Seaview: Using Fine-Grained Type Inference to Aid Log File Analysis

Sudheendra Hangal
Computer Science Department
Stanford University
\{hangal\}@cs.stanford.edu

ABSTRACT
Log files contain a lot of valuable information for debug and monitoring. However, sense-making using them is a cumbersome task because they are typically stored and interpreted as plain text. We propose a mechanism to restore some semantics to log files, by performing static analysis on a Java program to automatically infer fine-grained dimensional information for the values being logged \cite{5}. These dimensions can be program-specific, such as “port number”, “number of pixels”, or “name of a file”, and form aspects of a program along which a log file can be usefully queried and visualized. We have developed a program called Seaview that explores these ideas and provides an initial implementation. Seaview can automatically generate visualizations appropriate for different types of values, as well as add a dash of color to log files.

Author Keywords
Log files, units and dimensions, debugging, visualization

ACM Classification Keywords
H.5.2 Information Interfaces and Presentation: Miscellaneous—Optional sub-category

INTRODUCTION
Programs generate a tremendous amount of information while they are running. Of this information, programmer-inserted log statements are especially valuable, because they are created with the explicit goal of providing insight into the behavior of the program at runtime. Logs are extremely useful for trouble-shooting, spotting problems early, detecting sub-optimal or anomalous behavior, post-hoc analysis, and even just to aid humans in understanding the program. The popularity of logging libraries such as Apache commons logging, Log4j and java.util.logging proves the popularity and pervasiveness of logging.

However, the potential of log files is frequently under-utilized, and logs are often consulted only when there is evidence of a problem and no other information is available. In production, most programs are configured to run at info or warning log levels; logging at finer levels is turned off because it is too voluminous to examine manually, and swamps what may be more important alerts. This author has tried to diligently adopt and maintain logging infrastructures for several of his programs, but has found the overall value of logging to be modest, primarily because of the difficulty of consulting log files and making sense of them.

The main idea in this paper is to exploit the fact that logs are generated by computer programs with well-defined semantics. Log files need not (and should not) be viewed simply as a sea of opaque text; instead, the structure of the program can help illuminate the data present in them. In both a literal and metaphorical sense, we would like to add more ‘color’ to the logs. This observation has also been made independently by others, and some techniques are being developed to exploit it, as we discuss in the related work section.

When necessary, people invest time in writing log post-processing scripts, and coupling them with specific visualizations such as charts and dashboards. However, this is a largely manual task that we would like to automate at least in part. Our goal is to enable convenient visualization of the logs from any program with almost no effort, as well as a more systematic way for the user to explore the space of effective visualizations.

We propose fine-grained type inference as one enabler for the generation of log file visualizations. Such inference can generate detailed information about the values contained in the logs. For example, we may be able to infer that a specific log statement emits values of the type “Student ID”, or of type “name of a file”. Note that these types are more specific than their nominal language types such as int and String. Further for numeric types, we may be able to deduce that IDs have the property of being ordinal, but not quantitative (i.e. they can be compared with the less than operator, but it probably does not make sense to add 2 IDs to generate a third one). Such information can possibly let us choose appropriate visual representations automatically.

In this paper, we explore this overall idea and present Seaview, a preliminary tool that may illustrate the efficacy and limitations of our approach. This work has so far not been evaluated across multiple programs or users, but has been guided by the features in a specific visualization created by the author \footnote{http://suif.stanford.edu/hangal/groups/together.html is an example visualization that took the author several days to create manually}.

USE CASES
We expect two primary categories of users for our eventual system.

\footnote{http://suif.stanford.edu/hangal/groups/together.html is an example visualization that took the author several days to create manually}
The first is a programmer who is debugging a program that he or she has written, or is in charge of maintaining. The programmer would use the tool for one of the following purposes: (a) Understanding the program’s behavior when it is operating correctly. (b) Performing quick sanity checks to make sure logged values have expected characteristics (c) Visually comparing two versions of a program, for example two different implementations of the same algorithm, or the same program before and after making a change (d) Potentially identifying a bug when a program does something wrong or anomalous.

A second type of user is an operator who is not familiar with the program but is in charge of running it, and would like to (a) detect potential trouble spots early, or (b) perform some rudimentary diagnosis when something goes wrong and explore workarounds (e.g. what are the port numbers being used? Are the student ID numbers being handled outside the usual range?). Such diagnosis is very valuable because it is much quicker than referring the problem and log files back to a programmer who may not be easily accessible.

UNIFI
We use the UniFi program to analyze a Java program and provide information about types of values [5]. UniFi infers fine-grained types, much like physical dimensions, for primitive type variables and strings. Assuming the program is correct from a dimensional perspective, UniFi can infer and solve constraints between the dimensions of different variables. The entire set of rules is described in the cited paper, but here are a few examples. The fragments

\[ a = b + c; \]
\[ y < z; \]
\[ \text{arr}[i]; \]

imply that \( a \) must have the same dimensions as \( b \) and \( c \); \( y \) must have the same dimensions as \( z \); and \( i \) must have the same dimensions as \( \text{arr}.\text{length} \).

Multiplication and division constraints are more complicated, since they compose dimensions using arithmetic on the exponents of the base dimensions. It turns out that a Gaussian elimination style solver can resolve these constraints. As a simple example, given the program

\[ a = b \times c; \]
\[ d = a / b; \]

the solver can infer that \( d \) must have the same dimension as \( c \).

The analysis also works across procedure boundaries and can handle calls to polymorphic methods that operate on more than one type. For example:

```java
int doubleIt(int x) { return 2*x; }
... doubleIt(seconds);
doubleIt(pixels);
```

does not constrain \( \text{seconds} \) and \( \text{pixels} \) to have the same dimension.
UniFi has thus far been used to detect likely dimensionality errors in programs, but using it’s inferred types for illuminating log files is a novel and promising application.

**SYSTEM DESCRIPTION**

In Seaview, we first generate dimensional types for each program using UniFi. We built an instrumenter that analyzes the bytecode of class files and identifies log statements that call Apache commons or java.util.logging libraries. (Log4j is frequently used through the Apache commons library, but it should be straightforward to support it directly as well.) Given a log statement, we scan the bytecode backwards for calls that construct the string to log (there is necessarily a bit of pattern matching here, but it should be fairly robust since there are usually limited ways in which compilers append different types of values when constructing a string.)

If a value being used to construct the logged string is the result of a method call, a local variable, a field, we query UniFi for its dimension type, and insert a call to the Seaview runtime with the value and its associated dimension. Each enum class is assumed to have its own type. Information associated with that dimension, (such as how many different variables are associated with it, descriptive name, full name, etc) are stored separately and used later during visualization.

When the instrumented version of the program is run, the Seaview runtime receives and saves the logged values, along with their types. These values can be visualized using the Seaview interface, a screenshot of which is shown in Fig. 1. The user can select a particular dimension, or a particular code location emitting a value within that dimension. Currently the only visualization supported is a bar graph that is dynamically updated as the user selects and unselects dimensions or code locations. If values from different code locations are being displayed together, colors are used to differentiate between them. (Fig. 1 shows only one color because only one source line is selected.) If source code is available, it is displayed below the chart so the user can browse the code context. Hovering on a bar in the chart shows additional information about the value.

Optionally, Seaview can also alter the logged strings to embed dimensionality information in them. A post-processing script converts the log file to HTML with values of different dimensions encoded with different colors, thus creating a simple, colored log file that can be quickly skimmed for values of a particular type (See Fig. 2).
RELATED WORK
There is a good deal of recent interest in enabling better analysis of log files.

Splunk provides tools for log-file analysis across multiple machines in a data center [1]. It indexes log files, and allows the user to perform sophisticated queries, visualization and reporting on the logs. Splunk is aware of generic types in log files, such as IP addresses and date stamps. It relies on the programmer inserting specific syntax to recognize semantic types in the program (e.g. name=value in the log output gets Splunk to recognize that name is a semantic type). While Splunk has some ability to extract fields from textual patterns, it does not truly know which fields are semantically related, so it cannot, for example, associate the type across different log statements. Our approach is able to infer such semantics, and moreover can extract properties about the type from the program, such as whether it is natural suited to an ordinal, nominal or quantitative representation. Our work is therefore complementary to Splunk, since the information can be used by tools like Splunk for better analysis and visualization.

Historically, there have been many approaches attempting to infer patterns and anomalies in log files. There have also been attempts to reverse engineer some structure from log files, for example by the identification of “message types” which categorize log messages and break them up into fixed and variable parts. (I am yet to identify the best refs to cite here, but there are several of them.)

Recently, there seems to be growing interest in making better use of log files by taking the original program into account. Sherlog is a system that analyzes log files to infer paths the program must have taken based on the log statements, and offers programmers a way to potentially query variables along the path [10]. LogEnhancer takes a step towards improve the quality of logging by inferring what variables should be logged [11]. Both of these seem to be good ideas, leading to useful systems. An unnamed system by Xu et al extracts structure from logs, and performs Principal Component Analysis to detect likely anomalies [9]. This paper notes the importance of logged sequences of state variables, a particular type of categorical variable. It also attempts to group values logged from different locations by variable names, a crude form of type identification. SLCT is another system that uses techniques based on association rule mining to detect when anomalous sets of values are logged together [8]. Somewhat surprisingly, none of these systems lay a particular emphasis on visual debugging and monitoring.

Other examples of systems that attempt to perform some kind of querying and analysis of log files are Chainsaw [2], Logstash [3], PADS [4], work done by Nagappan et al [6] and by Oliner et al [7].

Ravi Parikh’s Data Profiler may possibly be relevant to this work in future as well.

CONCLUSION
The Seaview approach appears to be promising because, with a few clicks, it could create a rudimentary version of the visualization that took the author several days to create.

REFERENCES