From Dirt to Shovels: Fully Automatic Tool Generation from Ad Hoc Data

Kathleen Fisher
AT&T Labs Research
kfisher@research.att.com

David Walker, Kenny Q. Zhu
Princeton University
dpw,kzhu@CS.Princeton.EDU

Peter White
Galois Connections
peter@galois.com

Abstract

An *ad hoc data source* is any semistructured data source for which useful data analysis and transformation tools are not readily available. Such data must be queried, transformed and displayed by systems administrators, computational biologists, financial analysts and hosts of others on a regular basis. In this paper, we demonstrate that it is possible to generate a suite of useful data processing tools, including a semi-structured query engine, several format converters, a statistical analyzer and data visualization routines directly from the ad hoc data itself, without any human intervention. The key technical contribution of the work is a multi-phase algorithm that automatically infers the structure of an ad hoc data source and produces a format specification in the PADS data description language. Programmers wishing to implement custom data analysis tools can use such descriptions to generate printing and parsing libraries for the data. Alternatively, our software infrastructure will push these descriptions through the PADS compiler, creating format-dependent modules that, when linked with format-independent algorithms for analysis and transformation, result in fully functional tools. We evaluate the performance of our inference algorithm, showing it scales linearly in the size of the training data — completing in seconds, as opposed to the hours or days it takes to write a description by hand. We also evaluate the correctness of the algorithm, demonstrating that generating accurate descriptions often requires less than 5% of the available data.

1. Introduction

An *ad hoc data source* is any semistructured data source for which useful data analysis and transformation tools are not readily available. XML, HTML and CSV are *not* ad hoc data sources as there are numerous programming libraries, query languages, manuals and other resources dedicated to helping analysts manipulate data in these formats. However, despite the prevalence of standard formats, massive quantities of legacy ad hoc data persist in fields ranging from computational biology to finance to physics to networking to health care and systems administration. Moreover, engineers and scientists are continuously producing new ad hoc formats —despite the presence of existing standards— because it is often expedient to do so. Over time, these expedient formats become difficult to work with because of missing documentation, a lack of tools, and corruption caused by repeated, poorly thought-through redesign, reuse and extension.

The goal of the PADS project [5, 6, 14, 16] is to improve the productivity of data analysts who need to cope with new and evolving ad hoc data sources on a daily basis. Our central technology is a domain-specific language in which programmers can specify the structure and expected properties of ad hoc data sources, whether they be ASCII, binary, Cobol or a mixture of formats. These specifications, which resemble extended type declarations from conventional programming languages, are compiled into a suite of programming libraries, such as parsers and printers, which are then linked to generic data processing tools including an XML-translator, a query engine [4], a simple statistical analysis tool, and others. Hence, the most important benefit of using PADS is that a single declarative description may be used to generate many useful end-to-end data processing tools completely automatically.

On the other hand, the most important impediment to using PADS is the time and expertise needed to write a PADS description for a new ad hoc data source. For data experts possessing clear, unambiguous documentation about a simple data source, writing a PADS description may take anywhere from a few minutes to a few hours. However, it is relatively common to encounter ad hoc data sources that contain valuable information, yet have little or no documentation. Understanding the structure of the data and creating descriptions for such sources can easily take days or weeks depending upon the complexity and volume of the data in question. In one specific example, Fisher spent approximately three weeks (off and on) attempting to understand and describe an important data source used internally at AT&T. One of the stumbling points in this case was that the data source suddenly switched formats after approximately 1.5 million entries. Of course, if dealing with the vagaries of ad hoc data sources is time-consuming and error-prone for experts, it is even worse for novice users.

To improve the productivity of experts and to make the PADS toolkit accessible to new users with little time to learn the specification language, we have developed an automatic format inference engine. This format inference engine reads arbitrary ASCII data sources and produces an accurate, human-readable PADS description of the source. These machine-produced descriptions give experts a running start in any data analysis task as the libraries generated from these descriptions may be incorporated directly into an ordinary C program. The inference engine is also directly connected to the rest of the PADS infrastructure, making it possible for first-time users, with no knowledge of the PADS domain-specific language, to translate data into a form suitable for loading into a relational database, to load it into an Excel spreadsheet, to convert the data into XML, to query it in XQuery, to detect errors in additional data from the same source, and to draw graphs of various data components, all with just a "push of a button."

To summarize, this paper makes three main contributions.

- We have developed a new, multi-phase algorithm that infers the format of complex, ad hoc data sources, producing compact and accurate PADS descriptions.
- We have incorporated the inference algorithm into a modular software system that uses sample data to generate a toolkit of useful data processing tools, without requiring any human intervention.

- We have evaluated the correctness and performance of our system on a range of ASCII data sources. For many data sources, training on 5% or less of the data results in accuracy rates greater than 95% (often perfect). In *all cases*, additional training data elevates accuracy rates above 95% and in *no cases* need a user ever be unsure about the accuracy rate of generated descriptions – the automatically generated accumulator tool measures both overall accuracy and field-by-field accuracy of the description on any new data source. In all cases, after fixing a data source, the inference algorithm scales linearly with the quantity of data.

For readers interested in seeing our system operate live, we are currently creating an online demo to illustrate its many features (`http://www.padsproj.org`).

In the next section of this paper, we describe the internal representation used during the course of the inference algorithm. For those readers familiar with the PADS description language, this is largely a review. Section 3 describes our format inference algorithm in depth and illustrates its action on two sample data sources. Section 4 evaluates the performance and correctness of our system on 15 different ad hoc data sources, drawn mostly from systems and networking domains. Section 5 discusses how users can deal with errors in generated descriptions and points out weaknesses we plan to address in future work. Sections 6 and 7 present related work and conclude respectively.

2. The Internal Format Description Language

Our format inference algorithm is implemented as a series of phases that generate and transform an internal format description language we refer to simply as the IR. The IR is very similar to the IPADS language we developed and formalized in previous work [6]. Apart from syntax, the main differences are that the IR omits recursion and function declarations; the former being beyond the scope of our current inference techniques and the latter being unnecessary during the course of the inference algorithm.

2.1 The Language

Like all languages in the PADS family, the IR is a collection of type definitions. These "types" define both the external syntax of data formatted on disk and the shape of the internal representations that result from parsing. We rely upon both of these aspects of type definitions to generate stand-alone tools automatically. Figure 1 summarizes the syntax of the IR and of the generated internal representations.

The building blocks of any IR data description are the base types b, which may be parameterized by some number of arguments p. Arguments may either be constants c, which include characters a, integers i and strings s, or variables x bound earlier in the description. These base types include a wide range of different sorts of integers and strings. In its initial phases, the inference algorithm uses general integer `Pint`, alphanumeric string `Palpha` and punctuation character `Pother(`a`)` types. In later phases, these coarse-grained base types are analyzed, merged and refined, producing integers with ranges `PintRanged(min,max)`, integers with known size `Pint32` or `Pint64`, constant integers (`PintConst(`i`)` for some integer i), or floating-point numbers `Pfloat`. Likewise, later stages of our algorithm transform alphanumeric strings into arbitrary strings with terminating characters (`Pstring(`a`)` where a terminates the string), fixed width strings (`PstringFW(`i`)` where i is the length of the string) or string constants `PstringConst(`s`)`. For brevity in our descriptions, we normally just write the constant string s inline in a description instead of `PstringConst(`s`)`.

c ::= $a \mid i \mid s$ (constants)
x (variables)
p ::= $c \mid x$ (parameters)

Base types b ::=

`Pint`	(generic, unrefined integer)
`PintRanged`	(integer with min/max values)
`Pint32`	(32-bit integer)
`Pint64`	(64-bit integer)
`PintConst`	(constant integer)
`Pfloat`	(floating point number)
`Palpha`	(alpha-numeric string)
`Pstring`	(string; terminating character)
`PstringFW`	(string; fixed width)
`PstringConst`	(constant string)
`Pother`	(punctuation character)
`ComplexB`	(complex base type defined by regexp; *e.g.* date, time, *etc.*)
`Pvoid`	(parses no characters; fails immediately)
`Pempty`	(parses no characters; succeeds immediately)

Types T ::=

$b(p_1,...,p_k)$	(parameterized base type)
$x{:}b(p_1,...,p_k)$	(parameterized base type; underlying value named x)
`struct` $\{T_1; ... T_k;\}$	(fixed sequence of items)
`array` $\{T;\}$	(array with unbounded repetitions)
`arrayFW` $\{T;\}[p]$	(array; fixed length)
`arrayST` $\{T;\}$[sep,term]	(array; separator and terminator)
`union` $\{T_1; ... T_k;\}$	(alternatives)
`enum` $\{c_1; ... c_k;\}$	(enumeration of constants)
x:`enum` $\{c_1; ... c_k;\}$	(enumeration of constants; underlying value named x)
`option` $\{T;\}$	(type T or nothing)
`switch` x `of` $\{c_1 \Rightarrow T_1; \ldots c_k \Rightarrow T_k;\}$	(dependent choice)

Representations of parsed data d ::=

c	(constant)
$\text{in}_i(d)$	(injection into the i^{th} alternative of a union)
$(d_1,\ldots,d_k)$	(sequence of data items)

Figure 1. Selected elements of the IR.

In addition to these simple base types, the IR includes a collection of higher-level base types commonly found in ad hoc data, specified generally in Figure 1 as `ComplexB`. For example, we have implemented base types for IP addresses, email addresses, URLs, XML tags, dates, times and a variety of others. Finally, the types `Pvoid` and `Pempty` are two special base types that are introduced at various points in the inference process. The first fails immediately; the second succeeds immediately. Neither consumes any characters while parsing.

Complex descriptions are built from simpler ones using a variety of type constructors. Type constructors include basic struct types `struct`$\{T_1; ... T_k;\}$, which indicate a data source should contain a sequence of items matching T_1, ..., T_k, basic array types `array` T, which indicate a data source should contain a sequence of items of arbitrary length, each matching T, and union types `union` $\{T_1; ... T_k;\}$, which indicate a data source should match one of T_1, ..., T_k. Once again, initial phases of the inference algorithm restrict themselves to one of these three sorts of type constructors. However, later phases of the algorithm refine, merge and process these simple types in a variety of ways. For example, unions may be transformed into enumerations of constants `enum` $\{c_1; ... c_k;\}$ or options `option` $\{T;\}$. In addition, later phases of the algorithm bind variables to the results of parsing base types and enums. For example, $x{:}b(p_1,...,p_k)$ expresses the fact that variable x is bound to the value parsed by base type $b(p_1,...,p_k)$. These variables express dependencies between different parts of a descrip-

Crashreporter.log:

```
Sat Jun 24 06:38:46 2006 crashdump[2164]: Started writing crash report to: /Logs/Crash/Exit/ pro.crash.log
- crashreporterd[120]: mach_msg() reply failed: (ipc/send) invalid destination port
```

Sirius AT&T Phone Provisioning Data:

```
8152272|8152272|1|6505551212|6505551212|0|0||no_ii152272|EKRS_6|0|FRED1|DUO|10|1000295291
8152261|8152261|1|0|0|0|0||no_ii752261|EKRS_1|0|kfeosf2|DUO|EKRS_6|1001390400|EKRS_OS_10|1001476801
```

Figure 2. Example ad hoc data sources.

tion.[1] For example, the length of a string `PstringFW`(p) or an array `ParrayFW`(p) may depend upon either a constant or a variable and likewise for any other parameterized base type. In addition, unions may be refined into dependent switch statements `switch` x `of` $\{c_1$ => T_1; ... c_k => T_k;$\}$, where the data is described by T_1, ..., or T_k depending on the value associated with x, be it c_1, ..., or c_k.

The result of parsing according to a description is an internal representation of the data. We let metavariable d range over such data representations. For the purposes of this paper, a data representation may be a constant c, an injection into the i^{th} variant of a union $\mathtt{in}_i(d)$, or a sequence of data representations $(d_1, \ldots, d_k)$. The injections are used as the representations of any sort of union type, be it a union, an enumeration, an option or a switch. The sequences are used as the representations of any sort of sequence type, whether it be a struct or one of the array variants.

2.2 Running Examples

Figure 2 presents tiny fragments of two different ad hoc data files on which we have trained our inference algorithm. The first, Crashreporter.log, is a Macintosh system file that records information concerning process crashes.[2] The second, which we call Sirius, is an internal AT&T format used to record phone call provisioning information. We use the Crashreporter.log data source as our main example throughout the paper; periodically we refer to the Sirius data source to illustrate particular aspects of the inference algorithm.

Figure 3 presents a hand-written description of the Crashreporter.log file in the IR syntax. This description is most easily read from the bottom, starting with the definition of the `source` type. This definition specifies that the data source is an array of structs separated by newline characters and terminated by the end of file marker. In other words, the data source is a sequence of lines, with the struct in question appearing on each line. The struct itself indicates each line is a sequence of `dateoption`, `kind`, `dumpid` and `report` fields. The description also specifies that the delimiter `"["` appears between the `kind` and `dumpid` fields, and the delimiter `"]: "` appears between the `dumpid` and `report` fields.

Most of the variable names associated with fields (*e.g.* `date`, `dumpid`, *etc.*) merely serve as documentation for the reader. However, the `kind` field is different – it is used later in the description and hence illustrates a *dependency*. To be specific, the form of the `report` field depends upon the contents of the `kind` field. If its value is `"crashdump"`, then the `report` is a `dumpReport` type, while if the `kind` field is `"crashreporterd"`, the `report` is a `reporterReport` type.

[1] For the purposes of inference, every bound variable is assumed to be distinct from every other that appears in a description. Roughly speaking, the scope of such variables extends as far as possible to the right through the description. Understanding the fine details of the semantics is not important for understanding the central material in this paper.

[2] For expository purposes we have made a minor alteration to the Crashreporter.log format to allow us to explain more concepts with a single example. The evaluation section reports results on both the completely unmodified Crashreporter.log and the modified version.

```
dumpReport =
  union {
    struct {
      "Started writing crash report to: ";
      file:Ppath;
    };
    ...
  };

reporterReport =
  struct {
    function: Ppath; " reply failed: ");
    failuremsg: Pstring_('\n');
  };

dateOption =
  union {
    "- ";
    struct {
      day:    PDate;  " ";
      time:   PTime;  " ";
      year:   Pint32; " ";
    };
  };

source =
  arrayST {
    struct {
      date:   dateOption;
      kind:   enum {"crashdump";
                    "crashreporterd";}; "[";
      dumpid: Pint32; "]: ";
      report:
        switch kind of {
          "crashdump"      => dumpReport
          "crashreporterd" => reporterReport
    };
  }['\n',EOF];
```

Figure 3. Hand-written IR Crashreporter.log description.

Figure 3 contains three other definitions aside from `source`. These definitions specify the structure of the `dumpReport`, `reporterReport` and `dateOption` types.

3. The Inference Algorithm

Figure 4 gives an overview of our automatic tool generation architecture. The process begins with raw data, shown in blue (or grey) at the top left, which we pipe into the format inference engine (circumscribed by dotted lines in the picture). This engine produces a syntactically correct PADS description for the data through a series of phases: chunking and tokenization, structure discovery, information-theoretic scoring, and structure refinement. The system then feeds the generated PADS description into the PADS compiler. The compiler generates libraries, which the system then links to generic programs for various tasks including a data analysis tool (*a.k.a.,* the *accumulator*) and an ad-hoc-to-XML translator. At this point, users can apply these generated tools to their original raw data or to other data with the same format. The following subsec-

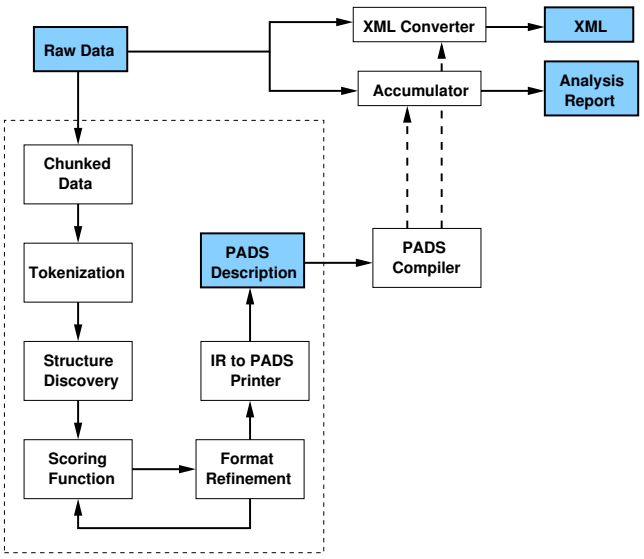

Figure 4. Architecture of the automatic tool-generation engine

tions describe the main components of the inference algorithm in more detail. We illustrate the effect of each phase on our running examples and present the output of some of the generated tools.

3.1 Chunking and Tokenization

The learning system first divides the input data, which we refer to as the *training set*, into *chunks* as specified by the user. Intuitively, a chunk is a unit of repetition in the data source. It is primarily by analyzing sequences of such chunks for commonalities that we are able to infer data descriptions. Our tool currently supports chunking on a line-by-line basis as well as on a file-by-file basis.

We use a lexer to break each chunk into a series of *simple tokens*, which are intuitively atomic pieces of data such as numbers, dates, times, alpha-strings, or punctuation symbols. Every simple token has a corresponding base type in the IR, though the converse is not true – there are base types that are not used as tokens. Nevertheless, since simple tokens have a very close correspondence with base types, we often use the word *token* interchangeably with *base type*.

Parenthetical syntax, including quotation marks, curly braces, square brackets, parentheses and XML tags, often provides very important hints about the structure of an ad hoc data file. Therefore, whenever the lexer encounters such parentheses, it creates a *meta-token*, which is a compound token that represents the pair of parentheses and all the tokens within.[3] For example, in Crashreporter.log, the syntax `[2164]` will yield the meta-token `[*]` instead of the sequence of three simple tokens `[`, `Pint`, and `]`. The structure-discovery algorithm eliminates all meta-tokens during its analysis; whenever it encounters a context consisting of matching meta-tokens, it cracks open the meta-tokens so it can analyze the underlying structure.

Our learning system has a default tokenization scheme skewed toward systems data, but users may specify a different scheme for their own domain through a configuration file. For example, computational biologists may want to add DNA strings `CATTGTT...` to the default tokenization scheme. The configuration file is essentially a list of name, regular expressions pairs. The system uses the configuration file to generate part of the system's lexer, a collection of new IR base types, and a series of type definitions that are incorporated into the final PADS specification.

[3] If parenthetical elements are not well-nested, the meta-tokens are discarded and replaced with ordinary sequences of simple tokens.

3.2 Structure Discovery

Given a collection of tokenized chunks, the goal of the structure-discovery phase is to quickly find a candidate description "close" to a good final solution. The rewriting phase then analyzes, refines and transforms this candidate to produce the final description. The high-level form of our structure-discovery algorithm was inspired by the work of Arasu and Garcia-Molina on information extraction from web pages [1]; however, the context, goals and algorithmic details of our work are entirely different.

Structure Discovery Basics. Our algorithm operates by analyzing the collection of tokenized chunks and guessing what the top-level type constructor should be. Based on this guess, it partitions the chunks and recursively analyze each partition to determine the best description for that partition. Figure 5 outlines the overall procedure in Pseudo-ML. The `oracle` function, whose implementation we hide for now, does most of the hard work by conjuring one of four different sorts of prophecies.

The `BaseProphecy` simply reports that the top-level type constructor is a particular base type.

The `StructProphecy` specifies that the top-level description is a struct with k fields. It also specifies a list, call it `css`, with k elements. The i^{th} element in `css` is the list of chunks corresponding to the i^{th} field of the struct. The oracle derives these chunk lists from its original input. More specifically, if the oracle guesses there will be k fields, then each original chunk is partitioned into k pieces. The i^{th} piece of each original chunk is used to recursively infer the type of the i^{th} field of the struct.

The `ArrayProphecy` specifies that the top-level structure involves an array. However, predicting exactly where an array begins and ends is difficult, even for the magical oracle. Consequently, the algorithm actually generates a three-field struct, where the first field allows for slop prior to the array, the middle field is the array itself, and the last field allows for slop after the array. If the slop turns out to be unnecessary, the rewriting rules will clean up the mess in the next phase.

Finally, the `UnionProphecy` specifies that the top-level structure is a union type with k branches. Like a `StructProphecy`, the `UnionProphecy` carries a chunks list, with one element for each branch of the union. The algorithm uses each element to recursively infer a description for the corresponding branch of the union. Intuitively, the oracle produces the union chunks list by "horizontally" partitioning the input chunks, whereas it partitions struct chunks "vertically" along field boundaries.

As an example, recall the Crashreporter.log data from Figure 2. Assuming a chunk is a line of data, the two chunks in the example consist of the token sequences (recall `[*]` and `(*)` are meta-tokens):

```
Pdate ' ' Ptime ' ' Pint ' ' Palpha [*] ':' ...
'-' ' ' Palpha [*] ':' ' ' Palpha (*) ' ' ...
```

Given these token sequences, the oracle will predict that the top-level type constructor is a struct with three fields: one for the tokens before the token `[*]`, one for the `[*]` tokens themselves, and one for the tokens after the token `[*]`. We explain how the oracle makes this prediction in the next section. The oracle then divides the original chunks into three sets as follows.

```
Pdate ' ' Ptime ' ' Pint ' ' Palpha          (set 1)
'-' ' ' Palpha

[*]                                          (set 2)
[*]

':' ...                                      (set 3)
':' ' ' Palpha (*) ' ' ...
```

On recursive analysis of set 1, the oracle again suggests a struct is the top-level type, generating two more sets of chunks:

```
Pdate ' ' Ptime ' ' Pint ' '                (set 4)
'-' ' '

Palpha                                      (set 5)
Palpha
```

Now, since every chunk in set 5 contains exactly one base type token, the recursion bottoms out with the oracle claiming it has found the base type `Palpha`. When analyzing set 4, the oracle detects insufficient commonality between chunks and decides the top-most type constructor is a union. It partitions set 4 into two more sets, with each group containing only 1 chunk (either {`Pdate ' ' ...`} or {`'-' ' '`}). The algorithm analyzes the first set to determine the type of the first branch of the union and the second set to determine the second branch of the union. With no variation in either branch, the algorithm quickly discovers an accurate type for each.

Having completely discovered the type of the data in set 1, we turn our attention to set 2. To analyze this set, the algorithm cracks open the `[*]` meta-tokens to recursively analyze the underlying data, a process which yields `struct {'['; Pint; ']';}`. Analysis of Set 3 proceeds in a similar fashion.

As a second example, consider the Sirius data from Figure 2. Here the chunks have the following structure:

```
Pint '|' Pint '|' ... '|' Pint '|' Pint
Pint '|' Pint '|' ... '|' Palpha Pint '|' Pint
```

The oracle prophecies that the top-level structure involves an array and partitions the data into sets of chunks for the array preamble, the array itself, and the array postamble. It does this partitioning to cope with "fence-post" problems in which the first or the last entry in an array may have slightly different structure. In this case, the preamble chunks all have the form {`Pint '|'`} while the postamble chunks all have the form {`Pint`}, so the algorithm easily determines their types. The algorithm discovers the type of the array elements by analyzing the residual list of chunks

```
Pint '|'
...
Pint '|'
Pint '|'
...
Palpha Pint '|'
```

The oracle constructs this chunk list by removing the preamble and postamble tokens from all input chunks, concatenating the remaining tokens, and then splitting the resulting list into one chunk per array element. It does this splitting by assuming that the chunk for each array element ends with a `'|'` token.

So far so good, but how does the guessing work? Why does the algorithm decide the Sirius data is basically an array but Crashreporter.log is a struct? After all, the Sirius chunks all have a `Pint`, just as all the Crashreporter.log chunks have a bracket meta-token `[*]`. Likewise, Crashreporter.log contains many occurrences of the `' '` token, which might serve as an array separator as the `'|'` token does in the Sirius data.

The Magic. To generate the required prophecy for a given list of chunks, the oracle computes a histogram of the frequencies of all tokens appearing the input. More specifically, the histogram for token t plots the number of chunks (on the y-axis) having a certain number of occurrences of the token (on the x-axis). Figure 6 presents a number of histograms computed during analysis of the Crashreporter.log and Sirius chunk lists.

```
type description (* an IR description *)
type chunk       (* a tokenized chunk *)
type chunks = chunk list

(* A top-level description guess *)
datatype prophecy =
   BaseProphecy   of description
 | StructProphecy of chunks list
 | ArrayProphecy  of chunks * chunks * chunks
 | UnionProphecy  of chunks list

(* Guesses the best top-level description *)
fun oracle : chunks -> prophecy

(* Implements a generic inference algorithm *)
fun discover (cs:chunks) : description =
 case (oracle cs) of
   BaseProphecy b => b

 | StructProphecy css =>
     let Ts = map discover css in
     struct { Ts }

 | ArrayProphecy (csfirst,csbody,cslast) =>
     let Tfirst = discover csfirst in
     let Tbody  = discover csbody  in
     let Tlast  = discover cslast  in
     struct { Tfirst; array { Tbody }; Tlast; }

 | UnionProphecy css =>
     let Ts = map discover css in
     union { Ts }
```

Figure 5. A generic structure-discovery algorithm in Pseudo-ML.

Intuitively, tokens associated with histograms with high *coverage*, meaning the token appears in almost every chunk, and *narrow* distribution, meaning the variation in the number of times a token appears in different chunks is low, are good candidates for defining structs. Similarly, histograms with high coverage and *wide* distribution are good candidates for defining arrays. Finally, histograms with low coverage or intermediate width represent tokens that form part of a union.

Concretely, consider histogram (a) from Figure 6. It is a perfect struct candidate– it has a single column that covers 100% of the records. Indeed, this histogram corresponds to the `[*]` token in Crashreporter.log. Whenever the oracle detects such a histogram, it will always prophecy a struct and partition the input chunks according to the associated token. All of the other top-level histograms for Crashreporter.log contain variation and hence are less certain indicators of data source structure.

As a second example, consider the top-level histograms (f), (b) and (g) for tokens `Palpha`, `Pint` and `Pwhite`, respectively, and compare them with the corresponding histograms (h), (i) and (j) computed for the same tokens from chunk set 1, defined in the previous subsection. The histograms for chunk set 1 have far less variation than the corresponding top-level histograms. In particular, notice that histogram (h) for token `Palpha` is a perfect struct histogram whereas histogram (f) for token `Palpha` contains a great deal of variation. This example illustrates the source of the power of our divide-and-conquer algorithm– if the oracle can identify *even one token* at a given level as defining a good partition for the data, the histograms for the next level down become substantially sharper and more amenable to analysis.

As a third example, consider histogram (k). This histogram illustrates the classic pattern for tokens involved in arrays– it has a very long tail. And indeed, the `|` token in the Sirius data does act like a separator for fields of an array.

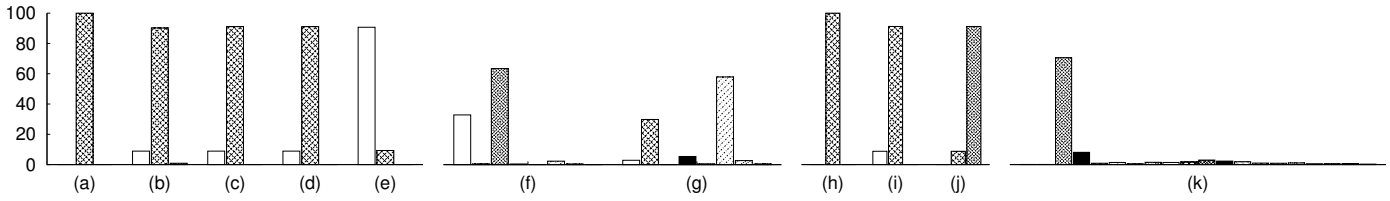

Figure 6. Histograms (a), (b), (c), (d), (e), (f) and (g) are generated from top-level analysis of Crashreporter.log tokens. The corresponding tokens are (a) `[*]`, (b) `Pint`, (c) `PDate`, (d) `PTime`, (e) `-`, (f) `Palpha` and (g) `Pwhite`. Histograms (h) `Palpha`, (i) `Pint`, and (j) `Pwhite` are generated from analysis of Crashreporter.log from set 1 (the second level of recursion). Histogram (k) is generated from top-level analysis of the `|` token from the Sirius data. Note that several of these histograms have many bars of very small height, including (f) with 7, (g) with 8, and (k) with 17.

To make the intuitions discussed above precise, we must define a number of properties of histograms. First, a histogram h for a token t is a list of pairs of natural numbers (x, y) where x denotes the token frequency and y denotes the number of chunks with that frequency. All first elements of pairs in the list must be unique. The *width* of a histogram (*width*(h)) is the number of elements in the list excluding the zero-column (*i.e.* excluding element $(0, y)$). A histogram $\bar{h}$ is in our normal form when the first element of the list is the zero column and all subsequent elements are sorted in descending order by the y component. For example, if h_1 is the histogram $[(0, 5), (1, 10), (2, 25), (3, 15)]$ then *width*(h_1) is 3 and its normal form $\bar{h_1}$ is $[(0, 5), (2, 25), (3, 15), (1, 10)]$.

We often refer to y as the *mass* of the element (x, y), and given a histogram h, we refer to the mass of the i^{th} element of the list using the notation $h[i]$. For instance, $h_1[3] = 15$ and $\bar{h_1}[3] = 10$. The *residual mass* (*rm*) of a column i in a normalized histogram h is the mass of all the columns to the right of i plus the mass of the zero-column. Mathematically, $rm(\bar{h}, i) = \bar{h}[0] + \sum_{j=i+1}^{width(\bar{h})} \bar{h}[j]$. For example, $rm(\bar{h_1}, 1) = 5 + 15 + 10 = 30$. The residual mass characterizes the "narrowness" of a histogram. Those histograms with low residual mass of the first column (*i.e.,* $rm(\bar{h_1}, 1)$ is small) are good candidates for structs because the corresponding tokens occur exactly the same number of times in almost all records.

To distinguish between structs, arrays and unions, we also need to define the *coverage* of a histogram, which intuitively is the number of chunks containing the corresponding token. Mathematically, it is simply the sum of the non-zero histogram elements: $coverage(\bar{h}) = \sum_{j=1}^{width(\bar{h})} \bar{h}[j]$.

Finally, our algorithm works better when the oracle considers groups of tokens with similar distributions together because with very high probability such tokens form part of the same type constructor. To determine when two histograms are *similar*, we use a symmetric form of *relative entropy* [13]. The (plain) relative entropy of two normalized histograms $\bar{h_1}$ and $\bar{h_2}$, written $\mathcal{R}(\bar{h_1} \mid\mid \bar{h_2})$, is defined as follows.

$$\mathcal{R}(\bar{h_1} \mid\mid \bar{h_2}) = \sum_{j=1}^{width(\bar{h_1})} \bar{h_1}[j] * log(\bar{h_1}[j]/\bar{h_2}[j])$$

To create a symmetric form, we first find the average of the two histograms in question (written $h_1 \oplus h_2$) by summing corresponding columns and dividing by two. This technique prevents the denominator from being zero in the final relative entropy computation. Using this definition, the symmetric relative entropy is:

$$\mathcal{S}(\bar{h_1} \mid\mid \bar{h_2}) = \frac{1}{2}\mathcal{R}(\bar{h_1} \mid\mid \bar{h_1} \oplus \bar{h_2}) + \frac{1}{2}\mathcal{R}(\bar{h_2} \mid\mid \bar{h_1} \oplus \bar{h_2})$$

Now that we have defined the relevant properties of histograms, we can explain how the oracle prophecies given a list of chunks.

1. Prophecy a base type when each chunk contains the same simple token. If each chunk contains the same meta-token, prophecy a struct with three fields: one for the left paren, one for the body, and one for the right paren.
2. Otherwise, compute normalized histograms for the input and group related ones into clusters using agglomerative clustering: A histogram h_1 belongs to group G provided there exists another histogram h_2 in G such that $\mathcal{S}(\bar{h_1} \mid\mid \bar{h_2}) < \mathrm{ClusterTolerance}$. where $\mathrm{ClusterTolerance}$ is a parameter of the algorithm. We do not require all histograms in a cluster to have precisely the same histogram to allow for errors in the data. A histogram dissimilar to all others will form its own group. We have found a $\mathrm{ClusterTolerance}$ of 0.01 is effective.
3. Determine if a struct exists by first ranking the groups by the minimum residual mass of all the histograms in each group. Find the first group in this ordering with histograms h satisfying the following criteria:
   - $rm(h) < \mathrm{MaxMass}$
   - $coverage(h) > \mathrm{MinCoverage}$

   where constants $\mathrm{MaxMass}$ and $\mathrm{MinCoverage}$ are parameters of the algorithm. This process favors groups of histograms with high coverage and narrow distribution. If histograms $h_1, \ldots, h_n$ from group G satisfy the struct criteria, the oracle will prophecy some form of struct. It uses the histograms $h_1, \ldots, h_n$ and the associated tokens $t_1, \ldots, t_n$ to calculate the number of fields and the corresponding chunk lists. We call $t_1, \ldots, t_n$ the *identified* tokens for the input. Intuitively, for each input chunk, the oracle puts all tokens up to but not including the first token t from the set of identified tokens into the chunk list for the first field. It puts t in the chunk list for the second field. It puts all tokens up to the next identified token into the chunk list for the third field and so on. Of course, the identified tokens need not appear in the same order in all input chunks, nor in fact must they all appear at all. To handle this variation when it occurs, the oracle prophecies a union instead of a struct, with one branch per token ordering and one branch for all input chunks that do not have the full set of identified tokens.
4. Identify an array by sorting all groups in descending order by coverage of the highest coverage histogram in the group. Find the first group in this ordering with any histograms that satisfy the following minimum criteria:
   - $width(h) > 3$
   - $coverage(h) > \mathrm{MinCoverage}$

   This process favors histograms with wide distribution and high coverage. If histograms $h_1, \ldots, h_n$ with corresponding tokens $t_1, \ldots, t_n$ satisfy the array criteria, the oracle will prophecy an array. It will partition each input chunk into (1) a preamble subsequence that contains the first occurrence of each identi-

fied token, (2) a set of element subsequences, with each subsequence containing one occurrence of the identified tokens, and (3) a postamble subsequence that contains any remaining tokens from the input chunk.

5. If no other prophecy applies, identify a union. Partition the input chunks according to the first token in each chunk.

3.3 Information-Theoretic Scoring

We use an information theoretic scoring function to assess the quality of our inferred descriptions and to decide whether to apply rewriting rules to refine candidate descriptions. Intuitively, a good description is one that is both *compact* and *precise*. There are trivial descriptions of any data source that are highly compact (*e.g.*, the description that says the data source is a string terminated by end of file) or perfectly precise (*e.g.*, the data itself abstracts nothing and therefore serves as its own description). A good scoring function balances these opposing goals. As is common in machine learning, we have defined a scoring function based on the *Minimum Description Length Principle* (MDL), which states that a good description is one that minimizes the cost (in bits) of transmitting the data [8]. Mathematically, if T is a description and $d_1, \ldots, d_k$ are representations of the k chunks in our training set, parsed according to T, then the total cost in bits is:

$$\textsc{Cost}(T, d_1, \ldots, d_k) = \mathrm{CT}(T) + \mathrm{CD}(d_1, \ldots, d_k \mid T)$$

where $\mathrm{CT}(T)$ is the number of bits to transmit the description and $\mathrm{CD}(d_1, \ldots, d_k \mid T)$ is the number of bits to transmit the data *given the description*.

Intuitively, the cost in bits of transmitting a description is the cost of transmitting the sort of description (*i.e.*, `struct`, `union`, `enum`, *etc.*) plus the cost of transmitting all of its subcomponents. For example, the cost of transmitting a struct type $\mathrm{CT}(\texttt{struct}\{T_1; \ldots; T_k;\})$ is $\textsc{Card} + \sum_{i=1}^{k} \mathrm{CT}(T_i)$ where $\textsc{Card}$ is the log of the number of different sorts of type constructors (24 of them in the IR presented in this paper). We have defined the recursive cost function mathematically in full, but space limitations preclude giving that definition here.

The cost of encoding data relative to selected types is shown in Figure 7. The top of the figure defines the cost of encoding all data chunks relative to the type T; it is simply the sum of encoding each individual chunk relative to T.

In the middle of the figure, we define the cost of encoding a chunk relative to one of the integer base types; other base types are handled similarly. Notice that the cost of encoding an integer relative to the constant type `PintConst` is zero because the type itself contains all information necessary to reconstruct the integer– no data need be transmitted. The cost of encoding data relative to `Pint32` or `Pint64` types is simply 32 or 64 bits, respectively. Finally, we artificially set the cost of ranged types $\texttt{PintRanged}(p_{min}, p_{max})$ to be infinity because our experiments reveal that attempting to define integer types with minimum and maximum values usually leads to overfitting of the data.[4]

The last section of Figure 7 presents the cost of encoding data relative to selected type constructors. The cost of encoding a `struct` is the sum of the costs of encoding its component parts. The cost of encoding a `union` is the cost of encoding the branch number ($log(k)$ if the union has k branches) plus the cost of encoding the branch itself. The cost of encoding an `enum` is the cost of encoding its tag only – given the tag, the underlying data is determined by the type. The cost of encoding a `switch` is the cost of encoding the branch only – the tag need not be encoded because it is determined by the type and earlier data.

[4] We nevertheless retain `PintRanged` types in our IR to encode the range of values found during the value-space analysis. During the rewriting phase, we use this range information to rewrite `PintRanged` into other integer types. Since the cost of encoding `PintRanged` is so high, the appropriate rewriting is guaranteed to be applied. In the future, we may emit this range information as comments in the generated descriptions.

Cost of encoding all training data relative to a type:

$$\mathrm{CD}(d_1, \ldots, d_k \mid T) \quad = \quad \sum_{i=1}^{k} \mathrm{CD'}(d_i \mid T)$$

Cost of encoding a single chunk relative to selected base types:

$$\begin{array}{lcl} \mathrm{CD'}(i \mid \texttt{PintConst}(p)) & = & 0 \\ \mathrm{CD'}(i \mid \texttt{Pint32}) & = & 32 \\ \mathrm{CD'}(i \mid \texttt{Pint64}) & = & 64 \\ \mathrm{CD'}(i \mid \texttt{PintRanged}(p_{min}, p_{max})) & = & \infty \end{array}$$

Cost of encoding a single chunk relative to selected types:

$$\begin{array}{l} \mathrm{CD'}((d_1, \ldots, d_k) \mid \texttt{struct}\{T_1; \ldots T_k;\}) \\ \qquad = \sum_{i=1}^{k} \mathrm{CD'}(d_i \mid T_i) \\ \mathrm{CD'}(in_i(d) \mid \texttt{union}\{T_1; \ldots T_k;\}) \\ \qquad = \texttt{log}(k) + \mathrm{CD'}(d \mid T_i) \\ \mathrm{CD'}(in_i(c) \mid \texttt{enum}\{c_1; \ldots c_k;\}) \\ \qquad = \texttt{log}(k) \\ \mathrm{CD'}(in_i(d) \mid \texttt{switch}\ x\ \texttt{of}\{c_1 \texttt{=>} T_1; \ldots c_k \texttt{=>} T_k;\}) \\ \qquad = \mathrm{CD'}(d \mid T_i) \end{array}$$

Figure 7. Cost of transmitting data relative to a type, selected rules

3.4 Structure Refinement

The goal of the structure-refinement phase is to improve the structure produced by the structure-discovery phase. We formulate the structure-refinement problem as a generalized search through the description space starting with the candidate produced by structure discovery. The objective of the search is to find the description that minimizes the information-theoretic scoring function.

Rewriting rules. To move around in the description space, we define a number of rewriting rules, the general form of which is

$$T \Rightarrow T', \text{ if some constraint } p(T) \text{ is satisfied,}$$

where T is a type in the candidate description and T' is its replacement after the rewriting. Some rules are unconditional and thus free of constraints. There are two kinds of rewriting rules: (1) data-independent rules which transform a type based exclusively on the syntax of the description; and (2) data-dependent rules which transform a type based both on the syntax of the description and on properties of the training data parsed by type T. In general, the data-independent rules try to rearrange and merge portions of the description while the data dependent rules seek to identify constant fields and enumerations, and to establish data dependencies between different parts of the description.

Figure 8 presents a selection of the rewriting rules used in the refinement phase. We have omitted many rules and have simplified others for succinctness. When $T[\![X]\!]$ appears in a pattern on the left-hand side of a rewriting rule, X is bound to the set of data representations resulting from using T to parse the appropriate part of each chunk from the training set. Furthermore, let $card(X)$ be the cardinality of the set X, and let $X(i)$ be the data representation resulting from parsing the i^{th} chunk in the training set. Finally, given a union value $\texttt{in}_j(v)$, we define $tag(\texttt{in}_j(v))$ to be j.

The Search. The core of the rewriting system is a recursive, depth-first, greedy search procedure. By "depth-first," we mean the algorithm considers the children of each structured type before considering the structure itself. When refining a type, the algorithm selects the rule that would *minimize* the information-theoretic score of the resulting type and applies this rule. This process repeats until

Data independent rules

1. Singleton structs and unions
   $\texttt{struct}\{T\} \Rightarrow T$ $\quad$ $\texttt{union}\{T\} \Rightarrow T$

   $\texttt{struct}\{\} \Rightarrow \texttt{Pempty}$ $\quad$ $\texttt{union}\{\} \Rightarrow \texttt{Pvoid}$
2. Struct and union clean-up
   $\texttt{struct}\{pre\_types; \texttt{Pvoid}; post\_types\} \Rightarrow \texttt{Pvoid}$

   $\texttt{struct}\{pre\_types; \texttt{Pempty}; post\_types\} \Rightarrow$
   $\texttt{struct}\{pre\_types; post\_types\}$

   $\texttt{union}\{pre\_types; \texttt{Pvoid}; post\_types\} \Rightarrow$
   $\texttt{union}\{pre\_types; post\_types\}$
3. Uniform struct to fixed-length array
   $\texttt{struct}\{T_1; \ldots; T_n\} \Rightarrow \texttt{arrayFW}\{T_1\}[n]$
   if $n \geq 3$ and $\forall i \in [1,\ n],\ j \in [1,\ n] :\ T_i = T_j$.
4. Common postfix in union branches
   $\texttt{union}\{\texttt{struct}\{pre\_types_1; T\};$
   $\texttt{struct}\{pre\_types_2; T\}\} \Rightarrow$
   $\texttt{struct}\{\texttt{union}\{\texttt{struct}\{pre\_types_1\};$
   $\texttt{struct}\{pre\_types_2\}\}; T\}$

   $\texttt{union}\{\texttt{struct}\{pre\_types; T\}; T\} \Rightarrow$
   $\texttt{struct}\{\texttt{option}\{\texttt{struct}\{pre\_types\}\}; T\}$
5. Combine adjacent constant strings
   $\texttt{struct}\{pre\_types; \texttt{PstringConst}(\texttt{c}_1);$
   $\texttt{PstringConst}(\texttt{c}_2); post\_types\} \Rightarrow$
   $\texttt{struct}\{pre\_types; \texttt{PstringConst}(\texttt{c}_1 @ \texttt{c}_2); post\_types\}$

Data dependent rules

1. Base type with unique values to constant
   $\texttt{Pint}[\![X]\!] \Rightarrow \texttt{PintConst}(\texttt{c})$
   if $\forall x \in X :\ x = c$.

   $\texttt{Palpha}[\![X]\!] \Rightarrow \texttt{PstringConst}(\texttt{c})$
   if $\forall x \in X :\ x = c$.

   $\texttt{Pstring}[\![X]\!] \Rightarrow \texttt{PstringConst}(\texttt{c})$
   if $\forall x \in X :\ x = c$.

   $\texttt{Pother}[\![X]\!] \Rightarrow \texttt{PstringConst}(\texttt{c})$
   if $\forall x \in X :\ x = c$.
2. Refine enums and ranges
   $\texttt{Pstring}[\![X]\!] \Rightarrow \texttt{enum}\{s_1; \ldots; s_k\}$
   if $\forall x \in X :\ x \in \{s_1, \ldots, s_k\}$.

   $\texttt{Pint}[\![X]\!] \Rightarrow \texttt{Pint32}$
   if $\forall x \in X :\ 0 \leq x < 2^{32}$.
3. Union to switch
   $\texttt{struct}\{pre\_types; \texttt{enum}\{c_1; \ldots; c_n\}[\![X]\!]; mid\_types;$
   $\texttt{union}\{T_1; \ldots; T_n\}[\![Y]\!]; post\_types\}$
   $\Rightarrow$
   $\texttt{struct}\{pre\_types, z : \texttt{enum}\{c_1; \ldots; c_n\}; mid\_types;$
   $\texttt{switch}(z)\{c_1 \Rightarrow T_{\Pi(1)}; \ldots; c_n \Rightarrow T_{\Pi(n)}\}; post\_types\}$
   where z is a fresh variable, and there exists a permutation Π, s.t. $\forall i \in [1,\ card(X)], \Pi(tag(X(i))) = tag(Y(i))$.

Figure 8. Selected and simplified rewriting rules

```
(* rewriting rules *)
type rule : description -> description
val rules : rule list

(* measure the score for a type *)
fun score : description -> float

(* find the type with best score from a list *)
fun best: description list -> description

(* improve the given type by one rewriting rule *)
fun oneStep (T:description) : description =
 let all = map (fn rule => rule T) rules in
 let top = best all                      in
 if (score top) < (score T) then oneStep top
 else T

(* main function to refine an IR description *)
fun refine (T:description) : description =
  let T' = case T of
      base b => b
    | struct { Ts } => struct { map refine Ts }
    | union { Ts } => union { map refine Ts }
    | switch x of { vTs } =>
       switch x of
         { map (fn (v, t) => (v, refine t)) vTs }
    | array { T } =>
              array { refine T }
    | option { T } => option { refine T } in
  oneStep T'
```

Figure 9. Generic local optimization algorithm in Pseudo-ML

no further reduction in the score is possible, at which point we say the resulting type T is *stable*.

The rewriting phase applies the algorithm given in Figure 9 three times in succession. The first time, the algorithm quickly simplifies the initial candidate description using *only* data-independent rules. The second time, it uses the data-dependent rules to refine base types to constant values and enumerations, *etc.*, and to introduce dependencies such as switched unions. This stage requires the value-space analysis described next. The third time, the algorithm re-applies the data-independent rules because some stage two rewritings (such as converting a base type to a constant) enable further data-independent rewritings.

Value-space analysis. We perform a value-space analysis prior to applying the data-dependent rules. This analysis first generates a set of relational tables from the input data. Each row in a table corresponds to an input chunk and each column corresponds to either a particular base type from the inferred description, or to a piece of meta-data from the description. Examples of meta-data include the tag number from union branches and the length of arrays. We generate a *set* of relational tables as opposed to a single table as the elements of each array occupy their own separate table (a description with no arrays will have only one associated table).

We analyze every column of every table to determine properties of the data in that column such as constancy and value range. To find inter-column properties, we have implemented a simplified variant of the TANE algorithm [11], which identifies functional dependencies between columns in relational data. Because full TANE is too expensive (possibly exponential in the number of columns), and produces many false positives when invoked with insufficient data, our simplified algorithm computes only binary dependencies. We use the result of this dependency analysis to identify switched unions and fixed-size arrays.

Running example. To illustrate the refinement process, we walk through a few of the steps taken to rewrite the Crashreporter.log description. The first part of the candidate description generated by the structure-discovery algorithm appears below.

```
struct {
  union {
    struct {
      Pdate; Pwhite; Ptime; Pwhite; Pint;
```

```
      Pwhite;          (*)
    };
    struct {
      "-";
      Pwhite;          (*)
    };
  }
  Palpha; "["; Pint; "]";
  union { ... };
};
```

In the first data-independent stage of rewriting, the common trailing white space marked `(*)` is pulled out of the union branches into the surrounding struct using the "common postfix in union" rule. This transformation leaves behind the single-element struct marked `(**)` in the result below; rewriting rules in stage three will transform this verbose form into the more compact constant string `"-"`. This first rewriting stage also pulls colon and whitespace characters out of the trailing union (not shown in the candidate description).

```
struct {
  union {
    struct { Pdate; Pwhite; Ptime; Pwhite; Pint; };
    struct { "-" };                    (**)
  }
  Pwhite;                              (*)
  Palpha; "["; Pint; "]"; ":"; Pwhite;
  union { ... };
};
```

In the second rewriting stage, data-dependent rules 1 and 2 convert appropriate base types into constants and enums. Moreover, TANE discovers a data dependency between the newly introduced enumeration involving `"crashdump"` and `"mach_msg"`, and the structure of the following message. Hence, we introduce a switched union. Notice that the switched union branches on a different enum than the hand-written IR in Figure 3 because the inference algorithm found a different way of structuring the data. Nonetheless, both of these descriptions are accurate.

```
struct {
  union {
    struct { Pdate; " "; Ptime; " "; 2006; };
    struct { "-" };
  };
  " "; enum {"crashreporterd", "crashdump"};
  "["; PintRanged [120...29874]; "]"; ":"; " ";
  x19:enum {"crashdump", "mach_msg", "Finished",
            "Started", "Unable", "Failed"};
  switch x19 of { ... };
};
```

In the third and final stage, data independent rule 5 combines constants and rule 1 flattens the singleton struct, resulting in the final IR description:

```
struct {
  union {
    struct { Pdate; " "; Ptime; " "; 2006; };
    "-";
  };
  " "; enum {"crashreporterd", "crashdump"};
  "["; Pint32; "]: ";
  x19:enum {"crashdump", "mach_msg", "Finished",
        "Started", "Unable", "Failed"};
  switch x19 of { ... };
  };
};
```

The information-theoretic complexity of the final description relative to the data in our training set is 304538 bits. The candidate description produced by the structure-discovery phase had a complexity of 416156 bits. The absolute values of these quantities are relatively unimportant, but the fact that the final complexity is substantially smaller than the original suggests that our search procedure optimized the description effectively.

Tiny fragment of XML output from crashreporter.log:

```
<Struct_114>
  <var_7>
    <var_6>
      <var_0><val>Sat Jun 24</val></var_0>
      <var_2><val>06:38:46</val></var_2>
      <var_4><val>2006</val></var_4>
    </var_6>
  </var_7>
  <var_11><val>crashdump</val></var_11>
  <var_14><val>2164</val></var_14>
...
```

Graph generated from ai.3000 web transaction volume at different times of the day (00:00-8:55 and 19:00-24:00):

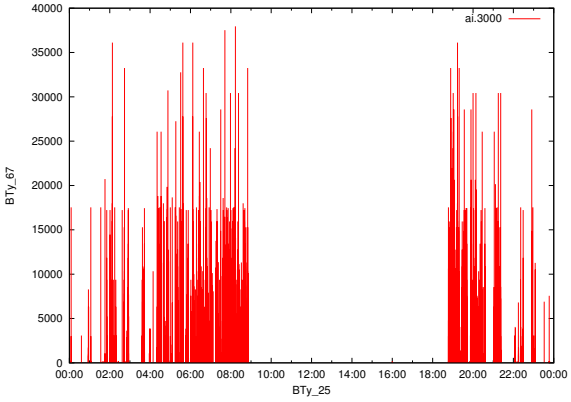

Figure 10. End products of automatically generated tools.

3.5 End Products

The previous subsections outline the central technical elements of our algorithms. The main tasks remaining include converting the internal representation into a syntactically correct PADS description, feeding the generated description to the PADS compiler and producing a collection of scripts that conveniently package the freshly-generated libraries with the PADS run-time system and tools. At the end of this process, users have a number of programming libraries and many powerful tools at their disposal. Perhaps the most powerful tools are the PADX query engine [4] and the XML converter, which allow users to write arbitrary XQueries over the data source or to convert the data to XML for use by other software. Other useful tools include the accumulator tool, mentioned earlier, converters to translate data into a form suitable for loading into a relational database or Excel spreadsheet, and a custom graphing tool that pushes data into `gnuplot` for data visualization. Figure 10 gives snapshots of the output of a couple of these tools.

4. Experimental Evaluation

We conducted a series of experiments to study the correctness and performance of our format inference algorithm. Table 1 lists the data sources we used in the experiments; they range from system logs to application outputs to government statistics. Except for sirius.1000, which is a proprietary format, the files are all available from `www.padsproj.org/learning.html`. The size of the benchmarks varies from a few thousand lines to just a few dozen.

Data source	KB/Chunks	Description
1967Transactions.short	70/999	transaction records
MER_T01_01.cvs	22/491	comma-sep records
ai.3000	293/3000	webserver log
asl.log	279/1500	log file of Mac ASL
boot.log	16/262	Mac OS boot log
crashreporter.log	50/441	original crash log
crashreporter.log.mod	49/441	modified crash log
sirius.1000	142/999	AT&T phone provision data
ls-l.txt	2/35	Stdout from Unix command ls -l
netstat-an	14/202	output from netstat
page_log	28/354	printer logs
quarterlypersonalincome	10/62	spread sheet
railroad.txt	6/67	US rail road info
scrollkeeper.log	66/671	application log
windowserver_last.log	52/680	log from LoginWindow server on Mac
yum.txt	18/328	log from pkg install

Table 1. Benchmark profile including filename, size in KB, number of chunks and brief description.

Data source	SD(s)	Ref(s)	Tot(s)	HW(h)
1967Transactions.short	0.20	2.32	2.56	4.0
MER_T01_01.csv	0.11	2.80	2.92	0.5
ai.3000	1.97	26.35	28.64	1.0
asl.log	2.90	52.07	55.26	1.0
boot.log	0.11	2.40	2.53	1.0
crashreport.log	0.12	3.58	3.73	2.0
crashreport.log.mod	0.15	3.83	4.00	2.0
sirius.1000	2.24	5.69	8.00	1.5
ls-l.txt	0.01	0.10	0.11	1.0
netstat-an	0.07	0.74	0.82	1.0
page_log	0.08	0.55	0.65	0.5
quarterlypersonalincome	0.07	5.11	5.18	48
railroad.txt	0.06	2.69	2.76	2.0
scrollkeeper.log	0.13	3.24	3.40	1.0
windowserver_last.log	0.37	9.65	10.07	1.5
yum.txt	0.11	1.91	2.03	5.0

Table 2. Execution times. SD: time for structure-discovery phase; Ref: time for scoring and refinement; Tot: end-to-end time for complete inference algorithm; HW: time taken *in hours* to hand-write the corresponding description.

Most of the data files are "line based," meaning that every line becomes a chunk for the purposes of learning the format. One exception is netstat-an, in which chunks comprise multiple lines. We include two versions of crashreporter.log: the original "crashreporter.log" and the slightly modified "crashreporter.log.mod" that we used as an example in this paper. We include both to demonstrate that our minor modifications were simply for expository purposes.

Performance. Our first set of experiments measures the time required to infer a description from example data. In all our experiments, we used an Apple PowerBook G4 with a 1.67 GHz Processor and 512 MB DDR RAM running on Mac OSX 10.4 Tiger. Table 2 presents the execution times for the structure-discovery phase (SD), the refinement phase (Ref) and the total (Tot) end-to-end time of the algorithm including printing PADS descriptions and other

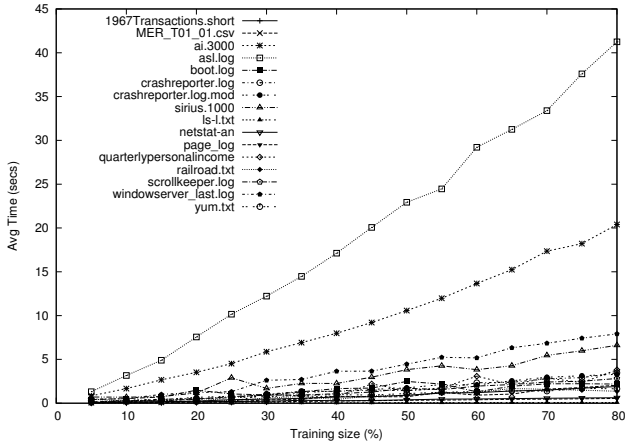

Figure 11. Execution times of training sets

overhead, all measured in seconds. For accurate timing measurements, we ran the algorithm 10 times, and found the average after removing the best and the worst times.

There are two main lessons to take away from this initial set of benchmarks. First, the overall time to infer the structure of any our example files was less than a minute, and was less than 10 seconds except on a couple of the larger files. Hence, although we have spent very little time optimizing our algorithm, it already appears perfectly capable of being used in real time by a programmer wishing to understand and process small ad hoc data files. Second, discovery of an initial format is usually very fast, taking less than 3 seconds in all cases. Most of the algorithm's time is spent in format rewriting, which often takes a factor of 10 or more time than structure discovery. Moreover, most of the rewriting time is taken in the data analysis phase (numbers not shown). Consequently, if format rewriting (particularly the data analysis phase) is taking too long, the user may abort it to produce a slightly less refined description that may nevertheless be perfectly sufficient.

To give a very rough idea of how using the inference system compares with programming descriptions by hand, we also measured the time it took for a person to write descriptions of all of the data sources (See Table 2 again). Initially, our programmer (a Ph.D. in computer science) knew very little about how the PADS system worked in practice, having only read a few of our conference papers. Consequently, writing the first description took a long time, approximately 48 hours (two days of working at an "ordinary" pace) for quarterlypersonalincome. While different people with different backgrounds will clearly learn at different rates, there is little doubt that the format inference algorithm is a tremendous benefit to novices, particularly to those data analysts without a Ph.D. in computer science, who could not care less about learning some new data description language. After some practice, our programmer was able to write most descriptions in 1 to 2 hours, so generating descriptions in a few seconds still has great benefit, even to experts.

To understand the scaling behavior of our algorithm, we randomly selected 5%, 10%, 15%, ..., 80% of the chunks in every data source and measured the performance of the algorithm on each subset of the data that was selected. Figure 11 plots the execution time against the percentage of each data source selected. These experiments suggest that once a format is fixed, the cost of inference grows linearly with the amount of data. However, it is also clear that the raw size of the data is not the only factor determining performance. The nature and complexity of the format is also a significant factor. For instance, widowserver_last.log is only one third the size of sirius.1000, but takes substantially longer for the inference algorithm to process.

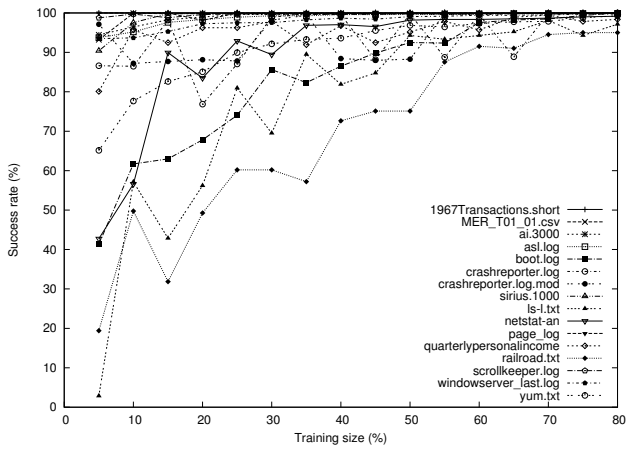

Figure 12. Success rates of training sets

Correctness. To evaluate the correctness of our algorithm, we again selected random subsets of each data source, trained our algorithm on those subsets and measured the error rate of the inferred parser on the remaining data. Figure 12 graphs the percentage of successfully parsed records versus the percentage of the data used in training. Note that accuracy does not uniformly improve. This variation is caused by the randomness in our data selection and the fact that in some cases, we have very small absolute quantities of data relative to the underlying complexity of the formats. For instance, at 5% training size, ls-l.txt is just one line of data.

To understand the correctness properties of our algorithm from a different angle, we record the minimum training sizes in percentages required to achieve 90% and 95% accuracy for all the benchmarks in Table 3. This table also reports the normalized cost of a description (NCT), which we compute by dividing the first component of the information-theoretic score in Section 3.3 by the number of bits in the data. NCT gives a rough indication of the complexity of the data source. The higher the normalized score, the more complicated the data, and the greater the fraction of data is needed to learn an accurate description. The rows of of Table 3 are sorted in ascending NS score. From the table, one can see that ls-l.txt and railroad.txt have high NS scores. This is because they are quite small data sources (2KB and 6KB respectively), yet have relatively complicated formats. Consequently, it takes a substantial portion of the data to learn an accurate parser. For most of the other data sources, a substantially smaller percentage of the data is required to achieve high accuracy. Overall, for 11 of 16 benchmarks, less than 15% of the data is needed to achieve 95% accuracy or more.

5. Discussion

Dealing with errors. In 1967, Gold [7] proved that learning a grammar for any remotely sophisticated class of languages, such as the regular languages, is impossible if one is only given positive example data.[5] Given this negative theoretical result, and the practical fact that it is hard to be sure that training data is sufficiently rich to witness all possible variation in the data, errors in inference are inevitable. Fortunately, however, detecting and recovering from errors in ad hoc data is one of the primary strengths of the PADS system.

[5] A positive example is a data source known to be in the grammar to be learned. A negative example is one known *not* to be in the target grammar. Learning with positive examples and negative examples is possible. Unfortunately, given that data analysts are unlikely to have access to ad hoc data that they know *does not* satisfy the format they are interested in learning, we are forced to tackle the more difficult problem of learning from positive examples only.

Data source	NCT	90%	95%
sirius.1000	0.0001	5	10
1967Transactions.short	0.0003	5	5
ai.3000	0.0004	5	10
asl.log	0.0012	5	10
scrollkeeper.log	0.0020	5	5
page_log	0.0032	5	5
MER_T01_01.csv	0.0037	5	5
crashreporter.log	0.0052	10	15
crashreporter.log.mod	0.0053	5	15
windowserver_last.log	0.0084	5	15
netstat-an	0.0118	25	35
yum.txt	0.0124	30	45
quarterlypersonalincome	0.0170	10	10
boot.log	0.0213	45	60
ls-l.txt	0.0461	50	65
railroad.txt	0.0485	60	75

Table 3. Correctness measures. NCT: normalized cost of description; Min Training size (%) to obtain required accuracy

To determine exactly how accurate an inferred description is on any new data source, a user may run the accumulator tool. This tool catalogs exactly how many deviations from the description there were overall in the data source as well as the error rate in every individual field. Hence, using this tool, a programmer can immediately and reliably determine the effectiveness of inference for their data. If there is a serious problem, the user can easily edit the generated description by hand – identification of a problem field, a minor edit and recompilation of tools might just take 5 minutes. Hence, even imperfectly-generated descriptions have great value in terms of improving programmer productivity. Moreover, all PADS-generated parsers and tools have error detection, representation and recovery techniques. For instance, when converting data to XML, errors encountered are represented explicitly in the XML document, allowing users to query the data for errors if they choose. Before graphing ad hoc data, an analyst may use the accumulator tool to check if any errors occur in the fields to be graphed. If not, there is no reason to edit the description at all – graphing the correct fields may proceed immediately.

Future work. Perhaps the most significant weakness of our current system is that its relative effectiveness is somewhat brittle with respect to exactly how we perform token processing. Discovering tokens like "IP address" and "date" is highly beneficial as they act as compact, highly descriptive abstractions, but unfortunately, they are also often mutually ambiguous. For instance, an IP address, a file name, a floating point number, the version number for a software product, and a phone number can all be represented as some number of digits separated by periods. At the moment, we disambiguate between them in the same way that lex does, by taking the first, longest match. To improve tokenization in the future, we plan to look at learning probabilistic models of a broad range of token types. We also intend to explore finding new tokens from the data itself, possibly by identifying abrupt changes in entropy [12].

6. Related Work

Researchers have been studying *grammar induction*, the process of learning the structure of a data source, since the 1960s; De La Higuera surveys some recent trends [9]. However, our system is unique in two important ways. First, our inference algorithm does not stand alone; it is part of the more general PADS programming

environment. The fusion of the PADS system, including its automatic data representation generation, its error detection facilities, its generic programming environment, and its powerful tool suite, together with grammar induction is one of our key contributions. Second, many researchers have focused either on grammar induction for natural language processing or for information extraction from XML or HTML documents. In contrast, we focus on ad hoc data sources such as system logs and scientific data sets. Ad hoc data is substantially less structured syntactically than XML, and yet, unlike natural language, it is possible to assign our data sources accurate, compact descriptions. After searching the literature and consulting with experts in grammar induction at the CAGI 2007 workshop, where we presented a two page overview of our system [2], we could find no existing work comparable to ours.

Perhaps the most closely related work is from Arasu and Garcia-Molina [1], who developed an information extraction system for sets of similar web pages. Arasu uses a top-down grammar induction algorithm somewhat similar to our rough structure-inference phase (though it does not use histograms), but has no description rewriting engine. This algorithm exploits the hierarchical nesting structure of XML documents in essential ways and so cannot be applied directly to ad hoc data.

The TSIMMIS project [3] aims to allow users to manage and query collections of heterogeneous, ad hoc data sources. TSIMMIS sits on top of the Rufus system [18], which supports automatic classification of data sources based on features such as the presence of certain keywords, magic numbers appearing at the beginning of files and file type. This sort of classification is materially different from the syntactic analysis we have developed.

Potter's Wheel [17] is a system that attempts to help users find and purge errors from relational data sources. It does so through the use of a spread-sheet style interface, but in the background, a grammar inference algorithm infers the structure of the input data, which may be "ad hoc," somewhat like ours. This inference algorithm operates by enumerating all possible sequences of base types that appear in the training data. Since Potter's Wheel is aimed at processing relational data, they only infer `struct` types as opposed to enumerations, arrays, switches or unions.

Other researchers have defined grammar induction algorithms that use bottom-up rewriting to search through description space for an optimal description. Many of these techniques, such as RPNI [15] require the availability of both positive and negative examples. In our context, negative examples never exist, making such techniques inapplicable. However, others, such as Stolcke and Omohundro [19] and Hong [10], do not assume the existence of negative examples. These and a number of other systems search through solution space using state-merging rewriting rules. One disadvantage of such techniques is that the initial state is large (representing the entire training data set explicitly) and the search space is enormous. Nevertheless, bottom-up state-merging is often used because it has been difficult to find an effective state-splitting algorithm. Our histogram-based structure-discovery procedure is a new state-splitting algorithm that appears to work well on ad hoc data when coupled with bottom-up rewriting.

7. Conclusions

Managing ad hoc data is a tedious, error-prone and costly enterprise. By augmenting the PADS data processing language and system with an efficient format inference engine, we have effectively cut the generation time for useful data analysis and transformation tools from hours or days to seconds. Now, within moments of receiving a new ad hoc data source, programmers can write complex semi-structured queries to extract information, produce informative graphs of key statistics, convert the data into a format amenable to easy loading into Excel or translate to XML for processing with other standard programming libraries and systems. Systems administrators, computational scientists, financial analysts, industrial data management teams and everyday programmers will all benefit substantially from this new capability to translate dirt into useful shovels for ad hoc data processing.

Acknowledgments

Our work benefited greatly from thoughts and comments from Alex Aiken, David Blei, David Burke, Vikas Kedia, John Launchbury, Chris Ramming, Rob Schapire and the organizers and attendees of the CAGI 2007 Workshop on Grammar Induction.

This material is based upon work supported by DARPA under grant FA8750-07-C-0014 and the NSF under grants 0612147 and 0615062. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of DARPA or the NSF.

References

[1] A. Arasu and H. Garcia-Molina. Extracting structured data from web pages. In *ACM SIGMOD conference*, pp. 337–348, 2003.

[2] D. Burke, K. Fisher, D. Walker, P. White, and K. Q. Zhu. Towards 1-click tool generation with PADS. In *ICML CAGI Workshop*, 2007.

[3] S. Chawathe, H. Garcia-Molina, J. Hammer, K. Ireland, Y. Papakonstantinou, J. D. Ullman, and J. Widom. The TSIMMIS project: Integration of heterogeneous information sources. In *16th Meeting of the Information Processing Society of Japan*, pp. 7–18, 1994.

[4] M. F. Fernández, K. Fisher, R. Gruber, and Y. Mandelbaum. PADX: Querying large-scale ad hoc data with xquery. In *Programming Language Technologies for XML*, 2006.

[5] K. Fisher and R. Gruber. Pads: A domain specific language for processing ad hoc data. In *PLDI* 2005, pp. 295–304.

[6] K. Fisher, Y. Mandelbaum, and D. Walker. The next 700 data description languages. In *ACM POPL* 2006.

[7] E. M. Gold. Language identification in the limit. *Information and Control*, 10(5):447–474, 1967.

[8] P. D. Grünwald. *The Minimum Description Length Principle*. MIT Press, May 2007.

[9] C. D. L. Higuera. Current trends in grammatical inference. *LNCS*, 1876:28–31, 2001.

[10] T. W. Hong and K. L. Clark. Using grammatical inference to automate information extraction from the Web. *LNCS*, 2168:216+, 2001.

[11] Y. Huhtala, J. Kärkkäinen, P. Porkka, and H. Toivonen. TANE: An efficient algorithm for discovering functional and approximate dependencies. *Comput. J.*, 42(2):100–111, 1999.

[12] J. L. Hutchens and M. D. Alder. Finding structure via compression. In D. M. W. Powers, editor, *Proceedings of the Joint Conference on New Methods in Language Processing and Computational Natural Language Learning*, pp. 79–82. ACL, 1998.

[13] J. Lin. Divergence measures based on the Shannon entropy. *IEEE Transactions on Information Theory*, 37(1):145–151, 1991.

[14] Y. Mandelbaum, K. Fisher, D. Walker, M. Fernandez, and A. Gleyzer. PADS/ML: A functional data description language. In *ACM POPL 2007*.

[15] J. Oncina and P. Garcia. Inferring regular languages in polynomial updated time. *Machine Perception and Artificial Intelligence*, 1:29–61, 1992.

[16] PADS project. `http://www.padsproj.org/`, 2007.

[17] V. Raman and J. M. Hellerstein. Potter's wheel: An interactive data cleaning system. In *VLDB 2001*, pp. 381–390.

[18] K. A. Shoens, A. Luniewski, P. M. Schwarz, J. W. Stamos, and I. Joachim Thomas. The Rufus system: Information organization for semi-structured data. In *VLDB 1993*, pp. 97–107.

[19] A. Stolcke and S. Omohundro. Inducing probabilistic grammars by bayesian model merging. In *International Conference on Grammatical Inference*, 1994.