A Reactive Language for Analyzing Cloud Logs

Guillaume Baudart
IBM Research
USA

Louis Mandel
IBM Research
USA

Olivier Tardieu
IBM Research
USA

Mandana Vaziri
IBM Research
USA

Abstract

Log analysis is required in many domains, and especially in the emerging field of cloud computing. Cloud applications are often built by composing diverse services. When something goes wrong, finding the root cause of the problem can be difficult. Many services are only reachable through their Application Programming Interfaces (APIs) with no possibility for live introspection. In this context, logs become an essential tool for monitoring and debugging. Cloud services typically generate very large quantities of log messages, with formats that may not be well specified and may vary over time. In this paper, we present CloudLens, a language for the analysis of semi-structured textual data as found in logs, and specify its formal semantics. CloudLens is a reactive language and views logs as streams of objects. Our objective is to facilitate exploring the contents of logs interactively and to write reusable analyses succinctly, using familiar constructs. We implemented an interpreter for the Apache Zeppelin notebook to provide an interactive IDE. Our prototype implementation is open source and we report on a detailed case study using logs from the Apache OpenWhisk project.

CCS Concepts • **Software and its engineering → General programming languages**; *Data flow languages*;

Keywords Log analysis, Programming language, Reactive programming

ACM Reference Format:
Guillaume Baudart, Louis Mandel, Olivier Tardieu, and Mandana Vaziri. 2018. A Reactive Language for Analyzing Cloud Logs. In *Proceedings of the 5th ACM SIGPLAN International Workshop on Reactive and Event-Based Languages and Systems (REBLS '18), November 4, 2018, Boston, MA, USA*. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3281278.3281280

REBLS '18, November 4, 2018, Boston, MA, USA

ACM ISBN 978-1-4503-6070-8/18/11…$15.00
https://doi.org/10.1145/3281278.3281280

1 Introduction

Log analysis is required in many domains, and especially in the emerging field of cloud computing [7]. Cloud applications are often built by composing diverse services. For example, consider a simple utility that notifies the user when the temperature is below freezing. It might use a Location service to locate the user, a Weather service to find out the current temperature, then a notification tool to communicate with the user. When something goes wrong, finding the root cause of the problem can be difficult. Many services are only reachable through their Application Programming Interfaces (APIs) with no possibility for live introspection. In this context, logs become an essential, maybe unique tool for monitoring and debugging applications. What Brian Kernighan wrote in 1978 [14] remains surprisingly true today: *"The most effective debugging tool is still careful thought, coupled with judiciously placed print statements."*

Cloud services typically generate very large quantities of log messages. These often have some structure such as a timestamp, a hostname, or a process identifier. But they also contain plain text with diverse information in free form, e.g., error messages. While log sources can often be made to be more or less verbose, developers typically have no control over the format of the logs, which tend to vary from source to source. Furthermore, these formats can change over time, as the services evolve but continue to be used via the same API. When a problem arises, developers may not know what to look for in the logs, and even if they do, the shape of that information can be elusive.

Traditionally, developers use grep or write ad-hoc scripts (e.g., in Perl or Python) for log analysis. To help with this arduous task, many different techniques have been proposed: mining techniques to automatically detect patterns and anomalies in logs [5, 16, 17, 20–23], visualization techniques that help browse or summarize logs more effectively [1, 4, 9, 18, 19], as well as specialized query and programming languages [2, 10–12, 15]. These approaches complement one another: mining and visualization techniques help to gain insights, specialized languages allow investigating further and turning these insights into actionable knowledge. Despite many advances however, log analysis remains difficult, time consuming, and repetitive.

In this paper, we present CloudLens (https://cloudlens.github.io), a reactive dataflow language for the analysis of semi-structured textual data as found in logs. Our objective is to facilitate exploring the contents of logs interactively and to write reusable analyses succinctly, using familiar constructs. CloudLens is a stream language suitable for consuming logs, which are by nature streams of data. Its basic premises are that (1) parsing and computation can be interleaved arbitrarily, and (2) parsing needs not be exhaustive. Traditional programming languages and tools insist on parsing the entire input or sizable chunks of the input with great precision before starting any computation. This monolithic approach is inadequate for computing with streams of semi-structured data. In contrast, a CloudLens script takes as input a log stream and consumes one log entry at a time. Each log entry is matched against a series of patterns to discover structure and information (*match* construct) without any attempt at building a complete schema for the log entry or log stream. Matches can trigger actions (*process* construct) to compute metrics, emit alerts, advance in a state machine, etc. CloudLens is similar in spirit to the Awk programming language [2] – a Unix text processing utility – but views logs as streams of objects: each log entry gets parsed into an object with fields corresponding to discovered structure. This approach facilitates building data in a hierarchical way, making it easier to express analyses that need to group or correlate different log entries. CloudLens differs from Splunk [6] – a popular commercial log analysis tool – in that CloudLens is imperative: intermediate results of analyses can be held in state variables and do not need to be kept as artificial fields inside the records extracted from log entries. This makes it easier to express complex analyses in a natural way, especially those that require matching the start and end of a related set of events (e.g. a transaction).

CloudLens allows encapsulation and reuse of scripts via the definition and invocation of *lens*es. It is designed to handle both finite and infinite streams as well as nested streams, making it possible for example to analyze session logs embedded inside server logs, including multi-pass analyses on finite logs and slices of infinite logs. These features are facilitated by the approach of viewing logs as object streams.

CloudLens is a domain-specific language built on top of JavaScript. Top-level constructs of the language are novel, but the programmer uses JavaScript to declare variables or functions and implement CloudLens actions. This helps a novice user to get started with CloudLens, and in reusing existing libraries. Our design however is agnostic to the base language, which could just as easily have been Python, Swift, or OCaml. We picked JavaScript because it is commonly used by cloud developers.

Our prototype implementation is open source and we report on a detailed case study using logs from the Apache OpenWhisk project[1], which provides a serverless computing service to execute application logic on the Cloud. OpenWhisk has logs that are semi-structured, and are a good representative of logs found in typical cloud applications. OpenWhisk developers implemented a Slack[2] bot powered by CloudLens, and incorporated it into their build system, which proved to be very useful. Finally, we have implemented an interpreter for the Apache Zeppelin[3] notebook to provide an interactive IDE for CloudLens.

Our contributions are the following:

- A novel language for analyzing log streams (Section 3).
- A formal semantics of the core language (Section 4).
- An open-source implementation (Section 5).
- An interactive IDE based on Apache Zeppelin (Section 2).
- A case study with Apache OpenWhisk (Section 6).

We discuss related work in Section 7.

2 Overview

In this section, we give a high-level overview of CloudLens to introduce its programming model and environment. We use a simple log and sample script to illustrate the idea informally. Consider the log of Figure 1,

```
Alice picked 4 apples
Bob picked 6 apples
Carol picked 7 oranges
Dan picked 3 apples
```

Figure 1. Simple log.

and suppose we wished to know the total number of apples picked. Notice that the log has some uniformity: each number of apples is followed by a space and by the word "apples". We cannot simply grep for the total number, since that information is not readily present in the log. However, there is data that we can process and aggregate in order to compute the desired result. Much of log analysis is about deriving higher-level data from bits of information.

Figure 2 shows a CloudLens notebook. This is a web-based interface used for data analytics. Notebooks are organized in *paragraphs*, which are code editors with an associated interpreter. Paragraphs can be executed independently, and their results appear below the code. This interface enables working with different languages and tools, and provides a wide variety of visualizations. It helps writing a script piece by piece and inspecting intermediate results. It is easily shared between developers, and requires very little installation since it is web-based. We have extended the Apache Zeppelin notebook with a CloudLens interpreter, and use it as our Integrated Development Environment (IDE).

The first paragraph in the notebook of Figure 2 loads the log. This gets displayed one line at a time. The content of each log entry appears under the field `message`. The second paragraph introduces a `match` section that contains a regular

[1] http://openwhisk.incubator.apache.org
[2] https://slack.com
[3] http://zeppelin.apache.org

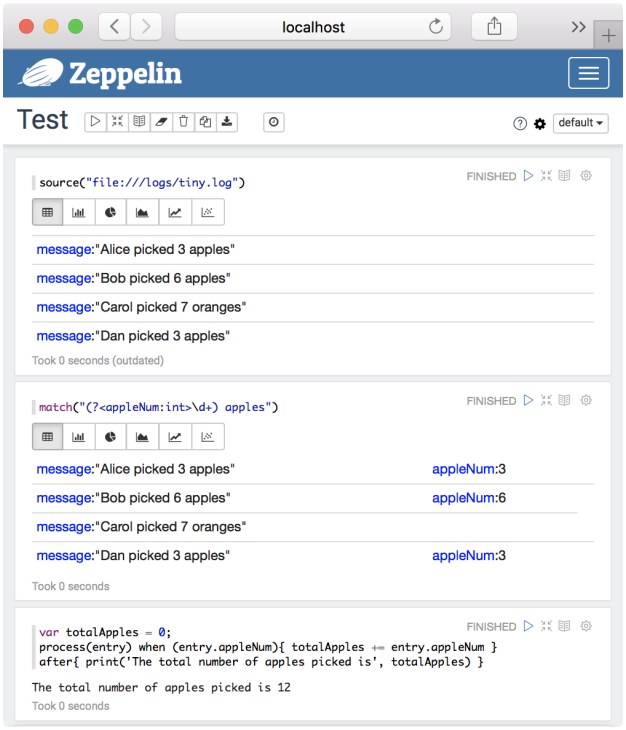

Figure 2. A CloudLens notebook.

expression and is used to extract how many apples were picked at each entry. The result of the second paragraph is a log that has been augmented with additional structure: some entries have a field `appleNum`. Notice that `Carol` picked oranges instead, so that line does not include this field. Not all entries need to have the same structure. Using `match`, CloudLens iteratively parses just enough data needed for a computation, and augments the existing data with more and more structure. Notice how each log entry gets transformed into an object with more and more fields iteratively. In general, these objects can contain hierarchical structures to represent nested streams and related groups of log entries.

The third paragraph in the notebook takes the augmented log as input to compute the total number of apples.[4] It introduces a variable `totalApples`. The section starting with `process(entry) when(entry.appleNum)` executes an action for each log line that has an `appleNum` attribute, using the variable name `entry` to refer to the line. For each entry, it aggregates the total number of apples by updating `totalApples`. The section `after` is executed when the processing is done for the entire (finite) log. It simply prints out the result. If we combine all the paragraphs into a single script and execute it, then parsing and computing are interleaved. The log is processed sequentially line by line. Every time a successful `match` happens, it creates the `appleNum` field which triggers the `process` section. In general, `process` can also add more fields to the log, which can trigger a chain of actions. Notice how we did not need an additional field to represent the total number of apples, which was simply represented as a state variable in the background. This feature helps to keep the parsed data separate from intermediate transient results, and helps to write analyses in a natural way. Section 3 gives a detailed description of the language.

CloudLens also has a command-line tool that supports batch processing as well as online monitoring. It is typically invoked with one CloudLens script and zero or more input log files. If no log file is provided, the tool reads from the standard input. UNIX pipes may be used to feed log streams into the tool from arbitrary sources. The running script can provide diagnostics on the standard output or in any other way, leveraging a rich ecosystem of JavaScript libraries. Code may be exported from the notebook for execution with the command-line tool, for instance for applying the implemented analyses to live sources.

3 The CloudLens Language

CloudLens is a stream programming language. Programs are executed on possibly infinite *streams* of log *entries* encoded as JSON objects. Log streams may originate from various sources including prerecorded log files or live sources such as TCP/IP sockets or Unix pipes. When a log stream is constructed from unstructured text, each text line becomes an entry in the log stream—a JSON object with a unique field `message` that contains the text of the line.

We now illustrate the different functionalities of CloudLens with a log file produced by Travis[5] for the Apache OpenWhisk project. Travis is a continuous integration service available to GitHub users. When a user modifies the source code of an application, Travis automatically starts a test suite to make sure that the new modifications do not cause a regression. OpenWhisk is an open source serverless cloud platform that executes functions in response to events. The example log file can be downloaded from the url: https://travis-ci.org/apache/incubator-openwhisk/builds/327525198

All the code fragments in this section are paragraphs of a notebook. They should be run in order as later paragraphs build upon log transformations from the earlier paragraphs. Together they form a log processing script comprising multiple analyses and diagnostics.

3.1 Match and Process

The first CloudLens script extracts the name of failed tests from `log.txt`. Tests are logged in the following format:

name > *description* `FAILED`

For instance, the log contains the line:

```
system.basic.WskBasicTests > Wsk Action CLI should reject delete of
      action that does not exist FAILED
```

[4] The input of a paragraph is the ouptut of the last executed paragraph. Re-running the same paragraph may return a different output.

[5] http://travis-ci.org

The CloudLens script and its output are shown below.

```
match("(?<failure>.*) > .* FAILED");
process(entry) when(entry.failure) {
  print("FAILED:", entry.failure);
}
```

```
FAILED: whisk.core.cli.test.WskBasicUsageTests
FAILED: system.basic.WskBasicTests
```

This script has two sections. A `match` section parses entries using regular expressions possibly augmenting entry objects with new fields. A `process` section executes actions written in JavaScript for specific entries. In this example, the `match` section identifies the log entries in the log stream with a `message` matching the regular expression `(?<failure>.*) > .* FAILED`. It adds to each matched entry object a field `failure` that contains the name of the failed test. The `process` section prints the names of the failed tests detected by the `match` section.

The `(?<failure>.*) > .* FAILED` expression uses a feature of regular expressions called a named capture group that makes it possible to identify by name a fragment of the match. Concretely it matches the same messages as the simpler expression `.* > .* FAILED` but, in addition, the substring matching the parenthesized subexpression is given the name `failure`. More generally, for each capture group `(?<` *ident*`>` *regex*`)`, the `match` section adds a field named *ident* to each entry object with a matching `message` and sets the field's value to the substring of `message` matching *regex*. For instance, the `match` section in our example mutates the log entry: `{message: "system.basic.WskBasicTests > ... FAILED"}` to `{message: "system.basic.WskBasicTests > ... FAILED", failure: "system.basic.WskBasicTests"}`.

The `process` section applies a function to selected log entries in the log stream. The function is specified as a block of JavaScript code which takes the log entry as its single argument. The parameter name of the function is specified in parentheses after the `process` keyword and the activation condition is specified in parentheses after the `when` keyword. In this example, the log entry is bound to the name `entry` and the function is to be executed only on log entries for which the field `failure` is defined. It is also possible to list multiple dependencies (e.g., `when(entry.a entry.b)`). In general, if no condition is specified, a rudimentary dependency analysis scans the section body for field accesses to ensures that a `process(entry)` section is executed only when all the fields on which it depends are defined in the `entry` being processed. In this example, the condition could have been omitted and we could write:

```
process(entry) { print("FAILED:", entry.failure); }
```

The following shorthand is also possible and preferred:

```
when(entry.failure) { print("FAILED:", entry.failure); }
```

In this form, the parameter name of the function is inferred from the condition.

While `process ... when` is a primitive construct, `match` is a built-in *lens*, i.e., a predefined CloudLens function. We discuss lenses in Section 3.6.

3.2 Variables

CloudLens is an *imperative* programming language. Scripts can mutate the log stream, like adding a `failure` field in the previous example, or declare and mutate variables. The `log.txt` file reports the beginning and the end of a test in the following format:

`Starting test` *description* `at` *date*
`Finished test` *description* `at` *date*

The next script identifies tests that lasted more than 30s.

```
match("Starting test (?<desc>.*) at (?<start:Date[yyyy-MM-
    dd' 'HH:mm:ss.SSS]>.*)");
match("Finished test (?<desc>.*) at (?<end:Date[yyyy-MM-dd
    ' 'HH:mm:ss.SSS]>.*)");
var start;
when(entry.start) { start = entry.start; }
when(entry.end) {
  entry.duration = entry.end - start;
  if(entry.duration > 30000) {
    print(entry.duration, "\t", entry.desc);
  }
}
```

```
53496    Wsk api should verify successful creation ...
61720    Whisk rules should not activate an action ...
68593    Wsk Sequence should execute a sequence in ...
60442    Wsk Action CLI should create, and invoke a...
```

This script leverages an extension of regular expressions to specify the expected type and format of a capture group. E.g., the field `start` has the type `Date` and the expected format in the log is `yyyy-MM-dd HH:mm:ss.SSS`.

The `start` variable is mutated every time the beginning of a test is detected, that is, when the `start` field is defined. At the end of the test, when the `end` field is defined, we add a `duration` field to the log entry. If this duration is greater than 30 seconds (30000ms), we print the description of the test `entry.desc`.

3.3 Finite Streams

CloudLens is designed to handle both finite and infinite streams. Live logs often embed finite sublogs that are key to application monitoring, e.g., session logs in a server log. The `after` section makes it possible to execute actions at the end of the log stream. For instance, we can count the number of failed tests as we detect them and report the final count.

```
var failed = 0;
when(entry.failure) { failed++; }
after { print(failed, "failed tests"); }
```

```
2 failed tests
```

Obviously the `after` section is never executed if the log is infinite. Like `process` sections, `after` sections are executed in program order. Conceptually, a special *EndOfStream* entry follows the last log entry, triggering `after` sections instead of `process` sections.

3.4 Return

By default, a section does not add, remove, or replace entries in the stream. It is possible to do so using `return` statements: `return e;` replaces the current entry with `e` whereas `return;` removes the current entry. A section may also insert multiple entries into the stream by returning an array. For instance, the following script prunes entries with empty messages, i.e., blank log lines, and appends a couple of log entries at the end of the log.

```
when(entry.message){if(entry.message.length == 0) return;}
after {
  return [{message: failed+" FAILED"}, {message: "END"}];
}
```

If no `return` statement gets executed, the entry being processed remains in the log, hence there is no need for an `else` branch in the `if` statement above.

3.5 Order of Execution

CloudLens is a *dataflow* language. A CloudLens script describes a pipeline of stages that each log entry has to go through. One log entry flows through the entire script before processing starts on the next entry. For each entry, sections are executed in program order. In particular, concatenating the scripts of Section 3.1 and 3.2 into a single paragraph of the notebook combines their analyses into a single processing pass, interleaving their outputs:

```
53496    Wsk api should verify successful creation ...
FAILED: whisk.core.cli.test.WskBasicUsageTests     ...
61720    Whisk rules should not activate an action ...
68593    Wsk Sequence should execute a sequence in ...
60442    Wsk Action CLI should create, and invoke a...
FAILED: system.basic.WskBasicTests
```

In other words, there is no implicit loop inside or around any CloudLens section, but rather an implicit source and sink of log entries around each notebook paragraph (cf. Section 4).

Variables are initialized on *first traversal*, i.e., the first time a log entry reaches the program point for the declaration. For instance, in the script of Section 3.3, the variable start is initialized once with the first log entry.

If desired, it is possible to serialize analyses on finite logs as illustrated in the next section.

3.6 Lenses

CloudLens scripts can encapsulate processing into *lenses*. Lenses are declared using the `lens` keyword followed by the name of the lens, the parameter list within parentheses, and the body within curly braces. The body of a lens has the same structure as a CloudLens script. For instance, the following lens makes it possible to implement multi-pass analyses by first buffering then replaying the (finite) stream.

```
lens rewind() {
  var stream = [];
  when(entry) { stream.push(entry); return; }
  after { return stream; } // replay the log
}
```

We can use the `rewind` lens to print the description of tests that took more than 9% of the total time.

```
var totalTime = 0;
when(entry.duration) { totalTime += entry.duration; }
after { print("Total Time:", totalTime/1000, "seconds"); }
rewind();
when(entry.duration entry.desc) {
  entry.prop = entry.duration * 100 / totalTime;
  if(entry.prop > 9) {
    print(entry.prop.toFixed(2) + "%", entry.desc);
  }
}
```

```
Total Time: 728.596 seconds
9.41% Wsk Sequence should execute a sequence in ...
```

In a first traversal of the log, we compute `totalTime` the total test time. We then traverse the log again to compute the proportion of time spent by each test.

Like `after` and `when` sections, lens invocations are not executed all at once on the entire log stream. Log entries flow one at a time through the script entering and leaving lens instances accordingly.

A variable declared inside a lens is scoped to the lens instance, e.g., in this example the `stream` variable is not visible outside of the `rewind` lens, but more importantly multiple instantiations of the lens get their own copy of the variable.

A lens can also be invoked inside a `when` section. It requires then an array as additional argument which is the stream to process by the lens. Finally, it is also possible to call a lens recursively.

3.7 Structured Logs

So far we have been processing `log.txt` one line at a time. But often logs have some structure. Modern logging frameworks produce structured log entries instead of simple text. Log messages may be split across multiple lines, etc. Navigating these structures is easy with CloudLens.

In our example log, Travis includes a stack trace for each failed test. Stack traces range over many lines but they logically belong to the most recent error message. Stack traces are recognizable by their non-zero indentation. To rebuild the logical structure of the log, we define a `group` lens that appends a log entry to the previous one if it matches a specific regular expression. More precisely, it combines the log entries matching the regular expression into a `group` array that gets embedded into the most recent unmatched entry.

```
lens group(regex) {
  var current = null;
  match("(?<partOfGroup>)" + regex);
  when(entry.partOfGroup) { // entry matches regex
    delete entry.partOfGroup; // remove helper tag
    if(current !== null) { current.group.push(entry); }
    return; // suppress entry
  }
  when(entry) { // remaining entries do not match regex
    var last = current;
    current = entry;
    current.group = [];
    return last;
  }
  after {
    if(current !== null) return current;
  }
}
```

A slightly more flexible version of this lens is included in the CloudLens library. Using the lens, we can look for indented log lines and build structured log entries.

```
group("^\\s");
```

The log entries of interest are now of the form:

```
{message: "error message",
 group: [{message: " stack trace element"},
         {message: " stack trace element"}, ...]}.
```

Theses traces however have a lot of noise. Suppose we wish to filter the traces to retain only the method calls associated with the OpenWhisk source code. We can define a `filter` lens and invoke it on the `group` array. For each failed test, the following CloudLens script prints a filtered stack trace.

```
lens filter() {
  match("at .*\((?<whisk>Wsk.*)\)");
  when(line.whisk) { print('  at', line.whisk); }
}
when(entry.failure) {
  print("FAILED", entry.failure);
  filter(entry.group);
}
```

```
FAILED whisk.core.cli.test.WskBasicUsageTests
  at WskBasicUsageTests.scala:1691
  at WskBasicUsageTests.scala:1688
  ...
FAILED system.basic.WskBasicTests
  at Wsk.scala:1046
  at Wsk.scala:714
  ...
```

In general, lenses can be invoked (1) at the toplevel or (2) inside of a `process` or `after` section. In the first case, as illustrated with the `group` invocation, the lens is implicitly applied to the log stream. In the second, the array argument must be specified explicitly at the end of the parameter list, as shown with the `filter` example. In both cases, the lens gets to process a stream of entries, either the log stream or the streamed array content.

4 Intuitive Semantics

We present here the intuition of the formal semantics of the CloudLens language. The complete semantics is presented in Appendix A.

To specify the evaluation of CloudLens scripts, we need to represent concomitantly the script and the log being processed. We do so by introducing the notion of a runtime script. A runtime script simply interleaves script sections and logs (i.e., sequences of log entries). Informally a log entry appears after section s_1 and before section s_2 if and only if it has already been processed by section s_1 and should be processed by s_2 next. Consider the following runtime script (ignoring the hollow dot $\circ$ and the filled dot $\bullet$ for now):

$$\bullet\cdot v_3\cdot v_2\texttt{process(e)\{f(e)\}}\ \circ\cdot v_1\texttt{process(e)\{g(e)\}}\ \texttt{process(e)\{h(e)\}}$$

This runtime script represents a point in the execution of the script

```
process(e){f(e)} process(e){g(e)} process(e){h(e)}
```

Specifically, it represents a point where entry v_1 is about to be processed by `g` whereas entries v_2 and v_3 are about to be processed by `f`. Logs are written right to left so as to match the intuition that the rightmost entry is ahead. In this example, v_2 comes before v_3 in the log. One can think of the logs in a runtime script as the content of the buffers that an implementation of CloudLens would maintain in-between pipeline stages.

The $\circ$ terminator indicates an empty log. For instance, the log $\circ\cdot v_1$ has a single entry v_1. The $\bullet$ terminator indicates the end of the log, i.e., the special log entry that triggers the execution of `after` sections. There is at most one $\bullet$ occurrence in a runtime script, and it can only be the leftmost symbol in the term since it will be the very last entry processed. There may be none when looking at a subscript of a larger script. Conversely, the $\circ$ terminator is not intended to be the leftmost term in a runtime script, but it may be when looking at a subscript of a larger script. Informally, $\bullet$ means "the end" whereas $\circ$ means "to be continued" since there are more entries to the left of $\circ$ (at the very least $\bullet$).

The processing of an element by a runtime script is defined by a relation of the following shape: $rsc\ /\ M \longrightarrow rsc'\ /\ M'$. This relation means that the runtime script rsc in the memory M reduces to rsc' in the memory M'. For example, the first rule of Figure 3 defines the behavior of a `process` section when there is a log entry (v) to process. If the condition evaluates to *true*, the body of the section is evaluated. We leverage JavaScript functions for proper variable scoping. Relations of the shape $js/M \Downarrow v/M'$ represent the JavaScript semantics. The evaluation of the body returns an array that is converted to a $\circ$-terminated log (operator $\odot$) which is transmitted to the next section of the script .

The second rule of Figure 3 defines the behavior of an `after` section when we reach the end of the log ($\bullet$). We

$$\frac{\texttt{(function (}x\texttt{) \{ return }c\texttt{ !== undefined \})(}v\texttt{)}/M \Downarrow \mathit{true}/M \qquad \texttt{(function (}x\texttt{) \{ }js\texttt{ \})(}v\texttt{)}/M \Downarrow a/M' \qquad \ell' = \circ \odot a}{\ell\cdot v\texttt{process(}x\texttt{) when(}c\texttt{) \{ }js\texttt{ \}}\ sc \,/\, M \longrightarrow \ell\texttt{process(}x\texttt{) when(}c\texttt{) \{ }js\texttt{ \}}\ \ell' sc \,/\, M'}$$

$$\frac{\texttt{(function () \{ }js\texttt{ \})()}/M \Downarrow a/M' \qquad \ell' = \bullet \odot a}{\bullet\texttt{after \{ }js\texttt{ \}}\ sc \,/\, M \longrightarrow \ell' sc \,/\, M'}$$

Figure 3. Some reduction rules

execute the body of the section. The section is removed from the runtime script. The returned array is converted to a •-terminated log that progresses to the next section.

Determinism, Progress The CloudLens semantics is deterministic. The chain of reductions is unique (assuming a deterministic JavaScript semantics): no two rules of the semantics can apply to the same term.

Because the evaluation of JavaScript expressions may get stuck, there is no guarantee of progress. The CloudLens semantics intends to guarantee progress for well-formed CloudLens scripts assuming no JavaScript evaluation gets stuck. Well-formedness basically requires lenses to be declared before use so that lens instantiation does not get stuck. The intuition for the proof is simple: the rules of the semantics cover all the possible runtime scripts that may be derived from $\bullet sc \,/\, M$. The proof of this result however requires carefully specifying not just well-formed scripts but also well-formed tuples $rsc \,/\, M$.

Infinite Logs While logs of arbitrary length can be trivially obtained by constructing large arrays, our inductive log definition does not permit infinite logs. Infinite logs could be introduced for instance by switching to a co-inductive interpretation. Alternatively, one could provide an infinite source of entries, e.g.,

$$\frac{\texttt{(function () \{ }js\texttt{ \})()}/M \Downarrow a/M' \qquad \ell = \circ \odot a}{\bullet\texttt{repeat \{ }js\texttt{ \}}\ sc \,/\, M \longrightarrow \bullet\texttt{repeat \{ }js\texttt{ \}}\ \ell sc \,/\, M'}$$

Termination Assuming that 1) the log is finite, 2) every JavaScript expression terminates and, 3) there is no recursive lens definitions, the chain of reductions from $\bullet sc \,/\, M$ is finite. We thus guarantee the termination of the script.

5 Implementation

Our implementation is open source.[6] It follows the semantics by implementing flows as iterators and sections as iterator transformers. The interpreter chains sections invocations according to the script source. Invoking `it.hasNext()` and `it.next()` on the last iterator `it` of the chain implements the semantics: pulling on the rightmost entry, while minimizing memory footprint, since there is no buffering of log entries.

The Zeppelin notebook framework is primarily implemented in Java. To facilitate integration we also implement CloudLens in Java. We use the Nashorn scripting engine[7] to evaluate JavaScript code in the Java Virtual Machine. For better performance, JavaScript code and regular expressions are compiled ahead of time using Nashorn and Java's regular expression library.

[6] https://github.com/cloudlens/cloudlens
[7] http://openjdk.java.net/projects/nashorn/

6 Case Study

In this section, we report on a detailed case study using CloudLens to analyze logs from Apache OpenWhisk: a distributed event-based programming service, which allows the user to execute code on the cloud in response to an event. It provides a serverless deployment environment hiding infrastructural complexity. The service is available via the command line, or through a graphical interface. When the user starts an action, a *transaction* gets initiated that goes through different steps, interacting with various services (e.g. a database). A controller guides transactions through their lifetime, logging different activities.

OpenWhisk is an industrial sized open-source project (65kLoC, about 100 contributors). We applied CloudLens to various aspects of OpenWhisk: to help developers get useful insights when they commit and build new revisions of the OpenWhisk codebase, as well as performance and usage analysis. OpenWhisk's logs are a good representative of the logs that need to be analyzed in the cloud. They are semi-structured, i.e., they contain some uniform data such as timestamps and transaction ids, but also a large part of unstructured messages. In our experience, cloud projects start by having unstructured logs and evolve to semi-structured logging to facilitate analysis. Even with better structure within the logs, analysis can be an arduous task with existing tools. CloudLens aims at facilitating this task.

6.1 Extracting Failures during Builds

OpenWhisk uses Travis, a continuous integration service available to GitHub users. When a user modifies the source code of an application, Travis automatically starts a test suite to make sure that the new modifications do not cause a regression. The build process generates large log files similar to `log.txt` used in Section 3 A lot of different types of information are logged and navigating the file manually is difficult. For example, to obtain information about failed tests, it does not suffice to search for the keywords "test" or "failed". This returns more than a hundred hits, many of which are not related to test results.

OpenWhisk developers built a bot powered by a CloudLens script similar to the one presented in Section 3. When a build runs to completion, the CloudLens script is automatically executed to extract information about test failures and filter stack traces. This data is then sent to the developer who initiated the build, via a Slack message. The bot was integrated in their build process, so no further action was required. We received positive feedback from developers on the ease-of-use and usefulness of this tool, compared to other industrial log analysis tools they tried. CloudLens made it easy to extract nested structures of unknown length, with just a few lines of code.

6.2 Transaction Duration

To analyze anomalies in performance, we extracted the duration of transactions to detect long-running ones. Matching log entries (such as start and end of a transaction) is typically not an easy task. The controller logs transaction activity, and for each event, it records a timestamp, a transaction id (tid), as well as markers indicating different stages. For example, the start of a transaction may be as follows:

```
[2017-10-16T09:48:34.948Z][INFO][#tid_59] GET /api/v1/namespaces/_/
    actions/shootcontroller
```

The final event for the same transaction is the following:

```
[2017-10-16T09:48:34.951Z][INFO][#tid_59][BasicHttpService][marker:
    http_get.200_count:3:3]
```

Log events from different transactions can appear interleaved, so our task is to match the start and end of the same transaction and compute the duration, printing it out, if it surpasses a limit. Below is a script that achieves this task:

```
var transactions = {};
var limit = 10000;
match("(?<tid>tid\_\d+)");
match("^\[(?<ts:Date[yyyy-MM-dd HH:mm:ss.SSS]>[^\]]+)\]");
process(e) when(e.tid) {
  transactions[e.tid] = transactions[e.tid] || [];
  transactions[e.tid].push(e);
}
after {
  for(tid in transactions) {
    var log = transactions[tid];
    var dur = log[log.length - 1].ts - log[0].ts;
    if(dur > limit) { print(tid, 'dur:', dur); }
  }
}
```

This scripts matches tids and timestamps, enters them in a data structure, and finally computes the duration of each transaction, printing out the ones that take longer than 10s. It can easily be extended to extract other information about the long-running transactions, such as action type.

This script can be adapted for online monitoring of infinite log streams. It cannot be used directly because the `after` section only gets triggered at the end of a finite stream. We can rewrite this script to compute durations on-the-fly provided that the final event of each transaction can be detected.

The following script discovers the shape of final events:

```
after {
  for(tid in transactions) {
      var log = transactions[tid];
      print(log[log.length - 1].message);
  }
}
```

```
[2017-10-16T09:48:22.330Z] [INFO] [#tid_9] [BasicHttpService] [
    marker:http_get.200_count:327:327]
[2017-10-16T09:48:22.961Z] [INFO] [#tid_11] [BasicHttpService] [
    marker:http_get.200_count:13:13]
[2017-10-16T09:48:23.154Z] [INFO] [#tid_13] [LoadBalancerService]
    received active ack for '7305cd5cccf8402285cd5cccf8e022a1'
[2017-10-16T09:48:24.065Z] [INFO] [#tid_15] [LoadBalancerService]
    received active ack for 'c2f1e91c9e4649c1b1e91c9e4639c123'
...
```

It appears that most transactions end with `http_get.200` or `received active ack`. This illustrates how the content of the logs can make it easier or harder to write a desired analysis. It would have been easier to have a single marker to end all transactions. In general, a developer may not have control over formats. We can still write a script to compute durations on-the-fly, and try to cover all cases that mark the end of a transaction:

```
var transactions = {};
var limit = 10000;
match("(?<tid>tid\_\d+)");
match("^\[(?<ts:Date[yyyy-MM-dd HH:mm:ss.SSS]>[^\]]+)\]");
match("(?<finishMarker>http_get.200)");
match("(?<finishMarker>received active ack)");
process(e) when(e.tid) {
  transactions[e.tid] = transactions[e.tid] || [];
  transactions[e.tid].push(e);
}
process(e) when(e.finishMarker) {
  var log = transactions[e.tid];
  var duration = log[log.length - 1].ts - log[0].ts;
  if(duration > limit) { print(e.tid, 'duration:',
     duration); }
  delete transactions[tid]; // to avoid a memory leak
}
```

The `finishMarker` triggers the second `process` section. This script computes the duration of transactions on-the-fly without requiring post-processing, so it is fully streaming. In general, `process` sections can be arbitrarily chained to trigger actions whenever a structure is discovered or computed.

Matching log entries (such as start and end of a transaction) is not an easy task. A combination of streaming logs and state greatly simplifies this task in CloudLens.

6.3 Database Operation Latency

OpenWhisk interacts with a database service, and we wrote a CloudLens script to create a histogram of database operation latency. This consists of identifying all matching start and finish log entries for each instance of every type of database operation, computing latencies, then aggregating results into a histogram per operation type. We used our Zeppelin notebook IDE to produce the histogram visualizations.

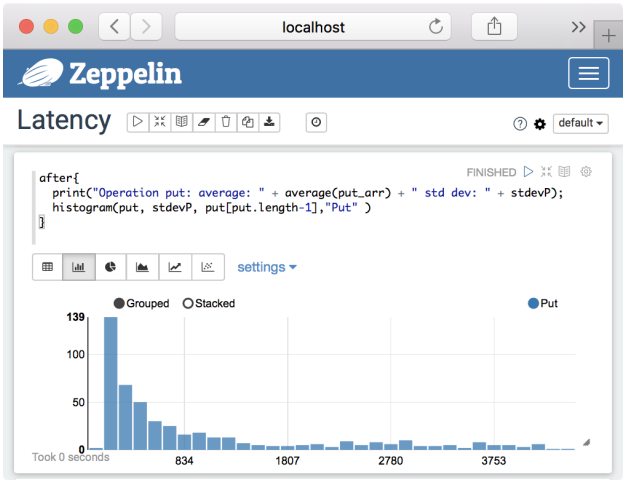

Figure 4. Database operation histogram.

Figure 4 shows a sample histogram for a database put operation. These are generated by printing data to the console with an appropriate format, and are built into Zeppelin, along with other visualizations. Our results showed outliers and pointed to a potential installation problem, which drove OpenWhisk developers to take corrective action.

CloudLens's Zeppelin IDE makes it easy for developers to interact with logs, try scripts, and visualize the results of their analyses.

7 Related Work

Traditionally, developers use grep or write ad-hoc scripts (e.g., in Perl or Python) for log analysis. To help with this arduous task, many techniques have been proposed: mining to automatically detect patterns and anomalies in logs, visualizations that help gain insights more effectively, monitoring systems that take actions in response to log patterns, as well as specialized query and programming languages.

There are many works on automated log mining either in isolation or in combination with other techniques such as source code analysis. Vaarandi describes a clustering algorithm for mining patterns in log files [20]. Stearly uses a pattern discovery algorithm originally developed for bioinformatics to mine syslogs [17]. Xu et al. apply log mining techniques [21, 22] to a large volume of logs from Google production systems [23]. They use static analysis of source code to extract all possible log printing statements, from which they generate regular expressions to parse log messages. Using machine learning techniques, they identify patterns over messages and can detect anomalies. Synoptic [5, 16] infers graph-based models of systems by mining existing logs. Statistical anomaly detection has its shortcomings: when a set of messages is detected to be an outlier, it is difficult to know if it truly constitutes an anomaly. Further investigation is necessary to validate the results. Our approach is complementary to these techniques in that we provide an expressive programming language to further explore logs and produce actionable insights.

Visualization techniques can help discover interesting patterns of information in logs (e.g., [9, 18, 19]) or complement log processing techniques to present results in more effective ways. ShiViz [1] visualizes a distributed log augmented with vector clocks to capture ordering dependencies. CSight [4] infers models in the form of communicating finite-state machines visualized graphically. These can help in program understanding and debugging. Visualization techniques and CloudLens are complementary and can be combined.

Swatch [12] is an automated system for monitoring and notification. It uses regular expressions to match against log entries, which can trigger actions such as executing a script. Logsurfer [15] is a rule-based tool for analyzing log files in real-time. Each rule specifies regular expressions for matching log messages. Upon a match an action is triggered that can evaluate an external script, or create/delete new rules. The rule-set can therefore change dynamically. In addition, Logsurfer provides the ability to group messages together via contexts, and triggering actions that take the context as input. Our approach generalizes from Swatch and Logsurfer in that it permits state that is carried over across actions. It also permits to interleave pattern matching and computations arbitrarily, enabling conditional pattern matching and gradual refinement.

Many languages can be used for log analysis, going back to the Awk programming language [2], which was designed for text processing and data extraction. It is a standard feature of most Unix-like operating systems. An Awk script consists of a series of rules with a pattern to match each line of the input text, and actions that are triggered when the pattern is satisfied. More recently, Sun Microsystems developed DTrace [10], a tracing framework inspired by Awk with minimal performance overhead for troubleshooting kernel and application problems. Our approach borrows ideas from Awk with some key differences. Like Awk, CloudLens has a special pattern for the end of the stream. In contrast with Awk, CloudLens views logs as streams of semi-structured objects. Moreover, the structure of these objects can be progressively refined over time. This approach facilitates building data in a hierarchical way, making it easier to express analyses that need to group or correlate different log entries. CloudLens scripts can mutate the stream so that later patterns and actions can be affected by earlier ones. In particular, scripts can restructure the log stream or buffer and replay it. Lenses can encapsulate not only helper functions and procedures but full-fledged log processing tasks.

CloudLens is inspired from dataflow languages [13]. A script defines a sequence of operations that each log entry has to go through. Dataflow languages typically permit these operations to be organized more freely into graphs

and worry not only about scheduling but also about routing. While the graph is elementary in CloudLens, the execution model remains the same: data tokens flow along the edges of the graphs from operation to operation according to a scheduling policy, which in CloudLens essentially amounts to a FIFO order. What CloudLens lacks in graph flexibility, it compensates with activation conditions adopting a reactive style of programming [3]. Operations can mutate the log entries in order to satisfy the activation conditions of subsequent operations and create chains of operations, but the order of activations for any given entry is prescribed by the program order. There are no back edges in the flow graph.

The PADS project is intended to "make it easier for data analysts to extract useful information from ad-hoc data files" [8]. PADS languages, e.g., PADS/ML, enable the declarative specification of data formats. Supporting tools can generate parsers, converters, checkers for particular formats. As with CloudLens, PADS is meant for imperfect data. Tools generated from PADS descriptions can be made tolerant to inconsistencies in the actual data. In contrast with CloudLens, however, PADS assumes that the data is mostly self-consistent and amenable to precise specification.

Sawzall [11] is a procedural domain specific language to specify queries on very large data sets. It is implemented on top of a map-reduce infrastructure and constrains expressivity to maximize parallelism. A Sawzall program specifies operations to be performed on each record and aggregates the resulting values. Sawzall's focus is on efficiently computing simpler queries. In contrast, our approach emphasizes expressivity and enforces determinism via a total order of execution. In principle, we could exploit pipeline parallelism at the expense of determinism as side effects could occur out of order.

Splunk [6] is a commercial tool for querying and visualizing textual data including logs. Queries in the Search Processing Language (SPL) allow pipelines of commands to search and transform data. Users can create new fields by parsing data with regular expressions entered in the query. Arbitrary scripts may be embedded and evaluated inside queries. CloudLens differs from SPL in that it is imperative: intermediate results of analyses can be held in state variables and do not need to be kept as artificial fields inside the records extracted from log entries. This feature helps to keep the parsed data separate from intermediate transient results, and helps to write analyses in a natural way.

8 Conclusion

In this paper, we presented CloudLens an imperative language for log processing that views logs as object streams. CloudLens does not require precise schemas, but enables localized parsing that can be freely interleaved with computation. CloudLens caters for both finite and infinite log processing, offline log discovery and online log monitoring. We focus here on expressivity and helping developers write interesting analyses easily and with a few lines of code. In the future, we plan to investigate performance, scalability and usability. As a first step, we re-implemented CloudLens as a Swift library.[8]

References

[1] J. Abrahamson, I. Beschastnikh, Y. Brun, and M. Ernst. 2014. Shedding Light on Distributed System Executions. In *ICSE'14*.

[2] A. Aho, B. Kernighan, and P. Weinberger. 1988. *The AWK Programming Language*. Addison-Wesley.

[3] E. Bainomugisha, A. L. Carreton, T. van Cutsem, Stijn M., and W. de Meuter. 2013. A Survey on Reactive Programming. *ACM Comput. Surv.* 45, 4, Article 52 (Aug. 2013), 34 pages.

[4] I. Beschastnikh, Y. Brun, M. Ernst, and A. Krishnamurthy. 2014. Inferring Models of Concurrent Systems from Logs of Their Behavior with CSight. In *ICSE'14*.

[5] I. Beschastnikh, Y. Brun, S. Schneider, M. Sloan, and M. Ernst. 2011. Leveraging Existing Instrumentation to Automatically Infer Invariant-Constrained Models. In *FSE'11*.

[6] D. Carasso. 2000. *Exploring Splunk, Search Processing Language (SPL) Primer and Cookbook*. CITO Research.

[7] L. Columbus. 2016. 2016 Big Data, Advanced Analytics & Cloud Developer Update: 5.4M Developers Now Building Cloud Apps. *Forbes* (Oct. 2016). http://www.forbes.com/sites/louiscolumbus/2016/10/16/2016-big-data-advanced-analytics-cloud-developer-update-5-4m-developers-now-building-cloud-apps/.

[8] K. Fisher and D. Walker. 2011. The PADS Project: An Overview. In *ICDT '11*. 11–17.

[9] L. Girardin and D. Brodbeck. 1998. A Visual Approach for Monitoring Logs. In *LISA'98*.

[10] B. Gregg and J. Mauro. 2011. *DTrace: Dynamic Tracing in Oracle Solaris, Mac OS X, and FreeBSD*. Prentice Hall Professional.

[11] R. Griesemer. 2008. Parallelism by Design: Data Analysis with Sawzall. In *CGO'08*.

[12] S. Hansen and T. Atkins. 1993. Automated System Monitoring and Notification with Swatch. In *LISA'93*.

[13] W. Johnston, J. R. P. Hanna, and R. Millar. 2004. Advances in Dataflow Programming Languages. *ACM Comput. Surv.* 36, 1 (March 2004), 1–34.

[14] B. Kernighan. 1978. *UNIX for Beginners*. Bell Laboratories.

[15] J. Prewett. 2003. Analyzing cluster log files using Logsurfer. In *ACLC'03*.

[16] S. Schneider, I. Beschastnikh, S. Chernyak, M. Ernst, and Y. Brun. 2010. Synoptic: Summarizing System Logs with Refinement. In *SLAML'10*.

[17] J. Stearley. 2004. Towards Informatic Analysis of Syslogs. In *Cluster'04*.

[18] T. Takada and H. Koike. 2002. Information Visualization System for Monitoring and Auditing Computer Logs. In *iV'02*.

[19] T. Takada and H. Koike. 2002. Mielog: A Highly Interactive Visual Log Browser Using Information Visualization and Statistical Analysis. In *LISA'02*.

[20] R. Vaarandi. 2003. A Data Clustering Algorithm for Mining Patterns from Event Logs. In *IPOM'03*.

[21] W. Xu, L. Huang, A., D. Patterson, and M. Jordan. 2009. Online System Problem Detection by Mining Patterns of Console Logs. In *ICDM'09*.

[22] W. Xu, L. Huang, A. Fox, D. Patterson, and M. Jordan. 2009. Detecting Large-Scale System Problems by Mining Console Logs. In *SOSP'09*.

[23] W. Xu, L. Huang, A. Fox, D. Patterson, and M. Jordan. 2010. Experience Mining Google's Production Logs. In *SLAML'10*.

[8] https://github.com/cloudlens/swift-cloudlens

A Formal Semantics

The semantics covers all the features discussed in Section 3 except for the syntactic shorthands—`process` without `when` and `when` without `process`—and the simple inference of the missing elements. To improve readability, we require that `process` and `after` sections always return arrays (with zero, one, or multiple elements) at no loss of expressivity, and assume that lenses have a single parameter. Generalizing to zero or multiple parameters is straightforward.

A.1 Syntax

The grammar of CloudLens is the following:

$$
\begin{array}{rcl}
sc & ::= & \emptyset \mid section\ \ sc \\
section & ::= & \texttt{var}\ x\ \texttt{=}\ js\texttt{;} \\
& \mid & \texttt{lens}\ f\texttt{(}x\texttt{)}\ \texttt{\{}\ sc\ \texttt{\}} \\
& \mid & \texttt{process(}x\texttt{)}\ \texttt{when(}path\texttt{)}\ \texttt{\{}\ js\ \texttt{\}} \\
& \mid & \texttt{after}\ \texttt{\{}\ js\ \texttt{\}} \\
& \mid & f\texttt{(}js\texttt{)} \\
path & ::= & x \mid path\texttt{.}x
\end{array}
$$

A script (sc) is a sequence of sections. There are three kinds of sections: declarations (`var` and `lens`), pipeline stages (`process` and `after`), and lens invocations ($f\,(js)$). The $\emptyset$ symbol denotes the end of the script. Variable initializers, actions (i.e., bodies of pipeline stages), arguments of lens invocations are all JavaScript expressions (js). Activation conditions in `process` sections are paths in JSON objects. These conditions can easily be extended to logic formulas over paths. Variable names and field names are identifiers (x).

Formally, a log (ℓ) is a right-to-left sequence of values v terminated by one of two terminators:

$$\ell \quad ::= \quad \circ \mid \bullet \mid \ell \cdot v$$

The $\circ$ terminator indicates an empty log, that is, the current log stream is empty but some more entries can come later. The $\bullet$ terminator indicates the end of the log. No more entries can come. In particulars, it triggers the execution of `after` sections.

We define the operator $\odot$ to append a JavaScript array to a log:

$$\ell \odot [v_1, \ldots, v_{n-1}, v_n] \quad = \quad \ell \cdot v_n \cdot v_{n-1} \cdot \ldots \cdot v_1$$

A runtime script (rsc) is a script during its execution. It contains both the program source and the logs to process by each sections. Formally, it is:

$$
\begin{array}{rcl}
rsc & ::= & \emptyset \\
& \mid & {}^{\ell}sc \\
& \mid & {}^{\ell}\texttt{process(}x\texttt{)}\ \texttt{when(}c\texttt{)}\ \texttt{\{}\ js\ \texttt{\}}\ rsc \\
& \mid & {}^{\ell}\texttt{after}\ \texttt{\{}\ js\ \texttt{\}}\ rsc
\end{array}
$$

This definition does not allow for all possible interleavings of logs and sections. Specifically, it forbids logs to appear to the right of a variable declaration, a lens declaration, or a lens invocation. Indeed, we will see below that the operational semantics erases declarations and inlines lenses as they get executed.

A.2 Semantics

JavaScript Semantics The semantics of JavaScript expressions is assumed and denoted by the following big step rule which means that the expression js evaluates in the memory M to the value v and the new memory state M':

$$js/M \Downarrow v/M'$$

CloudLens Semantics We formalize a small-step semantics for CloudLens scripts denoted by the relation:

$$rsc\ /\ M \longrightarrow rsc'\ /\ M'$$

This relation means that the runtime script rsc in the memory M reduces to rsc' in the memory M'. It is defined in Figures 5, 7 and 6 by means of inference rules. Figure 5 specifies reduction rules, i.e., how each kind of section processes one log entry. Figure 7 specifies context rules, i.e., the execution schedule. Figure 6 defines some auxiliary predicates. We will explain these rules later in the section.

Input log The semantics of the evaluation of a script sc on the empty input log in the memory M is given by the reflexive and transitive closure of the relation $\longrightarrow$ where the initial log is $\bullet$:

$$^{\bullet}sc\ /\ M \longrightarrow^{*} rsc\ /\ M'$$

We only need to consider such logs because scripts can generate log entries by themselves using combinations of `after` and `return`.

Reduction Rules We now discuss the rules in Figure 5. Declarations of variables and lenses are handled in a similar way. The name of the variable or the lens in the residue is substituted with a reference to a fresh memory location (α). This location contains for a variable the value obtained by evaluating the initializer. For a lens, the memory location contains a synthesized JavaScript function that implements the lens behavior and an attribute that contains the lens definition (the *packLens* predicate is defined Figure 6 (rule 1) and explained below). The common precondition $\ell \neq \circ$ indicates that the execution of a declaration is triggered when there is an entry to process ($\ell = \ell' \cdot v$) or if this is the end of the log ($\ell = \bullet$). The declarations are removed from the runtime scripts upon execution. They execute only once.

The first two rules for `process` evaluate the activation condition c given an entry to process v. If the condition evaluates to *true*, the body of the section is evaluated. We leverage JavaScript functions for proper variable scoping. The evaluation of the body returns an array that is converted to a $\circ$-terminated log which is transmitted to the next section of the script. If the condition evaluates to *false*, the entry advances untouched. The evaluation of a condition simply tests that the path mentioned in the condition is not undefined.

$$\frac{\ell \neq \circ \quad \alpha = \mathit{fresh}(M) \quad \mathit{js}[x \leftarrow \alpha]/M[\alpha \leftarrow \texttt{undefined}] \Downarrow v/M'}{\ell\,\texttt{var}\ x\ =\ \mathit{js};\ \ \mathit{sc}\ /\ M \longrightarrow \ell\,\mathit{sc}[x \leftarrow \alpha]\ /\ M'[\alpha \leftarrow v]}$$

$$\frac{\ell \neq \circ \quad \alpha = \mathit{fresh}(M) \quad \mathit{packLens}(\texttt{lens}(x)\ \{\ \mathit{sc}_f[f \leftarrow \alpha]\ \})/M[\alpha \leftarrow \texttt{undefined}] \Downarrow p/M'}{\ell\,\texttt{lens}\ f(x)\ \{\ \mathit{sc}_f\ \}\ \mathit{sc}\ /\ M \longrightarrow \ell\,\mathit{sc}[f \leftarrow \alpha]\ /\ M'[\alpha \leftarrow p]}$$

$$\frac{(\texttt{function}\ (x)\ \{\ \texttt{return}\ c\ \texttt{!==}\ \texttt{undefined}\ \})(v)/M \Downarrow \mathit{true}/M \quad (\texttt{function}\ (x)\ \{\ \mathit{js}\ \})(v)/M \Downarrow a/M' \quad \ell' = \circ \odot a}{\ell \cdot v\,\texttt{process}(x)\ \texttt{when}(c)\ \{\ \mathit{js}\ \}\ \mathit{sc}\ /\ M \longrightarrow \ell\,\texttt{process}(x)\ \texttt{when}(c)\ \{\ \mathit{js}\ \}\ \ell'\,\mathit{sc}\ /\ M'}$$

$$\frac{(\texttt{function}\ (x)\ \{\ \texttt{return}\ c\ \texttt{!==}\ \texttt{undefined}\ \})(v)/M \Downarrow \mathit{false}/M}{\ell \cdot v\,\texttt{process}(x)\ \texttt{when}(c)\ \{\ \mathit{js}\ \}\ \mathit{sc}\ /\ M \longrightarrow \ell\,\texttt{process}(x)\ \texttt{when}(c)\ \{\ \mathit{js}\ \}\ \circ \cdot v\,\mathit{sc}\ /\ M} \qquad \frac{}{\bullet\,\texttt{process}(x)\ \texttt{when}(c)\ \{\ \mathit{js}\ \}\ \mathit{sc}\ /\ M \longrightarrow \bullet\,\mathit{sc}\ /\ M}$$

$$\frac{}{\ell \cdot v\,\texttt{after}\ \{\ \mathit{js}\ \}\ \mathit{sc}\ /\ M \longrightarrow \ell\,\texttt{after}\ \{\ \mathit{js}\ \}\ \circ \cdot v\,\mathit{sc}\ /\ M} \qquad \frac{(\texttt{function}\ ()\ \{\ \mathit{js}\ \})()/M \Downarrow a/M' \quad \ell' = \bullet \odot a}{\bullet\,\texttt{after}\ \{\ \mathit{js}\ \}\ \mathit{sc}\ /\ M \longrightarrow \ell'\,\mathit{sc}\ /\ M'}$$

$$\frac{\ell \neq \circ \qquad M(f).\texttt{def} = \texttt{lens}(x)\ \{\ \mathit{sc}_f\ \} \qquad \mathit{instantiateLens}(\texttt{lens}(x)\ \{\ \mathit{sc}_f\ \}, \mathit{js})/M \Downarrow \mathit{sc}'_f/M'}{\ell\,f(\mathit{js})\ \ \mathit{sc}\ /\ M \longrightarrow \ell\,\mathit{sc}'_f\ \ \mathit{sc}\ /\ M'}$$

Figure 5. Reduction rules.

The last rule for `process` handles •. It simply removes the section from the runtime script as there are no entries left to process.

The next two rules define the behavior of `after`. The first of the two shows that log entries traverse `after` sections unchanged. The second rule executes the body of the section when the end of the log is reached. The section is removed from the runtime script and • progresses to the next section. The returned array is converted to a •-terminated log.

The last rule of Figure 5 takes care of lens invocations. As with declaration rules, the precondition $\ell \neq \circ$ ensures late instantiation.

Lens Declaration and Invocation Lenses can be invoked in two different contexts: either as a toplevel section of a CloudLens script or inside a JavaScript expression to execute the entire lens on the content of an array. In order to invoke a lens at toplevel, we need to access its definition. On the other hand in order to be invoked from JavaScript code, a lens must present itself as a JavaScript function. This function takes as arguments the argument of the lens and the array on which the lens must be applied (see for example the lens `filter`, Section 3.7). To be able to have this dual view of a lens, we use the ability in JavaScript to add fields to functions. It is the role of *packLens* in the rule for lens declarations (Figure 5). The predicate *packLens*(`lens(`x`) {` sc `}`) is defined in Figure 6. It creates the following function:

`function (x, a) {` *evalLens*(`lens(`x`) {` sc `}`, `x, a`) `}`

and adds to this function a field `def` that contains the lens definition: `lens(`x`) {` sc `}`.[9]

The predicate *evalLens* reifies the CloudLens semantics inside JavaScript giving the ability to evaluate a CloudLens script from JavaScript. The definition of *evalLens* is given in Figure 6 (rule 2). It instantiates the lens to gets its body, turns the JavaScript array a into a •-terminated log. Then it reduces the script on the log until all the elements are processed. The evaluation of a CloudLens script does not accumulate the processed log. The result of the evaluation is `null` but side effects on the memory are performed.

The predicate *instantiateLens* defined in Figure 6 (rule 3) creates a variable x local to the script sc whose initial value is the result of the evaluation of the expression js. The instantiation returns the body of the lens where the parameter has been instantiated.

Finally, we return to the lens invocation rule in Figure 5. This rule uses the `def` field of the lens to access its definition and the same *instantiateLens* predicate for instantiation. While a lens invocation in JavaScript evaluates the lens on the array to completion, i.e., on all the array elements at once, a lens invoked at toplevel in a CloudLens script is simply inlined into the surrounding script. The inlined sections are evaluated like any other.

Context Rules We now discuss the rules in Figure 7. The first two rules state that in a runtime script with multiple log terms, the execution proceeds with the rightmost log first. In other words, they define the execution schedule. Combined with the rules in Figure 5 they specify that the most advanced entry in a runtime script—rightmost entry—is the first one to advance.

The last three rules purge useless rightmost log terms from a runtime script so that the execution can move to the next log to the left. First, if the rightmost log is empty, it can be erased. Second, entries that have been through the whole script can be erased.

[9] `lens(`x`) {` sc `}` denotes a lens encoded as a JavaScript object. It does not matter if it is a string, an abstract syntax tree, etc, as long as there exists a way to evaluate the encoded lens.

$$\frac{f = \texttt{function f(x, a) \{ } \mathit{evalLens}(\texttt{lens}(x)\,\{\,sc\,\}, \texttt{x}, \texttt{a}) \texttt{ \}} \qquad \texttt{(function (}x\texttt{) \{ }f\texttt{; f.def = lens(}x\texttt{)\{}\,sc\,\texttt{\}; return f \})()}/M \Downarrow p/M'}{\mathit{packLens}(\texttt{lens}(x)\,\{\,sc\,\})/M \Downarrow p/M'}$$

$$\frac{\mathit{instantiateLens}(\texttt{lens}(x)\,\{\,sc\,\}, js)/M \Downarrow sc_f/M_f \qquad \ell = \bullet \odot a \qquad {}^{\ell}sc_f \;/\; M_f \longrightarrow^{*} \emptyset \;/\; M'}{\mathit{evalLens}(\texttt{lens}(x)\,\{\,sc\,\}, js, a)/M \Downarrow \texttt{null}/M'}$$

$$\frac{js/M \Downarrow v/M' \qquad \alpha = \mathit{fresh}(M')}{\mathit{instantiateLens}(\texttt{lens}(x)\,\{\,sc\,\}, js)/M \Downarrow sc[x \leftarrow \alpha]/M'[\alpha \leftarrow v]}$$

Figure 6. Auxiliary predicates.

$$\frac{rsc \;/\; M \longrightarrow rsc' \;/\; M'}{{}^{\ell}\texttt{process}(x)\ \texttt{when}(c)\ \texttt{\{}\ js\ \texttt{\}}\ rsc \;/\; M \longrightarrow {}^{\ell}\texttt{process}(x)\ \texttt{when}(c)\ \texttt{\{}\ js\ \texttt{\}}\ rsc' \;/\; M'} \qquad \frac{rsc \;/\; M \longrightarrow rsc' \;/\; M'}{{}^{\ell}\texttt{after\ \{}\ js\ \texttt{\}}\ rsc \;/\; M \longrightarrow {}^{\ell}\texttt{after\ \{}\ js\ \texttt{\}}\ rsc' \;/\; M'}$$

$$\frac{}{{}^{\ell}\texttt{process}(x)\ \texttt{when}(c)\ \texttt{\{}\ js\ \texttt{\}} \circ \mathit{section}\ \ sc \;/\; M \longrightarrow {}^{\ell}\texttt{process}(x)\ \texttt{when}(c)\ \texttt{\{}\ js\ \texttt{\}}\ \ \mathit{section}\ \ sc \;/\; M'}$$

$$\frac{}{{}^{\ell}\texttt{after\ \{}\ js\ \texttt{\}} \circ \mathit{section}\ \ sc \;/\; M \longrightarrow {}^{\ell}\texttt{after\ \{}\ js\ \texttt{\}}\ \ \mathit{section}\ \ sc \;/\; M'} \qquad \frac{}{{}^{\ell}\emptyset \;/\; M \longrightarrow \emptyset \;/\; M}$$

Figure 7. Context rules.