments \( E \) under which \( r \) is closed, \( \langle r \rangle_E = \langle r' \rangle_E \).

5.2 Eliminating predicates from patterns

Now that we have a binding-free pattern, we can transform it into a classic regular expression over a finite alphabet by eliminating all remaining predicates. We will continue using the binding-free example from above:

\[ b(p_1).b(p_2) \]

In this example, the alphabet \( A = \{ b \} \) and predicates \( P = [p_1, p_2] \), so the translation alphabet appends the possible truth values of each predicate onto the event type: \( \{ b00, b01, b10, b11 \} \). This new alphabet is finite, but contains all of the information we need to implement the corresponding pattern. In general, the alphabet for the translation of an FLM pattern \( r \) is defined as follows, where 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:

\[ \{ a.\text{bin}(n) \mid a \in A \land n < 2^{\text{len}(P(a))} \} \]
To translate patterns, we will define the translation $T_{re}$, which takes as input a pattern and the list of predicates present in the top-level pattern. To translate the single-event pattern $b(p_1)$, we take the union of any event with the same type where $p_1$ is true. In our example, this is $(b10 + b11)$. This is shown in general with the function $BigOr$:

**Definition 3.** $BigOr(b(p), P) = \forall \{b\} \circ \{bin(i) \mid i \leq 2^{len(P)} \text{ and the bit representing } p \text{ is } 1\}$

$BigOr(b(p_1), [p_1, p_2]) = (b10 + b11)$, 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 4.** The FLM pattern translation $T_{re}(r, P)$:

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

For our example, we use the above translations to get:

$T_{re}(b(p_1)\cdot b(p_2)) = (b10 + b11)\cdot (b01 + b11)$

This will match all strings with two $b$ events where the first matches $p_1$ and the second matches $p_2$, as intended.

### 5.3 Translating events

Next, 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 the event with parameter 12, $T_l(b\{12\}, y \leftarrow 15, [p_1, p_2]) = b10$, since $12 \geq 10$ and $12 \neq 15$. For the parameter 15, $T_l(b\{15\}, y \leftarrow 15, [p_1, p_2]) = b11$.

**Definition 5.** The event translation

$T_l(a(z), E, P) = a[\lfloor p_1(z) \rfloor]_E \ldots [\lfloor p_n(z) \rfloor]_E$

The language of our example translated regex (using the classical definition of regular expressions) is the set of four translated strings \{b10.b01, b10.b11, b11.b01, b11.b11\}. These are exactly the strings where the first event $b$ matches $p_1$ and the second matches $p_2$.

### 6 Translation Correctness

In this section, we define a notion of derivative of FLM patterns and use them to prove that the translation functions introduced in section 5 preserve the semantics of FLM patterns defined in subsection 4.2.

#### 6.1 Classical Derivatives

In 1964, Brzozowski [7] defined a derivative of a classical regular expression $r$ with respect to a character $c$ in its alphabet. This defines a new regular expression whose language is the set of strings $s$ for which $c.s$ is in the language of $r$. For example, $D_{class}(a.b.b) = b.b$, since $a.(b.b) \in L(a.b.b)$. Classical regular expression derivatives are used as an alternative to the Thompson construction to create a DFA from a regular expression. States are represented by intermediate regular expressions, and the transition function for each character leads to a new state given by the derivative on that character.

**Definition 6.** $D_{class}$ is the classical regular expression derivative from [23], and maps a character $c$ and regex $r$ to a new regex such that:

$L(D_{class}(c;r)) = \{ w \text{ such that } c.w \in L(r) \}$

Where $L(r)$ is the language of $r$.

#### 6.2 FLM Pattern Derivatives

To create state machines that recognize FLM patterns, we define the analogous notion of the derivatives of FLM patterns. We will define one for binding-free FLM patterns and one for lists of bindings. The first, $D_{re}$, outputs a new FLM pattern, and the other, $D_{bind}$, outputs a new environment and list of bindings. Using the example from before, where $p_1 = \lambda x.(x \geq 10)$ and $p_2 = \lambda x.(x == y)$:

$D_{re}(b\{12\}; b(p_1), b(p_2); y \leftarrow 12) = b(p_2)$

This denotes the derivative with respect to the event $b\{12\}$ of the binding-free FLM pattern $b(p_1), b(p_2)$ under an environment defining $y$. The result is correct because $b\{12\}.b(z)$ is in the set defined by $[b(p_1), b(p_2)]\{y \leftarrow 12\}$ for any $b(z) \in [b(p_2)]$.. On the right-hand side of Figure 3, we define the two derivatives.

As an example for the $D_{bind}$, consider a single binding:

$D_{bind}(b\{12\}; b(\@y); :) = \varepsilon, y \leftarrow 12)$

This starts with the empty environment. Because the incoming event type matches the binding, it adds its value to the output environment and removes it from the list. We show the following theorem about these derivative constructions. The proofs are contained in the appendix. First, for binding-free derivatives, $D_{re}$ aligns with the classical definition:

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

$\[[D_{re}(a(z); s; E)]_E = \{ w \mid a(z).w \in [s]_E \}$

Note that, because $s$ is binding free, the environment $E$ is fixed when evaluating the semantics of $[s]_E$. 

8
6.3 Correctness Theorem

Our translations are correct if, after we translate an FLM pattern into a DFA with \( T_r \), and feed it a string of events translated with \( T_l \), the DFA accepts only when the original string of events is in the language of the original pattern. If this is the case, we are safe to skip implementing the FLM pattern’s transitions, and instead implement the DFA’s.

Our proof of correctness is by induction on the length of an accepted word. We show that the relation between a pattern \( s \) and its translated DFA via \( T_r \) is preserved when transitioning the DFA using characters translated with \( T_l \). At the end of the word, we test whether the translated DFA is accepting, which we show is equivalent to the entire word being in the language of the original pattern. This means that it is sufficient to implement the translated DFA on the switch.

We summarize how correctness looks in Figure 4, which shows how matching a pattern evolves on a string of events \( w = a_1(z_1) \ldots a_n(z_n) \). The blue and gray bubbles at the top represent the binding, environment, and binding-free pattern beginning at \( B_0 \rightarrow S_0 \) and \( E_0 \). These evolve using \( D_{bind} \) and \( D_{re} \), which take as input the concrete events \( a_1(z_1) \ldots a_n(z_n) \). In the blue bubbles along the bottom, each \( A_i \) represents an FLM pattern translation; i.e. \( A_0 = T_r(S_0, P) \), where \( P \) is the list of predicates in \( S_0 \). Each \( c_i \) represents the letter translation of \( a_i(z_i) \). The \( A_i \) change with \( D_{clas} \) (a DFA transition), which takes as input a translated event \( c_i \). Finally, blue represents state and operations that must be implemented on the switch, while gray represents those we can leave out so as not to tax resources. The green numbers denote the ordering in the switch pipeline of the labelled operations. Because of the translation correctness, to recover whether \( w \in [B_0 \rightarrow S_0]z_n \), the switch simply tests whether \( \varepsilon \in L(A_0) \); equivalently, whether the final \( D_{clas} \) operation results in an accepting state of \( A_0 \).

To state this formally, we define the translation of a word \( \varepsilon \) (a string of events), which repeatedly applies \( T_l \) and \( D_{bind} \) to transform the word into a string of finite-alphabet characters. Using the same example, \( T_w = (b(12), b(20), b(@y), (p_1, p_2)) = b11.b10. \) Note that each step of this translation is implementable on a hardware switch, as \( T_l \) involves evaluating a simple predicate and \( D_{bind} \) checks equality of event types and stores parameters.

**Definition 7.** The word translation

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

Where \( \varepsilon', E' = D_{bind}(a_i(z_i); B; E) \)

The word translation of \( \varepsilon \) is \( \varepsilon \). \( T_w \) is placed before the state machine in the pipeline (green numbers 1 and 2 in Figure 4).

The rewrite rules defined 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. For the technical definition, see the Appendix. Finally, we have our correctness theorem illustrated in Figure 4, 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. \( Asgn(B) \) simply assigns 0 to each variable of \( B \).

**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
1 memop template (st, f, g):
2 b1 = [st,[0]] + [f,g,[0]] $$[=, !=, <, >]$$ c1;
3 b2 = [st,[0]] + [f,g,[0]] $$[=, !=, <, >]$$ c2;
4 if (b1 [1] & b2):
5   return [st,c1] [1,k,+,[0]] [f,g,c1];
6 else:
7   return [st,c1] [1,k,+,[0]] [f,g,c0];

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; represents a constant chosen by the synthesizer.

under E, and implementable, then:

\[ T_w(w, B, (E, \text{Asgn}0(B)), P) \in L(T_{re}(s, P)) \iff w \in \llbracket B \gg s \rrbracket_E \]

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 derivative (a DFA transition) in order to use theorem 2. We use SMT-based synthesis to fit a transition function into at most four register actions, which is the maximum allowed on the Tofino.

7.1 Synthesis Goal

Intuitively, to implement a DFA’s transition function in at most four register actions, the synthesizer will attempt to cleverly assign numbers to the DFA states and event types and simultaneously generate a short function composed of a fixed number of simple operations like bitwise operations, integer 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 \(q_0\), \(\Sigma = \{\sigma_0, \sigma_1 \cdots\}\) 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 \(Q = \{\text{Start}, \text{Acc}, \text{Ref}\}\), \(\Sigma = \{\text{Ack}, \text{IP0}, \text{IP1}\}\), \(F = \{\text{Acc}\}\), 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 \(\text{whichop}\) from \(\Sigma\) to \{0, 1, 2, 3\} that indicates which register action each character will use.
- Up to four functions that take the values of the maps \(R, f,\) and \(g\) on the current state \(q\) and input character \(\sigma\), and output \(R(\delta(q, \sigma))\) for all the letters in \(\Sigma\) that use each one. Furthermore, the functions must follow the syntax from Figure 5 in order to fit into a single register action.

An example of correct outputs for the DHCP example is also shown in Figure 2 (all characters use the same function).

7.2 Synthesis Implementation

To come up with these outputs, we use an SMT solver to do syntax-guided synthesis [1]. We make one bitvector variable for all of the states (the values of \(R\)), two bitvectors for all of the letters (\(f\) and \(g\)), and booleans to determine \(\text{whichop}\). To make the templates, we make boolean indicator variables for which operations, comparisons, and boolean comparisons are used. Then, we encode the templates as constraints over the state variables. If a satisfying assignment is found, we read it to get the output. An interesting problem is to find the best template for synthesis. There is a balance between expressiveness and generating a formula that is too large to be solved in a reasonable amount of time. For example, the operators plus, bitor, bitand, and xor are usually expressive enough, and adding more expands the SMT formula significantly without much benefit. The template shown in Figure 5 was sufficiently expressive but solvable for our examples.

8 Implementation

We implement the FLM compiler atop the Lucid framework [20, 26] using approximately 1500 lines of OCaml available on GitHub\(^1\). We also implemented DFA synthesis code in the compiler in OCaml using the z3 SMT solver [31]. 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) 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 and compile a diverse set of sequence monitoring tasks of interest to network operators. We identify 15 such tasks from prior work. Ten of them are pure monitoring tasks that we implemented from

---

\(^1\)GitHub link removed for anonymity
scratch in FLM, and the remaining five are Lucid programs with a complex set of internal events to which we added FLM-based runtime monitoring.

Table 1 shows these programs and summarizes our results, including the lines of code needed to express the examples, the synthesized DFA complexity, the compilation time of each compiler stage, and the number of Tofino resources needed to implement the task. 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.

We also find that our compiler is able to compile each of the programs to Intel Tofino, which demonstrate the line-rate monitoring capabilities of FLM. We discuss compilation results in more detail next.

### 9.1 Compilation time

Table 1 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 its three stages: from FLM to Lucid IR using our compiler, from IR to P4 using the existing Lucid backend, and from P4 to hardware using vendor compiler.

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 under a minute.

Programs decorated with FLM has slightly 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. Furthermore, the complexity of the pattern itself does not affect the backend’s and vendor compiler’s compiling time.

### 9.2 Hardware Resource Utilization

Table 1 also shows the hardware resource usage of the Tofino binaries for each program. The three most relevant metrics for FLM are the number of hardware pipeline stages used, the percentage of metadata fields (PHV) allocated, and the percentage of instruction words (VLIW) allocated. For Lucid programs we added monitors to (A1-A5), the resource usage is for the total functionality (base+monitoring); the delta over the base program is shown in parenthesis.
existing logic of an event handler.

Meanwhile, FLM Lucid programs uses minimal resource: around 7%-24% of PHV or VLIW, most of which is for setup (parsing and pre-processing). Adding FLM to an existing Lucid program only consumers 1%-7% additional PHV and VLIW. Interestingly, the resource usage for task A2 (Stateful (parsing and pre-processing). Adding FLM to an existing Lucid program only consumers 1%-7% additional PHV and VLIW. Interestingly, the resource usage for task A2 (Stateful (parsing and pre-processing). Adding FLM to an existing Lucid program only consumers 1%-7% additional PHV and VLIW. Interestingly, the resource usage for task A2 (Stateful (parsing and pre-processing). Adding FLM to an existing Lucid program only consumers 1%-7% additional PHV and VLIW. Interestingly, the resource usage for task A2 (Stateful (parsing and pre-processing). Adding FLM to an existing Lucid program only consumers 1%-7% additional PHV and VLIW. Interestingly, the resource usage for task A2 (Stateful (parsing and pre-processing). Adding FLM to an existing Lucid program only consumers 1%-7% additional PHV and VLIW. Interestingly, the resource usage for task A2 (Stateful

10 Related Work

The syntax for FLM programs is inspired by Aragog [29], 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.

A number of other works use data plane switches to recognize regular expressions appearing in packets for the purposes of content inspection. Early work that appeared before P4 was released includes [21] and [24]. More recent work has built on further hardware advances, and includes frameworks specialized for matching strings in a packet [15, 28]. [18] focuses on verifying the data plane execution path a packet experienced. Meanwhile, [13] focuses on searching for regular expressions within the payload of packets, and also 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-randing applications. Timed automata [2] can record the timing of events and place time constraints on transitions. Symbolic automata [9, 10] permit a very wide variety of predicates on transitions, but do not have memory other than their state, while register automata [17] can record characters in registers, and check only equality. Recently, symbolic register automata [8] were proposed, which combine their two namesakes.

Our preliminary paper [4] 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

Hardness of Synthesis. We have investigated the implementability and complexity of various DFA transition functions. Surprisingly, no simple relationships exist between a DFA’s description and its implementability as a compact branching program. Although more states and alphabet size generally increase synthesis difficulty, there are many implementable examples with more states and characters than known infeasible ones. Heuristics suggest that other DFA features are relevant, such as the number of unique outgoing and incoming edges for each state. For instance, a rejecting state with incoming edges from all other states imposes significant constraints on its numerical representation.

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. Our work is only platform-specific in one aspect, which is 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.

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.
References


13 Appendix

13.1 Implementability

Our rewrite rules in subsection 5.1 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 \( b_1 \langle @y_1 \rangle \triangleright b_2 \langle @y_2 \rangle \). We will write \( \text{Asgn}0(B) \) 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 \( B \):

\[
[D_r(a(z); s; E, G_1)]_{E,G_1} = [D_r(a(z); s; E, G_2)]_{E,G_2}
\]

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 \( a \) 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 \( D_r \) for events of type \( a \). We show that our rewrite rules preserve this property: if \( r \xrightarrow{r'} \) 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 \( D_{bind} \) instead of a constant one.

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

\[
[D_r(a(z); s; E', \text{Asgn}0(B'))]_{E', \text{Asgn}0(B')} = \{w|a(z), w \in [B \triangleright s]_E\}
\]

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

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 \( b(12) \), \( D_{bind}(b(12); b(\langle @y \rangle); ..) = (..(y \leftarrow 12)) \). Taking the pattern derivative using the new environment:

\[
D_r(b(12); b(\lambda x. x \geq 10). b(p_1); y \leftarrow 12) = b(p_2)
\]

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 \( [b(p_1), b(p_2)]_{y \leftarrow 12} \) is \( b(12)b(12) \).

13.2 Extensions

In this section, we go over various extensions to FLM to improve its expressiveness and usability.
13.2.1 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:

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

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 \textit{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 follow, using \textit{maintenance} for the new events.

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

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.

13.2.2 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 \textit{unambiguous concatenation}, which allows a pattern to be split into two parts that can be implemented sequentially.

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

\textbf{Definition 10.} The prefixes of an FLM pattern \( r \), denoted prefix\((r)\), is the set:

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

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

\textbf{Definition 11.} The Continuations of an FLM pattern \( r \), denoted continuation\((r)\) is the set:

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

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

\[ \text{prefix}(a.b^*) = \varepsilon + a.b^* \]
\[ \text{continuation}(a.b^*) = b^* \]
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 intersect 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.

### 13.3 Proofs of Theorems

**Theorem 4.** "Derivatives commute":

\( \forall a,z,s,E,P \) where \( s \) is binding free, both \( s \) and each predicate in \( P \) is closed under \( E \), and \( \text{preds}(s) \subseteq P \):

\[
D_{\text{clas}}(T_l(a(z);E,P);T_r(s,P)) = T_r(D_{\text{re}}(a(z);s;E),P)
\]

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

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

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

**Proof:** Proof by induction on the structure of \( s \). Base cases:

1. \( \emptyset \):

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

2. \( \varepsilon \):

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

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

Induction cases:
1. r + s:

\[ [D_r(a(z); r + s; E)]_E = [D_r(a(z); r; E)]_E \cup [D_r(a(z); s; E)]_E \]

By induction:

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

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

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

3. s*:

The following together show that \([D_r(a(z); s*; E)]_E \in \{ w | a(z).w \in [s*]_E \}

(a) \( \{ w | a(z).w \in [s*]_E \} \subseteq [D_r(a(z); s*; E)]_E \):

If \( a(z).w \in [s*]_E \), then \( a(z).w \in [s^i]_E \) for some smallest natural \( i \). \( i \) cannot be 0. Since \( a(z).w \in [s*]_E = [s.s^{i-1}]_E \), \( a(z).w = w_1.w_2 \) s.t. \( w_1 \in [s]_E \cap w_2 \in [s^{i-1}]_E \). \( s_1 \) cannot be \( e \), otherwise \( i \) could be smaller. So, \( w_1 = a(z).w'_1 \) and \( w = w'_1.w_2 \).

By induction, \([D_r(a(z); s; E)]_E = \{ w | a(z).w \in [s]_E \}\)

So, \( w'_1 \in [D_r(a(z); s; E)]_E \), \( w_2 \in [s*]_E \), so \( w \in [D_r(a(z); s; E).s*]_E \)

(b) \([D_r(a(z); s*; E)]_E \subseteq \{ w | a(z).w \in [s*]_E \}

If \( w \in [D_r(a(z); s*; E)]_E \), then: \( w = s_1.s_2 \), where:

\( s_1 \in [D_r(a(z); s; E)]_E \) and \( s_2 \in [s*]_E \)

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

4. r.s:

The following together show that \([D_r(a(z); r.s; E)]_E \in \{ w | a(z).w \in [r.s]_E \}

(a) \( \{ w | a(z).w \in [r.s]_E \} \subseteq [D_r(a(z); r.s; E)]_E \):

If a word \( a(z).w \in [r.s]_E \), then by definition \( a(z).w = w_1.w_2 \) s.t. \( w_1 \in [r]_E \cap w_2 \in [s]_E \). By definition, \([D_r(a(z); r.s; E)]_E = [D_r(a(z); r; s)]_E \cup [v(r).D_r(a(z); s; E)]_E \). There are two cases

i. \( w_1 = a(z).w'_1 \). Then by induction, \( w'_1 \in [D_r(a(z); r; E)]_E \) so \( w'_1.w_2 \in [D_r(a(z); r.s; E)]_E \)

ii. \( w_1 = e \), and \( w_2 = a(z).w'_2 \). Since \( w_1 = e \in [r]_E \), \( v(r) = e \). By induction, \( w'_2 \in [D_r(a(z); s; E)]_E \), so \( w \in [v(r).D_r(a(z); s; E)]_E \)

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

(b) \([D_r(a(z); r.s; E)]_E \subseteq \{ w | a(z).w \in [r.s]_E \}

If \( w \in [D_r(a(z); r.s; E)]_E \), then either:

i. \( w \in [D_r(a(z); r; s)]_E \). 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. \( w \in [v(r).D_r(a(z); s; E)]_E \). Then either \( v(r) = \emptyset \) and there are no such \( w \), or \( v(r) = e \) and by induction, \( a(z).w \in [s]_E \).

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

\( \square \)

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 $b(\cdot @ y) \triangleright B \triangleright s$ 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:

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

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. $B = a(\cdot @ y) \triangleright B' \triangleright s$). Also, since $B$ is not redundant, $D_{bind}(a(z); B'; E, y \leftarrow z) = (B', E, y \leftarrow z)$. Then $D_{bind}(a(z); B; E) = (B', (E, y \leftarrow z))$. Next, we can expand the definition of $[B \triangleright s]_E$:

$$[a(\cdot @ y) \triangleright B' \triangleright s]_E = \{w_1.a(z'), w_2 \in [B' \triangleright s]_{E, y \leftarrow z} \text{ and } a \notin w_1\} \cup \{w|z'.w \in [B' \triangleright s]_{E, y \leftarrow z} \text{ and } a \notin w\}$$

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 $. So:

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

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

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

$$[b(\cdot @ y) \triangleright B' \triangleright s]_E = \{w_1.b(z'), w_2 \in [B' \triangleright s]_{E, y \leftarrow z} \text{ and } b \notin w_1\} \cup \{w|z'.w \in [B' \triangleright s]_{E, y \leftarrow z} \text{ and } b \notin w\}$$

Next, we can expand the definition of $[b(\cdot @ y) \triangleright B' \triangleright D_{re}(a(z); s; E, \text{Asgn0}(b(\cdot @ y) \triangleright B'))]_{E, \text{Asgn0}(b(\cdot @ y) \triangleright B')}$:

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

$$= \{w_1.b(z'), w_2 \in [B' \triangleright D_{re}(a(z); s; E, y \leftarrow 0, \text{Asgn0}(B'))]_{E, y \leftarrow 0, \text{Asgn0}(B')} \text{ and } b \notin w_1\}$$

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

$$\cup \{w|z'.w \in [B' \triangleright D_{re}(a(z); s; E, y \leftarrow 0, \text{Asgn0}(B'))]_{E, y \leftarrow 0, \text{Asgn0}(B')}\}$$
Because \( b(\langle y \rangle) \triangleright B' \triangleright s \) is implementable, so is \( B' \triangleright D_{re}(a(z); s; E, y \leftarrow 0, \text{Asgn}(B')) \) by using 2 and 3. Using this, we have the following in both S3 and S4:

\[
\begin{align*}
B' \triangleright D_{re}(a(z); s; E, y \leftarrow 0, \text{Asgn}(B')) & = \{ w | a(z).w \in \| B' \triangleright s \|_{E, y \leftarrow z', \text{Asgn}(B')} \}
\end{align*}
\]

This lets us use the induction hypothesis on S3 and S4, since \( B' \triangleright D_{re}(a(z); s; E, y \leftarrow 0, \text{Asgn}(B')) \) is implementable by 2 and 3:

\[
\begin{align*}
B' \triangleright D_{re}(a(z); s; E, y \leftarrow z', \text{Asgn}(B')) & = \{ w | a(z).w \in \| B' \triangleright s \|_{E, y \leftarrow z'} \}
\end{align*}
\]

Plugging this into the definitions above gives the following:

\[
\begin{align*}
\| b(\langle y \rangle) \triangleright b' \triangleright D_{re}(a(z); s; E, \text{Asgn}(b(\langle y \rangle) \triangleright B')) \|_{E, \text{Asgn}(b(\langle y \rangle) \triangleright B')} & = \{ w' = w_1.b(z).w_2 | a(z).w' \in \| B' \triangleright s \|_{E, y \leftarrow z'} \text{ and } b \not\in w_1 \}
\end{align*}
\]

This concludes the proof, since these two sets are the same as S1 and S2, 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 \( B \):

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

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 \( \text{Asgn}(B) \). The values of all other variables do not change. If \( a \not\in B \), then neither \( B \) nor \( E \) change.

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

\[ \| B \triangleright s \|_E = \| B \triangleright s \|_{E, \text{Asgn}(B)} \]

**Proof.** By induction on the length of \( B \).

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

Induction step: \( B = b(\langle y \rangle) \triangleright B' \):

\[
\begin{align*}
\| B \triangleright s \|_{E, \text{Asgn}(B)} & = \{ w_1.b(z).w_2 | w_1, w_2 \in \| B \triangleright s \|_{E, \text{Asgn}(B), y \leftarrow z} \text{ and } b \not\in w \} \\
& = \{ w_1.b(z).w_2 | w_1, w_2 \in \| B \triangleright s \|_{E, y \leftarrow z, \text{Asgn}(B)} \text{ and } b \not\in w \} \\
\end{align*}
\]

By induction:

\[
\begin{align*}
\| B \triangleright s \|_E & = \{ w_1.b(z).w_2 | w_1, w_2 \in \| B \triangleright s \|_{E, y \leftarrow z} \text{ and } b \not\in w \} \\
& = \| B \triangleright s \|_E \]
\]

**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: \( T_\text{w}(w, B, (E, \text{Asgn}(B)), P) \in L(T_\text{re}(s, P)) \) if and only if \( w \in \| B \triangleright s \|_E \).
Proof. By induction on the length of $w$.

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

For the induction step: $w = a(z).w'$; $s' = D_r(a(z);s;E,\text{Asgn}0(B))$; and $B';E' = D_r(a(z);B;E)$. Starting with:

$T_w(a(z).w',B,(E,\text{Asgn}0(B)),P) \in L(T_	ext{re}(s,P))$

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

$T_i(a(z),(E',\text{Asgn}0(B'))),P).T_w(w',B',(E',\text{Asgn}0(B'),P) \in L(T_	ext{re}(s,P))$

Where $B',(E',\text{Asgn}0(B')) = D_{\text{bind}}(a(z),B,(E,\text{Asgn}0(B)))$. By the definition of the classic derivative, this is equivalent to:

$T_w(w',B',(E',\text{Asgn}0(B'),P) \in L(D_{\text{cl}}(T_i(a(z),(E',\text{Asgn}0(B'),P));T_	ext{re}(s,P)))$

By 4, this is equivalent to:

$w' \in [B' \triangleright D_r(a(z),s,(E',\text{Asgn}0(B')))][E',\text{Asgn}0(B')]$

By 6, this is true if and only if:

$a(z).w' \in [B \triangleright s][E,\text{Asgn}0(B)]$

Finally, we can use 5 to get the result:

$w \in [B \triangleright s][E]$

Lemma 6. If $r \xrightarrow{rw} r'$ 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. $r = b(\langle y \rangle;p) \xrightarrow{rw} b(\langle y \rangle) \triangleright b(p).s$

   $\llbracket b(\langle y \rangle) \triangleright b(p).s \rrbracket_E$

   $= \{ w_1.b(z).w_2 \in \llbracket b(p).s \rrbracket_{E,y \leftarrow z} | b \notin w_1 \} \cup \{ w | \forall z.w \in \llbracket b(p).s \rrbracket_{E,y \leftarrow z} \land b \notin w \}$

   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.

   $= \{ b(z).w_2 \in \llbracket b(p).s \rrbracket_{E,y \leftarrow z} \}$

   $= \bigcup z \in Z \{ b(z) \llbracket p(z) \rrbracket_E \triangleright r \rrbracket_{E,y \leftarrow z} \}$

   $= \llbracket b(\langle y \rangle;p).s \rrbracket_E$

2. $r = B_1 \triangleright (s_1.(b(\langle y \rangle) \triangleright s_2)) \xrightarrow{rw} (B_1 \triangleright b(\langle y \rangle) \triangleright s_1,s_2)$

   In this case, we know that $b(\langle y \rangle) \triangleright s_2$ and $B_1 \triangleright s_1$ are implementable, that $b \notin B_1 \triangleright s_1$, and that $y \notin B_1$. 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 $s_1$:

   $\llbracket s_1.(b(\langle y \rangle) \triangleright s_2) \rrbracket_E$

   $= \{ w_1,w_2 | w_1 \in \llbracket s_1 \rrbracket_{E} \land w_2 \in \llbracket (b(\langle y \rangle) \triangleright s_2) \rrbracket_{E} \}$

   $= \{ w_1,w_2 | (\forall z,w_1 \in \llbracket s_1 \rrbracket_{E,y \leftarrow z} \land w_2 \in \llbracket (b(\langle y \rangle) \triangleright s_2) \rrbracket_{E} \}$

   $= \{ w_1,w_2 | (\forall z,w_1 \in \llbracket s_1 \rrbracket_{E,y \leftarrow z} \land w_2 \in \llbracket s_1 \rrbracket_{E,y \leftarrow z} \land b \notin w_2 \} \cup \{ w | \forall z,w \in \llbracket s_1 \rrbracket_{E,y \leftarrow z} \land b \notin w \}$

   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 $\llbracket b(\langle y \rangle) \triangleright s_1,s_2 \rrbracket_E$, depending on which set $w_2$ is in:

   $\llbracket b(\langle y \rangle) \triangleright s_1.s_2 \rrbracket_E$

   $= \{ w''_1.b(z).w''_2 \in \llbracket s_1.s_2 \rrbracket_{E,y \leftarrow z} | b \notin w_1'' \} \cup \{ w'' | \forall z,w' \in \llbracket s_1.s_2 \rrbracket_{E,y \leftarrow z} \land b \notin w' \}$

21
If \( w'_2 \) is in the right-hand set, then \( w''_2 = w_1 . w'_2 \) and \( w''_2 = w'_2 \). Otherwise, \( w' = w_1 . w \). The same expansions show membership the other way; if a word \( w''_2 . b(z).w''_2 \in \llbracket s_1, s_2 \rrbracket_E, z = z_2 \), then a prefix of \( w''_2 \in \llbracket s_1 \rrbracket_E, z \in z_2 \), 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:

\[
\llbracket b(\langle y_1 \rangle \triangleright s_1 \cdot (b(\langle y \rangle) \triangleright s_2)) \rrbracket_E
= \{ w_1 . b(z) . w_2 \in \llbracket s_1 \cdot b(\langle y \rangle) \triangleright s_2 \rrbracket_E \mid b \notin w_1 \} \cup \{ w \mid \forall z . w \in \llbracket s_1 \cdot s_2 \rrbracket_E \}
\]

Using induction:

\[
\llbracket b(\langle y_1 \rangle \triangleright s_1 \cdot (b(\langle y \rangle) \triangleright s_2)) \rrbracket_E
= \{ w_1 . b(z) . w_2 \in \llbracket s_1 \cdot b(\langle y \rangle) \triangleright s_2 \rrbracket_E \mid b \notin w_1 \} \cup \{ w \mid \forall z . w \in \llbracket s_1 \cdot s_2 \rrbracket_E \}
\]

3. \( b(\langle y_1 \rangle \triangleright s_1) \cdot b(\langle y_2 \rangle \triangleright s_2) \overset{\text{rw}}{\rightarrow} b(y_1) \triangleright (s_1 + [y_1/y_2]s_2) \)

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

\[
\llbracket b(\langle y_1 \rangle \triangleright s_1 + [y_1/y_2]s_2) \rrbracket_E
= \{ w_1 . b(z) . w_2 \in \llbracket s_1 \cdot [y_1/y_2]s_2 \rrbracket_E \mid b \notin w_1 \} \cup \{ w \mid \forall z . w \in \llbracket s_1 \cdot [y_1/y_2]s_2 \rrbracket_E \}
\]

4. \( b(\langle y_1 \rangle \triangleright s_1) \cdot c(\langle y_2 \rangle \triangleright s_2) \overset{\text{rw}}{\rightarrow} b(y_1) \triangleright c(\langle y_2 \rangle) \triangleright (s_1 + s_2) \)

We assume that \( b(\langle y_1 \rangle \triangleright s_1) \) and \( c(\langle y_2 \rangle \triangleright s_2) \) are implementable. The proof follows directly from expanding the definitions of the two \( \triangleright \) expressions.

\[
\llbracket (b(\langle y_1 \rangle \triangleright s_1) + c(\langle y_2 \rangle) \triangleright s_2) \rrbracket_E
= \llbracket b(\langle y_1 \rangle \triangleright s_1) \rrbracket_E \cup \llbracket c(\langle y_2 \rangle) \triangleright s_2 \rrbracket_E
\]

Expanding the other definition:

\[
\llbracket b(\langle y_1 \rangle \triangleright c(\langle y_2 \rangle) \triangleright (s_1 + s_2)) \rrbracket_E
= \{ w \mid w = w_1 . b(z_1) . w_2 \in \llbracket c(\langle y_2 \rangle) \triangleright (s_1 + s_2) \rrbracket_E \mid b \notin w_1 \} \cup \{ w \mid \forall z . w \in \llbracket c(\langle y_2 \rangle) \triangleright (s_1 + s_2) \rrbracket_E \}
\]

After expanding this to another 4 sets, it is clear that \( \llbracket (b(\langle y_1 \rangle \triangleright s_1) + c(\langle y_2 \rangle) \triangleright s_2) \rrbracket_E \subseteq \llbracket b(\langle y_1 \rangle) \triangleright c(\langle y_2 \rangle) \triangleright (s_1 + s_2) \rrbracket_E \) by cases on which set \((S1, S2, S3, S4)\) a word is in, and similarly in the reverse direction.

5. \( b(\langle y \rangle) \triangleright s_2 \overset{\text{rw}}{\rightarrow} b(\langle y \rangle) \triangleright (s_1 + s_2) \)

This proof is very similar to the above with one fewer expansion, because the binding-free \( s_2 \) is always implementable.

The rules (equivalent to 3, 4, and 5) for \& have similar proof structure, but are simplified by using intersection rather than union.

\( \square \)

Lemma 7. If \( r \overset{\text{rw}}{\rightarrow} r' \) and \( r \) is implementable, then so is \( r' \).

Proof. By cases on which rewrite rule is used.
1. \( r = b(@y;p).s \xrightarrow{\text{rw}} b(@y) \triangleright b(p).s \)
We assume that \( s \) is binding free, so that \( b(@y) \triangleright b(p).s \) is in prefix form. Now, we show that it is implementable, for any \( a \neq b \), the derivative of \( s \) is the same no matter the environment, since the following does not depend on \( z_1 \) at all:

\[
D_r(e(a(z);b(p).s);E,y \leftarrow z_1)
= D_r(e(a(z);b(p).s);E,y \leftarrow z_1) + v(b(p).D_r(e(a(z);s);E,y \leftarrow z_1)
= 0.s + 0.D_r(e(a(z);s);E,y \leftarrow z_1)
= 0
\]

2. \( r = (B_1 \triangleright s_1).(b(@y) \triangleright s_2) \xrightarrow{\text{rw}} (B_1 \triangleright b(@y) \triangleright s_1.s) \)
In this case, we know that \( b(@y) \triangleright s_2 \) and \( B_1 \triangleright s_1 \) are implementable, that \( b \notin B_1 \triangleright s_1 \), and that \( y \notin B_1 \). Now, we can expand the expression, for some \( a(z) \notin B_1 \triangleright b \) and compatible environments \( G_1, G_2 \) to \( B_1 \triangleright b \). We will write these as \( G'_1, y \leftarrow z_1 \) and \( G'_2, y \leftarrow z_2 \), separating out the compatible environments to \( B_1 \) and \( B_2 \), and expand definitions:

\[
[D_r(e(a(z);s_1.s_2;(E,G_1)))]_{E,G_2}
= [D_r(e(a(z);s_1;(E,G_1)).s_2 + v(s_1).D_r(e(a(z);s_2;(E,G_1)))]_{E,G_2}
= [D_r(e(a(z);s_1;(E,G_1)).s_2 \cup [v(s_1).D_r(e(a(z);s_2;(E,G_1)))]_{E,G_2}
= \left[D_r(e(a(z);s_1;((E,y \leftarrow z_1),G'_1))).s_2 \cup [v(s_1).D_r(e(a(z);s_2;((E,G'_1),y \leftarrow z_1)))]_{(E,G'_2),y \leftarrow z_2}
\]

On the left, because \( y \) is not in scope in \( s_1 \), its value does not change the semantics of its derivative, so we can replace \( z_1 \) with \( z_2 \). Similarly, on the right we can replace \( G'_1 \) with \( G'_2 \) because those variables are out of scope in \( s_2 \):

\[
[D_r(e(a(z);s_1;((E,y \leftarrow z_2),G'_1))).s_2 \cup [v(s_1).D_r(e(a(z);s_2;((E,G'_2),y \leftarrow z_1)))]_{(E,G'_2),y \leftarrow z_2}
= [D_r(e(a(z);s_1;((E,y \leftarrow z_2),G'_1))).s_2 \cup [v(s_1).D_r(e(a(z);s_2;((E,G'_2),y \leftarrow z_1)))]_{(E,G'_2),y \leftarrow z_2}
\]

Finally, we can apply the inductive hypothesis for \( s_1 \) and \( s_2 \) and roll back up the definitions to get the desired result:

\[
[D_r(e(a(z);s_1;((E,y \leftarrow z_2),G'_2))).s_2 \cup [v(s_2).D_r(e(a(z);s_2;((E,G'_2),y \leftarrow z_2)))]_{(E,G'_2),y \leftarrow z_2}
= [D_r(e(a(z);s_1;((E,y \leftarrow z_2),G'_2))).s_2 \cup [v(s_2).D_r(e(a(z);s_2;((E,G'_2),y \leftarrow z_2)))]_{(E,G'_2),y \leftarrow z_2}
\]

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. Then, we can expand definitions, writing \( G_1 \) as \( (y_1 \leftarrow z_1) \) and \( G_2 \) as \( (y_1 \leftarrow z_2) \):

\[
[D_r(e(a(z);s_1 + [y_1/y_2]s_2;E,y_1 \leftarrow z_1))_{E,y_1 \leftarrow z_2}
= [D_r(e(a(z);s_1;E,y_1 \leftarrow z_1) + D_r(e(a(z);[y_1/y_2]s_2;E,y_1 \leftarrow z_1))_{E,y_1 \leftarrow z_2}
= \left[D_r(e(a(z);s_1;E,y_1 \leftarrow z_1))_{E,y_1 \leftarrow z_2} \cup [D_r(e(a(z);[y_1/y_2]s_2;E,y_1 \leftarrow z_1))_{E,y_1 \leftarrow z_2}
\]

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:

\[
[D_r(e(a(z);s_1;E,y_1 \leftarrow z_1))_{E,y_1 \leftarrow z_2} \cup [D_r(e(a(z);s_2;E,y_2 \leftarrow z_1))_{E,y_2 \leftarrow z_2}
= \left[D_r(e(a(z);s_1;E,y_1 \leftarrow z_1))_{E,y_1 \leftarrow z_2} \cup [D_r(e(a(z);s_2;E,y_2 \leftarrow z_2))_{E,y_2 \leftarrow z_2}
\]

\[
[D_r(e(a(z);s_1 + s_2;E,G_2))_{G_2}
\]

23
4. 
\[(b(@y_1) \triangleright s_1) + (c(@y_2) \triangleright s_2) \xrightarrow{nw} b(@y_1) \triangleright c(@y_2) \triangleright (s_1 + s_2)\]

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 \(y_1 \leftarrow z_1, y_2 \leftarrow z_2\) and \(G_2\) as \(y_1 \leftarrow z'_1, y_2 \leftarrow z'_2:\)

\[\|D_{re}(a(z):s_1 + s_2;E,y_1 \leftarrow z_1,y_2 \leftarrow z_2)\|_{E,y_1 \leftarrow z'_1,y_2 \leftarrow z'_2}\]

\[= \|D_{re}(a(z):s_1;E,y_1 \leftarrow z_1,y_2 \leftarrow z_2)\|_{E,y_1 \leftarrow z'_1,y_2 \leftarrow z'_2} + \|D_{re}(a(z):s_2;E,y_1 \leftarrow z_1,y_2 \leftarrow z_2)\|_{E,y_1 \leftarrow z'_1,y_2 \leftarrow z'_2}\]

\[y_1\) is out of scope in \(s_2\), and \(y_2\) is out of scope in \(s_1\). So, we can assign any value to them without changing the derivative:

\[= \|D_{re}(a(z):s_1;E,y_1 \leftarrow z_1,y_2 \leftarrow z'_2)\|_{E,y_1 \leftarrow z'_1,y_2 \leftarrow z'_2} \cup \|D_{re}(a(z):s_2;E,y_1 \leftarrow z'_1,y_2 \leftarrow z_2)\|_{E,y_1 \leftarrow z'_1,y_2 \leftarrow z'_2}\]

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

\[= \|D_{re}(a(z):s_1;E,y_1 \leftarrow z'_1,y_2 \leftarrow z'_2)\|_{E,y_1 \leftarrow z'_1,y_2 \leftarrow z'_2} \cup \|D_{re}(a(z):s_2;E,y_1 \leftarrow z'_1,y_2 \leftarrow z'_2)\|_{E,y_1 \leftarrow z'_1,y_2 \leftarrow z'_2}\]

5.

\[(b(@y) \triangleright s_1) + s_2 \xrightarrow{nw} b(@y) \triangleright (s_1 + s_2)\]

This proof is very similar to the above, because the binding-free \(s_2\) is always implementable.

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

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

\(y\) 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 \(\cap\).  