A Context-free Markup Language for Semi-structured Text

Qian Xi
Princeton University
qxi@CS.Princeton.EDU

David Walker
Princeton University
dpw@CS.Princeton.EDU

Abstract

An *ad hoc data format* is any nonstandard, semi-structured data format for which robust data processing tools are not easily available. In this paper, we present ANNE, a new kind of markup language designed to help users generate documentation and data processing tools for ad hoc text data. More specifically, given a new ad hoc data source, an ANNE programmer edits the document to add a number of simple annotations, which serve to specify its syntactic structure. Annotations include elements that specify constants, optional data, alternatives, enumerations, sequences, tabular data, and recursive patterns. The ANNE system uses a combination of user annotations and the raw data itself to extract a context-free grammar from the document. This context-free grammar can then be used to parse the data and transform it into an XML parse tree, which may be viewed through a browser for analysis or debugging purposes. In addition, the ANNE system generates a PADS/ML description [19], which may be saved as lasting documentation of the data format or compiled into a host of useful data processing tools.

In addition to designing and implementing ANNE, we have devised a semantic theory for the core elements of the language. This semantic theory describes the editing process, which translates a raw, unannotated text document into an annotated document, and the grammar extraction process, which generates a context-free grammar from an annotated document. We also present an alternative characterization of system behavior by drawing upon ideas from the field of relevance logic. This secondary characterization, which we call *relevance analysis*, specifies a direct relationship between unannotated documents and the context-free grammars that our system can generate from them. Relevance analysis allows us to prove important theorems concerning the expressiveness and utility of our system.

Categories and Subject Descriptors D.3.m [*Programming languages*]: Miscellaneous

General Terms Languages, Algorithms

Keywords Domain-specific Languages, Tool Generation, Ad Hoc Data, PADS, ANNE

1. Introduction

The world is full of *ad hoc data formats* — those nonstandard, semi-structured data formats for which robust data processing tools are not easily available. Examples of ad hoc data formats include the billions of log files that are generated by web servers, file servers, billing systems, network monitors, content distribution systems, and other applications that require monitoring, debugging or supervision. The data analysts and programmers who find themselves working with ad hoc data formats waste significant amounts of time on various low-level chores like parsing and format translation to extract the valuable information they need from their data. Making these tasks more difficult is the fact that many ad hoc data sets have limited or out-of-date documentation. Moreover, these data formats evolve, so documentation that is up-to-date one month may be deprecated the next.

In the past, two starkly different research communities, the programming languages (PL) community and the machine learning (ML) community, have attempted to apply their technologies to help solve the problem of using ad hoc data files productively.

PL Solutions. In the programming languages community, work has centered on the development of a variety of domain-specific languages that allow data analysts to both *document* and *program with* their ad hoc data. Examples of such languages include DEMETER [18], PACKETTYPES [21], DATASCRIPT [3], PADS [9, 19] and BINPAC [25]. When used for documentation purposes, these languages provide a means to write clear, concise and declarative specifications of a data source's syntax and important semantic properties. Moreover, the fact that the documentation produced is executable (*i.e.,* there exist tools for checking that ad hoc data sources adhere to the format specification given) means that there is an automatic way to check whether documentation is up to date or falling behind. When used for programming support, these languages and their associated compilers provide a means to generate a variety of useful programming libraries for manipulating ad hoc data including parsers, printers, and end-to-end data processing tools.

While these language-based solutions have many useful, even essential features, there is still room for improvement. In particular, producing descriptions of unknown data sources is still a somewhat tedious, time-consuming and error-prone process. For instance, experiments with the PADS system[1] suggest that expert users can create descriptions for many simple line-based system logs in roughly one to two hours, on average, and sometimes less than that. Beginners take substantially longer – often a day or two to read relevant parts of the manual, figure out the syntax, grasp the meaning of various error messages and complete a robust description. For more complicated data sources, and especially for data sources of massive size, the process of creating descriptions becomes substantially more difficult, even for experts. Kathleen Fisher reported that she struggled off-and-on for three weeks in her attempts to describe one particularly massive data file at AT&T that had the unfortunate property of switching formats after a million and a half lines.[2]

ML Solutions. On the other end of the spectrum, the machine learning community has sought to tame ad hoc data sources by developing algorithms for analyzing complex data sources and

PLDI'10, June 5–10, 2010, Toronto, Ontario, Canada.

[1] See table 2, page 10 of earlier work on PADS [11] for anecdotal evidence regarding creation of descriptions for a variety of simple system log formats.

[2] Personal communication, 2008.

either automatically extracting key bits of information from the data sources in question [28, 16, 2, 5] or inferring a grammar that describes them [13, 6, 23, 29, 15, 12, 22, 26, 11].

Whereas the programming languages approaches incur some significant start-up cost, the machine learning approaches usually require less initial work by the programmer. For example, in supervised learning approaches, users must label some subset of their data to indicate the content of interest. Then, various machine learning algorithms can be used to learn the features of the labelled data in order to be able to extract it from its context.Naturally, if a lot of labelling is required of a machine learning approach, then it too has a substantial start-up time, perhaps even more than that of a PL approach. A great deal depends upon the domain in which each approach is used and the specifics of the approach itself, but once a machine learning model has been set up in one domain, it can reduce the start-up time of other learning tasks in the same domain to some degree. Even better, unsupervised approaches require no initial user input. They merely analyze a given dataset, uncover patterns and produce a synthesized grammar. In principal, perfect grammatical inference is impossible [13] but, nevertheless, researchers such as Stolke and Omohundro [29] have shown empirically that one can sometimes synthesize useful grammars using statistical techniques and heuristic search.

While fully automated approaches involving machine learning are usually easy to try, they often suffer from the joint problems of producing *unreliable results* and having those results *hard to understand or analyze*. By unreliable results, we do not mean unsound results — rather we mean that the grammars produced may not be particularly compact or well-organized. Moreover, even when an automated system performs perfectly in a structural sense, it will generate a description teaming with machine-generated names for data subcomponents such as "`Union_237`" or "`Enum_99`." Such descriptions are naturally difficult for people to use and require a human post-processing pass to add semantically meaningful identifiers.

Yet another difficulty with fully automatic grammar induction is that it appears difficult to design a single system that operates well over a broad range of domains. For example, experience with the LEARNPADS system [11] suggests that though it works well for the sorts of systems log files on which it has been tuned, it can easily be thrown off when it encounters data outside its domain. In this latter case, it often generates far more complex, difficult-to-read and difficult-to-use descriptions than a human would. This problem commonly occurs when the data in question depends on some new basic format element – a new sort of date representation, a different way of formatting phone numbers, *etc.* Humans draw upon their worldly experience to identify, modularize, and especially, *name* the new element effectively whereas the LEARNPADS algorithms are often unable to tease apart the details of the new element from the rest of the description and they certainly cannot choose a reasonable name for it. Hence, even though LEARNPADS, and other systems like it, can certainly be improved, the overall approach has some fundamental limitations.

1.1 ANNE: A New Approach

Given the challenges faced by both traditional ML approaches and traditional PL approaches, we have developed a new system, called ANNE, to help improve the productivity of programmers who need to understand, document, analyze and transform ad hoc text data. In particular, we have focused on text data organized in line-by-line or tabular formats, as this is the most common sort of layout in systems log files and a variety of other domains. However, in principle, our techniques are sufficiently general to handle any data format that can be described as a context-free grammar.

Rather than requiring programmers to write complete data descriptions, as in the conventional PL approach, or simply accepting the unvarnished results of a fully automatic, heuristic algorithm, as in the conventional ML approach, ANNE combines ideas from both communities in search of the best of all worlds. To be more specific, the process of generating a description for a text document begins by having the user edit the text itself to add annotations that help describe it. These annotations, and the surrounding unannotated text, are used to generate a human-readable PADS description. The PADS description may then be fed through the PADS compiler, generating a host of useful artifacts ranging from programming libraries for parsing, printing and traversal to end-to-end tools for format-conversion, querying, and simple statistical analysis. In addition to generating a PADS description, the system will translate the text data into a structured XML parse tree. The XML parse tree can be viewed through a browser, analyzed and used for debugging purposes. In a word, with help of programmer annotations, ANNE will translate the original text data into a set of data description end products.

The annotations that constitute the ANNE language perform a number of different roles including each of the following:

- associating user-friendly names with bits of text or descriptions generated from sub-documents
- defining atomic abstractions such dates, ip addresses, times, and urls using regular expressions,
- identifying sequences, constants and enumerations,
- delimiting tabular data and its headers,
- relating different variants of a field to one another, and
- introducing recursive descriptions.

Together this set of annotations is both convenient and powerful, and overall, the benefits of this new approach are numerous.

First, as in the PL approaches, ANNE provides the user with *great control* over the resulting description, when they want it. The user can introduce meaningful, human-readable names, identify the correct atomic abstractions, and shape key parts of the grammar however they desire.

Second, again as in the PL approaches, ANNE is extremely *powerful*. For example, ANNE easily supports tables and recursive grammars even though identifying tables in text data is a difficult machine learning challenge [22, 26, 17] and learning context-free grammars is even harder than the already-hard challenge of learning regular expressions. LEARNPADS supports neither of these features.

Third, as in the ML approaches, *less work is required* of the programmer. Importantly, unannotated text in the surrounding context is used to "fill in the blanks" left in a description using various default mechanisms. This means that the programmer does not have to, and is not encouraged to, write the entire description. Hence, in some respects, ANNE resembles a supervised learning approach except that rather than using simple labels to identify important data, ANNE uses more powerful, higher-level commands.

Fourth, the annotation language has small number of constructs in it and, perhaps more subjectively, we find it is relatively *easy to use*. Ease of use comes from the fact that programmers can stare directly at the text they are interested in and directly wrap an annotation around it to capture it. There is no counting of fields or the possibility of off-by-one errors. In this way, the system supports a "what-you-annotate-is-what-you-get" style of interaction. The XML-generation tool provides immediate feedback and facilitates debugging.

In addition to designing and implementing ANNE, we have developed an elegant theory to explain its semantics. This theory is based around IDEALIZED ANNE (IA for short), an idealized

```
207.136.97.49 - - [15/Oct/1997:18:46:51 -0700] "GET /turkey/amnty1.gif HTTP/1.0" 200 3013
207.136.97.49 - - [15/Oct/1997:18:46:51 -0700] "GET /turkey/clear.gif HTTP/1.0" 200 76
polux.entelchile.net - - [15/Oct/1997:21:02:07 -0700] "GET /latinam/spoeadp.html HTTP/1.0" 200 8540
152.163.207.138 - - [15/Oct/1997:19:06:03 -0700] "GET /images/spot5.gif HTTP/1.0" 304 -
ip160.ridgewood.nj.pub-ip.psi.net - - [15/Oct/1997:23:45:48 -0700] "GET /whatsnew.html HTTP/1.0" 404 168
ppp31.igc.org - amnesty [16/Oct/1997:08:40:11 -0700] "GET /members/afreport.html HTTP/1.0" 200 450
...
```

Figure 1. Excerpt from the web server log `ai.3000`.

core annotation calculus. The semantics of the IA programming process is given by a relation between annotated and unannotated documents and the semantics of IA itself is given by a function that generates context-free grammars from annotated documents.

In order to understand the capabilities of IA in greater depth, we prove theorems that characterize the kinds of grammars that can be generated by our system. In doing so, we introduce an interesting new set of relations, inspired by relevance logic [1], that more precisely define the relationship between generated grammars and the data they describe. We use these relations to prove important theorems concerning the expressiveness of our system.

Contributions. To summarize, this paper makes a number of major contributions:

- We introduce a highly practical, new technique for generation of format specifications from text data. We illustrate its use on a number of examples and evaluate its effectiveness.
- We develop an idealized, core annotation calculus that captures the key elements of our design. We give a semantics to the calculus to describe how ANNE programming and grammar extraction works.
- We introduce a secondary characterization of ANNE based on concepts drawn from relevance logic. We use this secondary characterization to analyze the expressive power of our system.
- We have implemented the system and combined it with the PADS language and compiler, allowing users of our system to easily generate useable documentation along with a suite of programming libraries and end-to-end data processing tools.

In the following section of the paper, we explain our language design and how to use it in more detail. In section 3, we develop the syntax and semantics IDEALIZED ANNE. In section 4, we introduce our relevance analysis and use it to prove key theorems about the expressiveness of IDEALIZED ANNE. In Section 5, we comment further on our experiences using ANNE to generate format specifications and evaluate its effectiveness relative to both manual construction of PADS formats and the grammar induction system developed in earlier work [11]. Section 6 describes related work and Section 7 concludes.

2. ANNE by Example

ANNE is a language and system for deriving grammatical specifications and text processing tools directly from example text files. In this section, we will illustrate the basic functionality of the language through a number of examples.

2.1 A Web Server Log

Our first example involves the problem of processing a web server log. We will be highlighting text added to the file using a grey background. The log itself is presented in Figure 1. System administrators query, transform and analyze logs just like this (and hundreds of variants thereof) as part of their day-to-day job of assessing the health and security of the systems they oversee.

The Preamble. The first step in processing any log like this is to edit the file at the top to add the following lines.

```
!#
#include "systems.config"
!#
```

This step adds the preamble defined by the file `systems.config`, which is presented in Figure 2. A config file such as this is composed of a series of lines with one regular expression definition per line. Each line begins with either `def` or `exp` and is followed by a name and a regular expression. Those lines beginning with `exp` will *export* the named regular expression so it can be used in describing formats. Those lines beginning with `def` provide a *local definition* for the name. A local definition can be used in subsequent `def`s or `exp`s but is not in scope in the rest of the file. Comment lines begin with a `#` symbol. The `systems.config` file has been specially designed for system administrators dealing with log files. Each new domain can create its own set of common, reusable data definitions to speed up data format construction.

Introducing Nonterminals. The next step is to identify, describe and give names to elements of interest in the file. For instance, a sysadmin might start with the first line after the preamble and begin to edit it as follows (though the annotation process can start at any place in the file that happens to be convenient). To format lines within the boundaries of the narrow `sigplanconf` style, we will break lines where necessary with a slash and continue them indented two spaces on the next line.

```
{Record: 207.136.97.49 - - \
  [15/Oct/1997:18:46:51 -0700] \
  "GET /turkey/amnty1.gif HTTP/1.0" 200 3013 }
```

Intuitively, the simple annotation {`Name: ...`} begins the process of defining a scannerless context-free grammar. Note that if braces "{" and "}" already appear in the file, a command line switch can alter the bracketing syntax. In this case, the portion of the grammar so-defined involves a single nonterminal named `Record`. Moreover, since there are no other annotations to guide grammar generation, the system uses a simple default rule to generate the right-hand side – it assumes the desired right-hand side is a simple concatenation of basic tokens derived by running a default lexer over the data enclosed in braces.

```
Record ::= Num '.' Num WS '-' WS '-' WS '[' ...
```

In order to maintain predictability and ease-of-use, the set of default tokens has been kept to the barest minimum. It includes numbers (`Num` – integer or floating point), punctuation symbols (*e.g.,* `'['` or `'.'` or `']'`, *etc.*), words (`Word`), and whitespace (`WS`). The default tokenization scheme can be overridden by extending the preamble with new programmer-defined tokens expressed as regular expressions. However, doing so changes the tokenization globally for the entire file, which is not particularly useful here.

Using the Preamble. Instead of overriding the preamble, we will take advantage of some of the regular expression definitions in `systems.config` to further refine the grammar for the `Record` nonterminal:

```
{Record: {IP<: 207.136.97.49 } - - \
  [ {Date<: 15/Oct/1997 } : {Time<: 18:46:51 -0700 } ] \
  "GET /turkey/amnty1.gif HTTP/1.0" 200 3013}
```

```
#   Name  Regular Expression
def trip  [0-9][0-9][0-9]\|[0-9][0-9]\|[0-9]
def db    [0-9][0-9]
def zone  [+-][0-1][0-9]00
def ampm  am\|AM\|pm\|PM
...
exp Time  {db}:{db}:{db}\([ ]*{ampm}\)?\([ \t]+{zone}\)?
exp IP    {trip}\.{trip}\.{trip}\.{trip}
...
```

Figure 2. Excerpt from `systems.config`

Above, we used several annotations with the form {`Name<: ...`} to introduce regular expressions named `Name`. For instance, we identified an ip address (`IP`), a date (`Date`) and a time (`Time`). All of these named regular expressions were introduced in the preamble (by including their definitions from `systems.config`). After this refinement, our generated grammar has the following form.

```
IP ::= ...
Date ::= ...
Time ::= ...
Record ::= IP WS '-' WS '-' WS '[' Date ':' Time ']' ...
```

The right-hand sides of `IP`, `Date` and `Time` will be regular expressions defined by the preamble.

Annotations for Termination Symbols. The next refinement of the grammar involves dealing with the string `"GET /turkey/amnty1.gif HTTP/1.0"`. In many applications, the internal structure of this string might be irrelevant. If this is the case, one could simply wrap the contents of the string with an annotation of the form {`Name>: ...`}. In this case, `Name` introduces another nonterminal into the grammar and the greater-than sign indicates that the extent of nonterminal's reach is defined by a terminating character – the character that follows the close brace. Here is the annotation used in context:

```
{Record:{IP<:207.136.97.49} - - \
  [{Date<:15/Oct/1997}:{Time<:18:46:51 -0700}] \
  " {Message>: GET /turkey/amnty1.gif HTTP/1.0 } " \
  200 3013}
...
```

In the text above, the > annotation introduces the Message nonterminal and its extent is terminated by a quotation symbol. Such a token can easily be defined by a regular expression, but experience with the PADS data description language [9] confirms that this idiom is extremely common in all kinds of log files. Building in this shorthand is a nice programmer convenience.

Generating XML and Debugging Results. At this point, the "programming burden" has been minimal. It consists of including the preamble in the data source and writing five simple annotations, which mainly involve naming key parts of the data. All-in-all the job of describing the data may have taken a minute or two. To debug the work, one can invoke the ANNE compiler, which will generate a number of artifacts, including a PADS description and an XML parse tree of the data. Viewing the XML through a browser, as shown in the screen shot in Figure 3, reveals that the grammar generated so far only covers a subset of the data in the file – colored lines indicate lines covered by the generated grammar and greyed out lines indicate lines that are uncovered. A quick examination of the first greyed out line indicates that there is more variation in the data file than had been apparent at first glance. Fortunately, generating a complete cover is relatively easy with just a few more annotations.

Introducing Alternatives. Alternatives can be introduced into the grammar in several ways. The simplest way is merely to use a particular nonterminal name repeatedly. We illustrate this technique below by using the nonterminal `Size` twice, once around an integer (which represents the normal case – the number of bytes returned by the server is reported properly) and once around `"-"` (which represents the nonstandard case of no data available).

```
{Record:{IP<:207.136.97.49} - - \
  [{Date<:15/Oct/1997}:{Time<:18:46:51 -0700}] \
  "{Message>:GET /turkey/amnty1.gif HTTP/1.0}" 200 \
  {Size: 3013 } }
...
152.163.207.138 - - \
  [15/Oct/1997:19:06:03 -0700] \
  "GET /images/spot5.gif HTTP/1.0" 304 {Size: - }
```

Such annotations extend the grammar with a union of two or more options:

```
Size ::= Num + '-'
Record ::= IP WS '-' WS '-' WS ... Size
```

An alternative technique is to use a collection of annotations of the form {`Name/Name1: ...`} and {`Name/Name2: ...`} and {`Name/Name3: ...`}, *etc.* as follows.

```
{Record:{IP<:207.136.97.49} - - \
  [{Date<:15/Oct/1997}:{Time<:18:46:51 -0700}] \
  "{Message>:GET /turkey/amnty1.gif HTTP/1.0}" 200 \
  {Size/S: 3013 } }
...
152.163.207.138 - - \
  [15/Oct/1997:19:06:03 -0700] \
  "GET /images/spot5.gif HTTP/1.0" 304 {Size/Dash: - }
```

This technique names the alternatives and generates the following equivalent grammar.

```
S ::= Num
Dash ::= '-'
Size ::= S + Dash
Record ::= IP WS '-' WS '-' WS ... Size
```

One reason to use the more verbose form with named alternatives is that it will generate a nicer PADS/ML description for the user – one that is an ML-style description and uses data type descriptions with well-named constructors (See Section 2.3).

There is one other detail to consider when it comes to alternatives: the most concise grammar is sometimes one in which alternatives overlap. PADS, and many other systems, use prioritized choice to disambiguate between overlapping alternatives. In ANNE, priorities may be specified as integers using a syntax with the form {`Name1/Name2[priority]: ...`}.

Finishing up the Web Log Example. With just a few more annotations, the web log annotation job is complete. In total, it was necessary to add the preamble and annotate four lines of text. Three of the four lines only required annotating one bit of data. The whole process might have taken five minutes. The resulting generated grammar is presented in Figure 4. Notice that by default, the top-level nonterminal symbol is `Source` and that the top-level grammatical rule is as follows.

```
Source ::= Record (NL Record)*
```

In the line above, `NL` is the newline character and the asterisk is the familiar Kleene star. In other words, the entire source is a sequence of `Record`s separated by newline characters. In general, a programmer can create annotations for any number of top-level items, which may be line-by-line descriptions or tables, and ANNE will produce a top-level grammar with the form

```
Source ::= (Item1+...+Itemk) (NL (Item1+...+Itemk))*
```

2.2 Additional Language Features

The web log document discussed in the previous subsection is one example of the sort of ad hoc data source that ANNE was designed

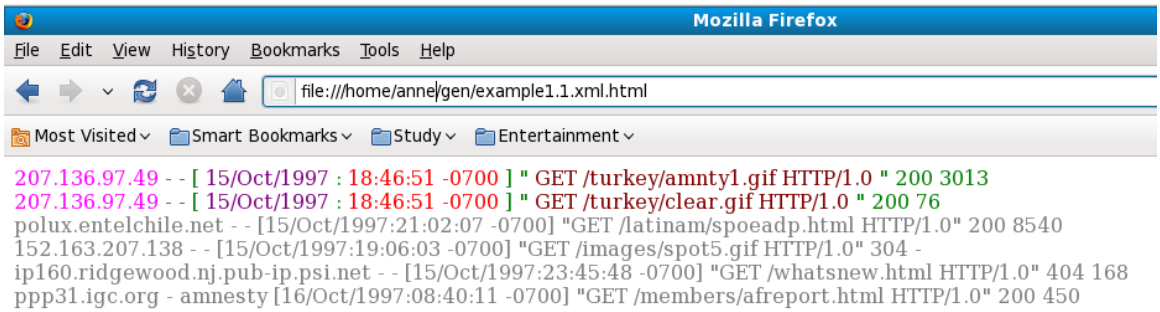

Figure 3. View of generated XML after partial data description.

```
S ::= Num
Dash ::= '-'
Size ::= S + Dash
Sender ::= IP + Hostname
ID ::= '-' + Word
Record ::= Sender WS '-' WS ID WS '[' Date ':' Time ']'
           WS '\"' Message '\"' WS Num WS Size
Source ::= Record (NL Record)*
```

Figure 4. Generated Grammar. Regular expression definitions of IP, Hostname, Date, Time, and Message are omitted.

to service. It used a good number of different kinds of annotations, but there are a number of other features of the language, which we describe more briefly in this section.

Repetition. The web log had an implicit, repeated structure at the top-level, but no internal repetition. Many other ad hoc data files do. To generate a grammar with a repeated sequence of items, one may use a starred annotation as in the following pipe-separated number sequence, which is drawn from one of our data sources:

```
{Record*[|]: 9152271|9152271|1|0|0|0|0|... }
```

In the annotation above, `Record` names a part of a grammar involving a sequence of items in which each item is separated by a `'|'` symbol. By default, if there are no further annotations, the record element structure will be any character sequence not including the separator.

```
NoBar ::= ...
Record ::= (NoBar ('|' NoBar)*)?
```

Alternatively, the record elements can be specified exactly using the syntax {`Name1/Name2*[sep]: ...`}, as in the following example.

```
{Record/Elem*[|]: 9152271| {Elem: 9152271 } |1| \
  0|0|0|0|... }
```

The separator (defined in square brackets prior to the colon) is optional and, if desired, the programmer can add an optional terminator string.

Optional data. Optional data occurs often. The annotation {`Name?: ...`} defines `Name` to either be formatted as the grammar generated by "`...`" or the empty string.

Constants and Enumerations. In order to specify a nonterminal that has a constant value, as opposed to generating a more liberal grammar, one can uses an equality annotation {`Name=: ...`} or its unnamed variant {`=: ...`}. Sometimes the nonterminal may contain a small number of constant values instead of a single one, and then, to generate a grammar involving the list of constants that

```
{E#h:Name  GP  Goals Assists Points +/-
Jason Blake,   78   25   38   63   -2
Alexei Ponikarovsky,   82   23   38   61   6
...}
Name  GP  Goals  Assists Points +/-
Alexander Ovechkin,   79   56   54   110  10
Nicklas Backstrom,   82   22   66   88   3
...
```

Figure 5. Fragment of an annotated document containing NHL player statistics from the 2008-2009 season, one table per team.

```
ptype IP = Pstring_ME(...)
ptype Hostname = Pstring_ME(...)
...
ptype Size =
  S of Num
| Dash of '-'
ptype Sender =
  IP of IP
| Hostname of Hostname
ptype ID =
  '-'
| Num
ptype Record = Sender * WS * '-' * WS * ID * WS * '['
    * Date * ':' * Time * ']' * WS * '\"' * Message
    * '\"' * WS * Int * Size
ptype Source = Record plist(No_sep, No_term)
```

Figure 6. PADS/ML description generated from annotated web log. Regular expression definitions of IP, Hostname, *etc.* are omitted.

actually appear in the file, one can use an enumeration annotation. An enumeration is written as {`Name//enum>: ...`}. It generates an initial grammar in the same way that our termination symbol specification generates a grammar (by looking for a terminating symbol). That initial grammar is used to parse the document at hand and collect all strings that match the spec in the document. The final grammar is one defined using the instances that match.

Tables. The last important feature of ANNE involves tables. Even though tables can be specified using concatenation and Kleene star, it is worthwhile building special support for them as they appear frequently. Identifying tables is a useful programmer convenience and also makes it easier to generate a good query interface for the data.

Figure 5 shows a small portion of a document containing a series of tables describing NHL player statistics, with one table per NHL team. Hockey aficionados use such data regularly to compute player values and argue important points such as "Is Crosby better than Ovechkin?" or "Was John Ferguson Junior the worst Leafs GM since the early 80s?" Tables such as the ones displayed here often have a header row followed by some number of rows with

a fixed number of columns. Using ANNE, deriving a grammar for such a table simply involves using one of the hash annotations, either {`Name#: ...`} or {`Name#h: ...`}. The `h` in the second variant indicates that the table has a header row that varies in structure from the table data. The number and structure of the columns is determined by counting the number of each sort of token in every line. If some token t appears k times in every line then there are $k+1$ columns and t serves as the separator between columns. If more than one token satisfies this property, one such token is selected heuristically (tokens that serve frequently as separators such as tab, comma, and vertical bar are prioritized). However, the programmer is free to specify the separator in question explicitly using square braces as in the Kleene star annotations.

Assertions. In a number of situations, and particularly when data is recursive, it is useful for a programmer to be able to assert that some part of the data satisfies a nonterminal definition without going to the trouble of annotating all its subparts. We allow such assertions through annotations with the form {`Name!: ...`}. For example, given a simple string of parentheses such as "`((((()))))`", the simplest way to annotate the data is as follows.

```
{Parens?: ( {Parens!: ((()))} ) }
```

An annotation associates the data enclosed in braces with a nonterminal name, but it doesn't generate a grammar rule for the nonterminal. Thus, the above annotation scheme will give rise to the following grammar.

```
Parens ::= ('(' Parens ')')?
```

2.3 Generating PADS Descriptions

In the previous subsection, we explained the semantics of the ANNE language by presenting the context-free grammars that are generated from each annotation scheme. These context-free grammars are used to parse the data source and generate an XML parse tree that can be viewed through a browser or processed using any one of a number of XML-based tools, languages or libraries.

In addition to generating structured XML, an ANNE mark-up will also generate a PADS description [9, 10, 19]. The PADS description language uses augmented type declarations to describe the syntactic structure of a document as well as the programming language data structures one generates by parsing the document. Figure 6 shows the PADS description generated from the annotated web log presented in Figure 4.

A PADS description such as the one in Figure 6 can serve as permanent executable documentation for the data source. It can also be used to generate a variety of libraries such as parsers, printers, and traversal functions for processing other data sources with the same format. Finally, the PADS compiler can link generated libraries against various generic tools including a query engine [8], data synchronization engine [7], and various format translators. Consequently, while using ANNE is a quick and simple process, the result of this minimal bit of labour is an enduring piece of human-readable documentation (the PADS description) and a valuable collection of reusable tools.

3. IDEALIZED ANNE

The previous section introduced ANNE through a series of examples, but did not answer any general questions about the principles involved in the language design: What do these annotations mean? What grammars do they generate? When do we have sufficient data to generate a particular grammar? In this section, we make some initial headway towards answering these more general questions by defining the syntax and semantics of IDEALIZED ANNE (IA), a simplified variant of the full ANNE language that encapsulates its essential features. IA doesn't reflect features of optional data, constants and enumerations, tables, while most of these features can be represented by essential features in IA.

Regular Expressions:
$b \quad ::= \quad \epsilon \mid c \mid b.b \mid \cdots$

Annotated Documents:
$$\begin{array}{rcl} ad & ::= & v \mid ad_1 ad_2 ... ad_n \mid \{ad\} \mid \{[b] : v\} \mid \{A : ad\} \\ & \mid & \{A/inl : ad\} \mid \{A/inr : ad\} \\ & \mid & \{A/A_{elem}* : ad\} \mid \{A/A_{elem}*_0 :\} \\ & \mid & \{A! : ad\} \end{array}$$

Figure 7. IDEALIZED ANNE documents.

Nonterminal Clauses:
$s \quad ::= \quad b \mid A \mid s_1 \cdot s_2 \cdot ... \cdot s_n$

Nonterminal Right-hand Sides:
$r \quad ::= \quad s \mid s_1 + s_2 \mid ? + s_2 \mid s_1 + ? \mid A* \mid A*_0$

Nonterminal Definitions:
$\mathsf{G} \quad ::= \quad [] \mid \mathsf{G}[A = r]$

Grammars:
$gram \quad ::= \quad (A, \mathsf{G})$

Figure 8. Grammar Syntax.

3.1 IDEALIZED ANNE Syntax and Programming

In the following formal work, we will let c range over characters while v and w range over strings (our *unannotated documents*). We let "" denote the empty string and $v_1 v_2$ denote the concatenation of two strings. Meta-variable A ranges over nonterminal names and b ranges over regular expressions. We write $\mathcal{L}(b)$ to denote the language of regular expression b. Regular expressions with an empty language are prohibited.

Syntax. The syntax of *annotated documents* is defined in Figure 7. An annotated document may either be unannotated (v) or a sequence of annotated documents ($ad_1 ad_2 ... ad_n$). Other annotations include the following.

- $\{ad\}$ identifies a sub-document
- $\{[b] : v\}$ identifies the data v as inhabiting the language of regular expression b.
- $\{A : ad\}$ assigns a nonterminal A to the format inferred from annotated sub-document ad.
- $\{A/inl : ad\}$ and $\{A/inr : ad\}$ introduce the left- and right-hand elements of a union respectively
- $\{A/A_{elem}* : ad\}$ introduces a repetition named A with elements named A_{elem}. Sub-document ad is used to infer A_{elem}. $\{A/A_{elem}*_0 :\}$ is a related annotation, added to the calculus to simplify certain inductive proofs. It need not be used by programmers. It's semantically equivalent to ϵ.
- $\{A! : ad\}$ claims that the data ad has the format given by A without checking. Sub-document ad may be used to infer other parts of the grammar.

The programming process. In order to use IDEALIZED ANNE, a programmer need simply apply some collection of annotations to their data. This programming process is formalized by a judgement written $v \rightarrow ad$, which relates an unannotated document v to any

$\boxed{v \to ad}$

$$\frac{}{v \to v}\ (a\text{-}none)$$

$$\frac{v_i \to ad_i \qquad i = 1..n}{v_1 v_2 ... v_n \to ad_1 ad_2 ... ad_n}\ (a\text{-}con)$$

$$\frac{v \to ad}{v \to \{ad\}}\ (a\text{-}group) \qquad \frac{v \in \mathcal{L}(b)}{v \to \{[b] : v\}}\ (a\text{-}re)$$

$$\frac{v \to ad}{v \to \{A : ad\}}\ (a\text{-}name)$$

$$\frac{v \to ad}{v \to \{A/inl : ad\}}\ (a\text{-}inl) \qquad \frac{v \to ad}{v \to \{A/inr : ad\}}\ (a\text{-}inr)$$

$$\frac{v \to ad}{v \to \{A/A_{elem}* : ad\}}\ (a\text{-}rep)$$

$$\frac{}{\text{“”} \to \{A/A_{elem}*_0 :\}}\ (a\text{-}rep\text{-}empty)$$

$$\frac{v \to ad}{v \to \{A! : ad\}}\ (a\text{-}assert)$$

Figure 9. Document annotation.

$\boxed{\mathsf{G} \vdash^c v \in s}$

$$\frac{v \in \mathcal{L}(b)}{\mathsf{G} \vdash^c v \in b}\ (g\text{-}re) \qquad \frac{\mathsf{G}(A) = r \quad \mathsf{G} \vdash v \in r}{\mathsf{G} \vdash^c v \in A}\ (g\text{-}name)$$

$$\frac{\mathsf{G} \vdash^c v_i \in s_i \qquad i = 1..n}{\mathsf{G} \vdash^c v_1 v_2 ... v_n \in s_1 \cdot s_2 \cdot ... \cdot s_n}\ (g\text{-}con)$$

$\boxed{\mathsf{G} \vdash v \in r}$

$$\frac{\mathsf{G} \vdash^c v \in s}{\mathsf{G} \vdash v \in s}\ (g\text{-}clause)$$

$$\frac{\mathsf{G} \vdash^c v_1 \in s_1}{\mathsf{G} \vdash v_1 \in s_1 + ?}\ (g\text{-}sum1) \qquad \frac{\mathsf{G} \vdash^c v_2 \in s_2}{\mathsf{G} \vdash v_2 \in ? + s_2}\ (g\text{-}sum2)$$

$$\frac{\mathsf{G} \vdash^c v_1 \in s_1}{\mathsf{G} \vdash v_1 \in s_1 + s_2}\ (g\text{-}sum3) \qquad \frac{\mathsf{G} \vdash^c v_2 \in s_2}{\mathsf{G} \vdash v_2 \in s_1 + s_2}\ (g\text{-}sum4)$$

$$\frac{\mathsf{G} \vdash^c v_i \in A \qquad i = 1..n}{\mathsf{G} \vdash v_1 v_2 ... v_n \in A*}\ (g\text{-}rep) \qquad \frac{}{\mathsf{G} \vdash \text{“”} \in A*_0}\ (g\text{-}rep\text{-}emp)$$

$\boxed{\vdash v \in gram}$

$$\frac{\mathsf{G} \vdash v \in A}{\vdash v \in (A, \mathsf{G})}\ (g\text{-}gram)$$

Figure 10. Semantics of Grammars.

one of its annotated variants ad. Figure 9 presents the annotation rules. For instance, rule (*a-none*) says that annotating a document can involve doing nothing. Rule (*a-con*) says that annotating a document can involve subdividing the document into arbitrarily many subpieces, each of which is recursively annotated. All of the other rules simply wrap one of the particular annotation forms around a sub-document (usually after recursively annotating the sub-document, except for repetitions and assertions).

3.2 Grammars

Syntax. The purpose of IDEALIZED ANNE is to generate grammars of the form given in Figure 8. Reading from the bottom of the figure towards the top, one sees that a grammar is a pair of a start nonterminal A and finite partial map G from nonterminal names to right-hand sides. A right-hand side may be a clause s, a union of clauses $(s_1 + s_2)$ or a repetition of some nonterminal $A*$. A right-hand side may also be one of three *partial* right-hand sides: $(? + s)$ or $(s + ?)$ or $A*_0$ (other right-hand sides are called *complete*). Intuitively, the $?$ symbol represents a missing part of the grammar, and both $?$ and $*_0$ symbols indicate that no underlying data is recognized by that part of the grammar. Partial right-hand sides appear during the course of constructing a grammar (or inductively in the midst of our proofs), but should not appear in any final result. A clause (s) is either a regular expression (b), a nonterminal (A), or a sequence of clauses $s_1 \ldots s_n$.

Semantics. The semantics of grammars is defined by the judgement $\vdash v \in gram$, which depends upon judgements $\mathsf{G} \vdash v \in r$ and $\mathsf{G} \vdash^c v \in s$. Intuitively, the latter two may be read "string v is in the language of r (or s) when nonterminals are defined by G." The rules defining this judgement are presented in Figure 10. Many of these rules are self-explanatory. For instance, rule *g-name* states that a string is in the language of A provided it is in the language of its defining right-hand side. In rule *g-rep*, a sequence of strings is recognized. In a slight abuse of notation, we allow n to be 0, in which case we interpret the rule to say that the repetition recognizes the empty string.

The only unusual rules are the rules for the partial right-hand sides. The rules for partial unions $s + ?$ and $? + s$ state a value is in their language provided it is in the known alternative s. The rule for partial repetitions $A*_0$ states that the empty string is in its language.

3.3 Grammar Extraction

Once a document has been annotated, the IDEALIZED ANNE run time system can extract a grammar from it. This extraction process is implemented by recursively traversing the annotated document and extracting partial grammars from the subpieces. A final grammar results from *fusing* (*i.e.,* combining in a special way) collections of partial grammars.

We will define the fusion relation (written $\mathsf{G}_1 \oplus \mathsf{G}_2$) in a moment, but first we will direct the reader's attention to Figure 11, which presents the grammar extraction function itself. This function, written $ad \rightsquigarrow (s, \mathsf{G})$, analyzes annotated document ad and generates a clause s as well as partial grammar G to describe it.

The first rule in the extraction definition (*p-none*) explains how unannotated data will generate a description. This occurs by finding a sequence of regular expressions that matches the data. These regular expressions are drawn from the *default set* $\mathcal{D}$. The default set for our implementation contains basic tokens such as numbers, words, whitespace and punctuation symbols. The choice of defaults is unimportant in the theory. Like Lexer, this tokenization procedure is deterministic, which means, for any string, IDEALIZED

$\boxed{ad \rightsquigarrow (s, \mathsf{G})}$

$$\frac{v_i \in \mathcal{L}(b_i) \qquad b_i \in \mathcal{D} \qquad i = 1..n}{v_1 v_2 ... v_n \rightsquigarrow (b_1 \cdot b_2 \cdot ... \cdot b_n, [])} \ (p\text{-}none)$$

$$\frac{ad_i \rightsquigarrow (s_i, \mathsf{G}_i) \qquad i = 1..n}{ad_1 ad_2 ... ad_n \rightsquigarrow (s_1 \cdot s_2 \cdot ... \cdot s_n, \mathsf{G}_1 \oplus \mathsf{G}_2 \oplus ... \oplus \mathsf{G}_n)} \ (p\text{-}con)$$

$$\frac{ad \rightsquigarrow (s, \mathsf{G})}{\{ad\} \rightsquigarrow (s, \mathsf{G})} \ (p\text{-}group) \qquad \frac{v \in \mathcal{L}(b)}{\{[b] : v\} \rightsquigarrow (b, [])} \ (p\text{-}re)$$

$$\frac{ad \rightsquigarrow (s, \mathsf{G})}{\{A : ad\} \rightsquigarrow (A, \mathsf{G} \oplus [A = s])} \ (p\text{-}name)$$

$$\frac{ad_1 \rightsquigarrow (s_1, \mathsf{G}_1)}{\{A/inl : ad_1\} \rightsquigarrow (A, \mathsf{G}_1 \oplus [A = s_1 + ?])} \ (p\text{-}inl)$$

$$\frac{ad_2 \rightsquigarrow (s_2, \mathsf{G}_2)}{\{A/inr : ad_2\} \rightsquigarrow (A, \mathsf{G}_2 \oplus [A = ? + s_2])} \ (p\text{-}inr)$$

$$\frac{ad \rightsquigarrow (s, \mathsf{G})}{\{A/A_{elem}* : ad\} \rightsquigarrow (A, \mathsf{G} \oplus [A = A_{elem}*])} \ (p\text{-}rep)$$

$$\frac{}{\{A/A_{elem}*_0 :\} \rightsquigarrow (A, [A = A_{elem}*_0])} \ (p\text{-}rep\text{-}emp)$$

$$\frac{ad \rightsquigarrow (s, \mathsf{G})}{\{A! : ad\} \rightsquigarrow (A, \mathsf{G})} \ (p\text{-}assert)$$

Figure 11. Grammar Extraction.

ANNE will produce exactly 1 concatenation of regular expressions $b_1, ..., b_k$.

The next rule (*p-con*) explains how to handle a sequence of annotated sub-documents. In this case, each sub-document is analyzed recursively, producing a clause and a right-hand side. The result is a concatenation of clauses and a grammar formed by fusing together the generated subgrammars.

Many of the other rules should now be relatively self-explanatory. However, the reader should take note of rules (*p-inl*) and (*p-inr*), as these rules are primary points where partial grammars are generated. Notice in particular that rule (*p-inl*) infers the shape of the left-hand side of a union from its sub-document, but has no information about the right-hand side and hence leaves $?$ in its place. Rule (*p-inr*) behaves in a complementary fashion.

In addition, rules (*p-rep*) and (*p-assert*) are other 2 rules that should raise the reader's attention. Rule (*p-rep*) infers the top-level structure of a repetition and fuses $A = A_{elem}*$ with partial grammar G, whereas A_{elem} can be defined by any annotation anywhere, inside its sub-document or from other documents. The fusion of $A = A_{elem}*$ with G or other grammar parts defined elsewhere will bring the definition of A together with the definition of A_{elem} to create a final grammar of the repetition. For instance, recall the example in Section 2.2, A_{elem}, in this case called $Elem$, is defined relatively deeply within the string that makes up the array. Rule (*p-assert*) discards the right-hand side generated from its sub-document and instead uses the specified grammar symbol A. It doesn't generate any grammar rule, which means, the rule for nonterminal A is generated from elsewhere.

Grammar fusion. Intuitively, fusing two right-hand sides together involves eliminating the $?$ symbols and replacing them with real grammar parts. For instance, fusing $(s_1 + ?)$ with $(? + s_2)$ results in $(s_1 + s_2)$. Fusing two grammars together involves taking the union of the disjoint grammar parts and fusing together the right-hand sides of the overlapping grammar parts. More formally, the right-hand side fusion relation $r_1 \oplus r_2$ is defined as the symmetric closure of the following rules.

$$\begin{array}{lcl} r \oplus r & = & r \\ (s_1 + ?) \oplus (? + s_2) & = & s_1 + s_2 \\ (s_1 + s_2) \oplus (s_1 + ?) & = & s_1 + s_2 \\ (s_1 + s_2) \oplus (? + s_2) & = & s_1 + s_2 \\ A* \oplus A*_0 & = & A* \end{array}$$

We have chosen the weakest possible fusion operation – basing it upon syntactic equality of grammars. A stronger fusion operation could be based upon semantic equality of grammars, but this is an undecidable problem for context-free grammars. We chose the weak fusion operation because of its simplicity, predictability, ease of understanding and implementation, and, importantly, because it is effective in practice.

Given the right-hand side fusion, we define the fusion of two grammars $\mathsf{G}_1 \oplus \mathsf{G}_2$ as follows. $\mathsf{D}(\mathsf{G})$ denotes the domain of grammar G (*i.e.,* the set of defined nonterminals).

$$\mathsf{G}_1 \oplus \mathsf{G}_2(A) = \begin{cases} \mathsf{G}_1(A) & \text{if } A \in \mathsf{D}(\mathsf{G}_1) \text{ and } A \notin \mathsf{D}(\mathsf{G}_2) \\ \mathsf{G}_2(A) & \text{if } A \in \mathsf{D}(\mathsf{G}_2) \text{ and } A \notin \mathsf{D}(\mathsf{G}_1) \\ \mathsf{G}_1(A) \oplus \mathsf{G}_2(A) & \text{if } A \in \mathsf{D}(\mathsf{G}_1) \text{ and } A \in \mathsf{D}(\mathsf{G}_2) \end{cases}$$

Finally, the fusion of two grammars with the same start symbol, $(A, \mathsf{G}_1) \oplus (A, \mathsf{G}_2)$, is defined to be $(A, \mathsf{G}_1 \oplus \mathsf{G}_2)$.

Formalism vs. Implementation. The observant reader will notice that the formal system requires a few more annotations be made explicit than the implemented system. In other words, for the sake of convenience and brevity, the implemented system performs some simple "annotation inference" for the user in various situations. For example, consider the following text fragment.

```
{Foo: 123 }
{Foo: cat }
```

In formal system, inconsistent right-hand sides, number vs. word, are rejected by the definition of fusion operator. In the implemented system, however, rather than fail and ask the programmer to add more annotations, we infer `inl` and `inr` union annotations as follows:

```
{Foo/inl: 123 }
{Foo/inr: cat }
```

With these additional annotations in place, the formal system (and the implementation) will successfully extract a union grammar.

Another difference between formal system and implementation is that the implementation contains many "complex" annotations. However, these additional complex annotations can be compiled into the lower-level annotations presented the formal system. As an example, consider the repetition operator used in Section 2.2:

```
{Record/Elem*[|]: 9152271| {Elem: 9152271 } |1| \
  0|0|0|0|... }
```

This idiom may be compiled into the following collection of annotations drawn from the formal system:

```
{NewRecord:{Record/NewElem*: 9152271| {NewElem:{Elem: \
9152271 } | } 1|0|0|0|0|...| }{Elem!: 0 }}
```

The extraction process will give rise to this grammar:

```
Elem ::= Num
NewElem ::= Elem '|'
Record ::= NewElem*
NewRecord ::= Record Elem
```

4. IDEALIZED ANNE Properties

Now that we have defined the semantics of IDEALIZED ANNE, we can answer some important questions about its properties and expressive power. For instance, suppose one has some data v that inhabits the language of a grammar $gram$, is it always the case that one can annotate v in such a way as to extract $gram$? Unfortunately, the answer to this question is no. The simplest counter-example involves choosing the empty string as the data and a grammar $(A, [A = \epsilon + Num])$ as the target to extract — in this example, there is no way to annotate the empty string to enable generation of the right side of the union. However, we can extract $(A, [A = \epsilon])$ — an *approximation* of the grammar we might have wanted.

Intuitively, we can extract $[A = \epsilon]$ but not $[A = \epsilon + Num]$, because in the former case all branches of the grammar are *used* during a parse of the empty string whereas in the latter case, some branches (the Num branch) go *unused* during the parse. Hence, in order to better understand the grammars that can be extracted from data, we need a theory that captures those parts of the grammar that are used during recognition of a string. That theory is closely connected to the substructural logic known as relevance logic.

4.1 Relevance Analysis

Relevance Logic [1] is a simple logic requires every hypothesis be *used at least once* during the course of a proof. If we think of grammar rules as hypotheses in a proof, we can develop an analogous theory in which each grammar rule, and all of its subparts, must be *used at least once* in the derivation that a string belongs to the grammar.

Based on this intuition, we have developed a *relevance analysis* that directly relates grammars to the values that can generate them. The central judgements for this analysis have the form $\mathsf{G} \vdash_{rel} v \in r$ and $\mathsf{G} \vdash^{c}_{rel} v \in s$. These judgements affirm that all elements of G are used during the course of proving that v is an element of r and s respectively. A third judgement, $\vdash_{rel} v \in gram$, affirms that all elements of $gram$ are used during the course of proving v is in $gram$. Figure 12 presents the inference rules for these judgements.

Rule (*e-re*) provides an example of how these rules work. It states that v is recognized by b provided it is in $\mathcal{L}(b)$. Moreover, this rule uses no parts of a grammar. Hence, the grammar to the left of the turnstile must be empty. Rule (*e-name*) states that if G is used in recognizing that v belongs to r then $\mathsf{G} \oplus [A = r]$ is used in recognizing that v belongs to A. Rule (*e-con*) states that if G_1 through G_n are used in recognizing s_1 to s_n then the grammar fusion is used to recognize the concatenation of clauses.

It is also important to observe how the unions work. In particular, there are rules (*e-sum1*) and (*e-sum2*) to explain what the partial right-hand sides $? + s$ and $s + ?$ use, but there are no rules for the complete right-hand side $s_1 + s_2$. This is because no derivation can use both the left-hand side and the right-hand side of a union simultaneously.

The rules for repetitions are also interesting. Notice that the rule (*e-rep*) is constrained so that i is greater than 0. This guarantees that the underlying element grammar is used. The rule (*e-rep-empty*) is for the situation in which the empty string matches an iteration. The entire reason for including the right-hand side $A*_0$ is to distinguish this case in which the underlying element type is not used.

Our relevance analysis may be viewed as a relevance logic primarily because the structural rules for *exchange* and *contraction* are admissible but *weakening* is not.

4.2 Relevant Properties

The key property of relevance analysis stems from the following essential property: if a grammar is relevant to a string then a programmer can use IDEALIZED ANNE to extract it from the string. In particular, the programmer does not need to annotate with assertions. In other words, assertions are not essential features and they are only introduced for annotation convenience.

Theorem 1 (Relevance implies grammar extraction.)

i. If $\mathsf{G} \vdash^{c}_{rel} v \in s$, then there exists ad such that $v \rightarrow ad$, $ad \rightsquigarrow (s, \mathsf{G})$ and the rule (a-assert) has never been used in the derivation of ad;

ii. If $\mathsf{G} \vdash_{rel} v \in r$, then for any A, there exists ad such that $v \rightarrow ad$, $ad \rightsquigarrow (A, \mathsf{G} \oplus [A = r])$ and the rule (a-assert) has never been used in the derivation of ad;

iii. If $\vdash_{rel} v \in gram$, then there exists ad such that $v \rightarrow ad$ and $ad \rightsquigarrow gram$.

The theorem above states properties of a single string, but IDEALIZED ANNE can sometimes do more for us when there is more than one string to annotate. To make this idea precise, we first define what it means to extract a grammar from a collection of strings.

Definition 2 (Collective Extraction.)

For grammar $gram = (A, \mathsf{G})$ and data $v_1, v_2, ..., v_k$, $v_1, v_2, ..., v_k \rightsquigarrow gram$ iff there exists $ad_1, ..., ad_k$ such that

- $v_i \rightarrow ad_i$, *for all* $i = 1, ..., k$;
- $ad_i \rightsquigarrow (s_i, \mathsf{G}_i)$ *for all* $i = 1, ..., k$;
- $\mathsf{G}_1 \oplus \mathsf{G}_2 \oplus ... \oplus \mathsf{G}_k = \mathsf{G}$.

Next, we present the following theorem, which states that no matter what data one has in hand, one can extract an *approximation* of any grammar for that data, where approximate grammars are defined as follows: (A, G_1) is an approximation of (A, G) provided that there exists G_2 such that $\mathsf{G} = \mathsf{G}_1 \oplus \mathsf{G}_2$. We write $(A, \mathsf{G}_1) \leq (A, \mathsf{G})$ when (A, G_1) is an approximation of (A, G).

Theorem 3 (Sound collective extraction.)

Given some data $v_1, v_2, ..., v_k$, if $\vdash v_i \in gram$ for all i, then there exists $gram'$ such that $v_1, v_2, ..., v_k \rightsquigarrow gram'$ and $\vdash v_i \in gram'$ and $gram' \leq gram$.

We proved Theorem 3 by showing that relevance analysis is a sound approximation of ordinary grammar recognition and then constructing the grammar $gram'$ from the grammars $gram_1, \ldots, gram_n$ that are relevant for each datum $v_1, \ldots, v_n$.

To summarize, given a grammar $gram$ and some data v, our theoretical analysis has told us two useful facts: (1) if $gram$ is a grammar for v then IDEALIZED ANNE can extract some approximation to $gram$, and (2) if $gram$ is relevant for v then IDEALIZED ANNE can extract $gram$ exactly. Given the second point, one might say relevance analysis is a *sound* characterization of IDEALIZED ANNE. However, it is not a *complete* characterization. There exist some "unusual" grammars that are not relevant for any data, but can be extracted by IDEALIZED ANNE. In particular, IDEALIZED ANNE can extract grammars with disconnected nonterminals. One simple example is the grammar $(A, [A = int, B = int])$. This grammar is not relevant for any data, but can be extracted from two example documents that each contain a single integer.

$\boxed{\mathsf{G} \vdash^{c}_{rel} v \in s}$

$$\frac{v \in \mathcal{L}(b)}{[] \vdash^{c}_{rel} v \in b}\ (\textit{e-re}) \qquad \frac{\mathsf{G} \vdash_{rel} v \in r}{\mathsf{G} \oplus [A = r] \vdash^{c}_{rel} v \in A}\ (\textit{e-name})$$

$$\frac{\mathsf{G}_i \vdash^{c}_{rel} v_i \in s_i \qquad i = 1..n}{\mathsf{G}_1 \oplus \mathsf{G}_2 \oplus ... \oplus \mathsf{G}_n \vdash^{c}_{rel} v_1 v_2 ... v_n \in s_1 \cdot s_2 \cdot ... \cdot s_n}\ (\textit{e-con})$$

$\boxed{\mathsf{G} \vdash_{rel} v \in r}$

$$\frac{\mathsf{G} \vdash^{c}_{rel} v \in s}{\mathsf{G} \vdash_{rel} v \in s}\ (\textit{e-clause})$$

$$\frac{\mathsf{G} \vdash^{c}_{rel} v_1 \in s_1}{\mathsf{G} \vdash_{rel} v_1 \in s_1 + ?}\ (\textit{e-sum1}) \qquad \frac{\mathsf{G} \vdash^{c}_{rel} v_2 \in s_2}{\mathsf{G} \vdash_{rel} v_2 \in ? + s_2}\ (\textit{e-sum2})$$

$$\frac{\mathsf{G}_i \vdash^{c}_{rel} v_i \in A \qquad i = 1..n \quad n > 0}{\mathsf{G}_1 \oplus \mathsf{G}_2 \oplus ... \oplus \mathsf{G}_n \vdash_{rel} v_1 v_2 ... v_n \in A*}\ (\textit{e-rep})$$

$$\frac{}{[] \vdash_{rel} \text{“”} \in A*_0}\ (\textit{e-rep-empty})$$

$\boxed{\vdash_{rel} v \in gram}$

$$\frac{\mathsf{G} \vdash_{rel} v \in A}{\vdash_{rel} v \in (A, \mathsf{G})}\ (\textit{e-gram})$$

Figure 12. Relevance analysis.

5. Evaluation

We conducted a series of experiments to compare using ANNE against the process of writing PADS descriptions by hand and against the process of learning descriptions automatically using LEARNPADS [11]. When comparing ANNE with hand-written descriptions, we focused on the time and efforts users dedicated to creating descriptions; when comparing ANNE with the LEARNPADS system, we focus on the readability and compactness of descriptions. Our benchmark formats include 19 different ad hoc data sources, drawn mainly from various different kinds of system logs. The same benchmarks have been used previously to evaluate the effectiveness of PADS and its variants [11]. Those readers interested in the specifics can find the benchmarks on the web [24].

Comparison with hand-written descriptions. In the first set of experiments, we measured the time and effort spent constructing descriptions using ANNE. For each benchmark, Table 1 shows the total number of annotations the programmer needed to construct the description (# annots), the total number of lines that were annotated (# lines) and the approximate time in minutes for the user to complete the description. The number of annotations did not include the preamble or the regular expressions defined therein.

The table shows that for most of our benchmarks, the user needed to insert anywhere from 1 to 14 annotations (with the median being 5). On average, the user was required to annotate 3 or 4 lines of data. The time taken varied between 5 and 15 minutes. In contrast, a previous study [11] of the time taken to write the same descriptions by hand showed users with some experience spent anywhere from 1/2 an hour to an hour or two. Part of the reason users would take longer to write descriptions by hand is that they can add additional information in the form of constraints – something that is not supported by ANNE right now. However, from the experience of several PADS and ANNE programmers, the main reason is simply that ANNE is easier to use.

Data Source	# Annots	# Lines	Time(min)
1967Transactions	6	1	5
ai.3000	14	4	10
yum.txt	6	1	15
rpmpkgs	2	1	1
railroad.txt	10	4	10
dibbler.1000	6	3	5
asl.log	7	2	5
scrollkeeper.log	4	1	3
page_log	5	1	5
MER_T01_01.csv	1	1	1
crashreporter.log	4	1	3
ls-l	4	2	5
windowserver_last	5	1	10
netstat-an	10	3	10
boot.txt	7	1	5
quarterlyincome	3	2	5
corald.log.head	3	2	5
irvpiv1.sel	7	1	15
latitude.txt	10	3	15

Table 1. Number of annotations, lines touched and time taken to construct descriptions using ANNE.

Data source	Type Complexity		Desc. Size	
	A	L	A	L
1967Transactions	52	175	13	26
ai.3000	328	437	56	47
yum.txt*	84	640	17	74
rpmpkgs*	7	314	4	70
railroad.txt *	89	975	28	150
dibbler.1000	76	85	21	25
asl.log	551	1545	78	102
scrollkeeper.log	44	372	8	14
page_log	206	729	23	22
MER_T01_01.csv	96	211	22	12
crashreporter.log *	105	973	16	63
ls-l*	195	721	25	80
windowserver_last	148	85	24	11
netstat-an	822	1324	57	138
boot.txt*	98	944	19	123
quarterlyincome	520	579	86	87
corald.log.head	793	1094	106	71
irvpiv1.sel*	284	1334	44	130
latitude.txt*	140	500	11	77

Table 2. ANNE (A) vs. LEARNPADS (L): Type complexity in bits and description size in lines. Asterisks indicate meaningful qualitative differences in the performance of the two systems.

***Comparison with* LEARNPADS.** Before comparing LEARNPADS with ANNE in detail, we would like to review the basics of how LEARNPADS works. LEARNPADS uses a multi-stage algorithm to automatically discover the structural information in the input data source according to which the format specification, presented as a PADS description, is generated. During the first stage, LEARNPADS tokenizes the text data using a fixed set of base tokens. During the second stage, it analyzes the distribution of tokens found within the data and infers a candidate grammar. During the last stage, the system applies a set of rewriting rules to optimize the candidate structure according to the minimum de-

```
total 275528
drwxr-xr-x  3  dpw  fac  4096  Jan 21  2005  as9
drwxr-xr-x  4  dpw  fac  4096  Jan 21  2005  as8
-rw-r--r--  1  dpw  fac 15878  Jan 23  2002  asynch.txt
drwxr-xr-x  2  dpw  fac  4096  Jan  2 13:44  cv
...
```

Figure 13. Excerpt from `ls-l`

scription length principle, which balances specificity of a grammar against compactness. Because of the nature of the second stage of the algorithm, LEARNPADS is incapable of inferring context-free grammars.

Table 2 presents a detailed comparison between the two systems. This table presents two metrics: the *type complexity* of the resulting description and the number of lines of the resulting description when printed. The type complexity measures the number of bits it would take to encode the syntax of the PADS description. It is one of the metrics that the LEARNPADS system optimizes for. The number of lines of the resulting description is simply the number of lines of output from the respective pretty printers. Differences of 20% or so are usually meaningless in this table. On the other hand, differences on the order of a factor of 5 or 10 are quite meaningful — we placed asterisks in Table 2 to indicate those formats for which the differences between the results produced by ANNE and those by LEARNPADS were very significant.

Significant differences occur for several reasons, but perhaps the most pervasive is that the performance of LEARNPADS is quite sensitive to the set of basic tokens (definitions of times, dates, ip addresses, *etc.*) that it starts out with. Unfortunately, while the LEARNPADS designers would like to create the "perfect" tokenizer, doing so for a broad set of formats is extremely difficult. As one begins to add more and more token definitions to the token set, the token definitions become mutually ambiguous with no obvious way to resolve the ambiguities. For instance, the proper definition of URLs is incredibly broad and is ambiguous with just about anything. Some date formats are ambiguous with URLs, file paths, phone numbers, floating point numbers or IP addresses. As a result, the LEARNPADS strategy has been to use a relatively simple default tokenizer. However, the consequence is that unanticipated token types will show up in data files with some frequency and when this happens, LEARNPADS often produces overly complex grammars. ANNE does not suffer from this problem because users can override the default tokenizer with local annotations whenever they need to.

LEARNPADS will also do a suboptimal job learning some formats because the rewriting heuristics it uses fail. As an example, consider the ls-l data source presented in Figure 13. This data was obtained by executing Unix command "ls -l." The difficulty with this data file is that it contains an insufficiently diverse set of examples from which to learn a accurate format. In particular, when LEARNPADS is applied to such a data source, it fails to properly generalize in the following ways:

- The access control strings are turned in to an enumeration of four possibilities instead of a more general string description.
- The owner and group fields are turned in to constants `dpw` and `fac`.
- There is a switch on the constant "4096" because that file size happens to show up uncharacteristically often in the file.

If the example data used was more varied, the learning system would work as expected. More generally, since LEARNPADS is driven by a statistical analysis and heuristic rewriting rules, its results can be unpredictable, a problem that ANNE does not have.

In addition to sometimes *under*generalizing, LEARNPADS will sometimes *over*generalize. For instance, Table 2 indicates that the LEARNPADS system produced a much smaller description than ANNE when applied to the `windowserver_last` benchmark. This occurs because LEARNPADS uses a heuristic to simplify grammars, and in this case, it over-simplified, eliminating some useful information about the format. Of course, if the simple description was the desired one, it would have been possible for the ANNE programmer to generate it.

6. Related Work

ANNE was designed to improve the productivity of data analysts by providing a quick, simple way to generate documentation and data processing tools for an ad hoc data source given the availability of example data. Many of its commands are directly inspired by the design of domain-specific languages and language extensions such as PADS [9, 10, 19], DATASCRIPT [3], PACKETTYPES [21], Demeter [18], BINPAC [25] and Erlang binaries [30, 14].

ANNE is not designed to work in the binary domain - it will only work well when a human can stare at a data source, uncover it's structure, and add annotations in place. When it comes to the domain of semi-structured text data, ANNE provides an alternative to writing format specifications (like PADS specifications) by hand. The main advantage of ANNE comes in its ease-of-use and ability to fill in details such as separators, terminating characters, and members of an enumeration automatically. Having a machine fill in such details is both more convenient and less error-prone than manually constructing the description. One limitation of ANNE right now is that it does not support the full range of PADS features. In particular, it is missing dependency and constraints. We believe the overall ANNE framework can support these features; we are currently working on extending our theory and implementation to include them.

Potter's Wheel [27] is another system with some similarities to to ANNE in that it supports an interactive process to manage, clean and transform data. Unlike ANNE, it uses a spreadsheet-style interface to represent classical relational data and its goal is to help users detect errors and transform data to make it ready to load into a commercial database. Whereas Potter's Wheel is limited to managing relational tables, ANNE is designed for a broader range of context-free grammars; whereas Potter's Wheel is an on-the-fly interactive transformation system, ANNE is a descriptive system that produces documentation and programming tools for later use.

Whereas Potter's Wheel operates over relational data, many other data cleaning and transformation systems operate over XML. For example, SchemaScope [4] is a powerful new tool developed by Bex, Neven and Vansummeren to infer DTDs and XML Schemas from unknown XML documents and to visualize and edit existing schema. The inference mechanisms used in SchemaScope are highly effective as they are tuned to common properties of DTDs [20]. Unfortunately, the grammar inference problem for ad hoc data sources is substantially different from the schema inference problem for XML in part because the basic tokenization problem for ad hoc data is so ambiguous — there is no standard tag-based syntax to delineate different parts of an ad hoc document. On the contrary, ANNE was created to provide a means for programmers to delineate and disambiguate elements of their data sources.

The machine learning community has developed a number of tools that perform *wrapper induction*, where a *wrapper* is a program that can extract information from designated "slots" in a document or set of documents. Two examples of such work are Kushmerick's HLRT induction system [16] and Soderland's Whisk system [28]. One high-level difference between a system like Whisk and one like ANNE or PADS is that Whisk is designed to work on

data with very little regular structure. For example, the working example in Soderland's paper involved extraction of features such as price and location from Craigslist apartment advertisements. Such advertisements are pseudo-English blurbs and have much less structure than web logs, for instance. Hence, while Whisk and similar systems can be effective at solving the information extraction problem, they are not designed to produce the kind of documentation or programming tools that ANNE is.

7. Conclusions

In this paper, we have presented the design and implementation of ANNE, a new kind of markup language for text data. This markup language allows users to specify the syntactic structure of documents by adding annotations that indicate the presence of constants, enumerations, repetitions, optional data, tables, and recursive data. The markup language also allows users to import from libraries of pre-defined regular expressions and to name parts of their data as they choose. A markup can be used to generate a context-free grammar, an XML parse tree and a PADS description. The XML parse tree facilitates debugging and the PADS description serves as useful documentation that may be compiled into many more useful tools. Experience with ANNE suggests that compact, human-readable descriptions can be constructed more quickly than by writing PADS descriptions by hand and more reliably than using LEARNPADS.

In addition, we have defined and analyzed the semantics of ANNE. In the process of doing so, we have uncovered a connection to relevance logic, which we have used to prove illuminating theorems concerning the expressiveness of our system.

Acknowledgments

We would like to thank Kathleen Fisher, Nate Foster and Kenny Zhu for many fruitful conversations on this topic.

This material is based upon work supported by the NSF under grants 0612147 and 0615062 and by a gift from Google. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF or Google.

References

[1] A. R. Anderson, N. Belnap, and J. Dunn. *Entailment: The Logic of Relevance and Necessity*. Princeton University Press, Princeton, NJ, 1975.

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

[3] G. Back. DataScript - A specification and scripting language for binary data. In *GPCE*, volume 2487, pages 66–77. Lecture Notes in Computer Science, 2002.

[4] G. J. Bex, F. Neven, and S. Vansummeren. SchemaScope: a system for inferring and cleaning xml schemas. In *SIGMOD*, pages 1259–1262, 2008.

[5] V. Crescenzi, G. Mecca, and P. Merialdo. Roadrunner: Towards automatic data extraction from large web sites. In *VLDB*, pages 109–118, San Francisco, CA, USA, 2001.

[6] F. Denis, A. Lemay, and A. Terlutte. Learning regular languages using rfsas. *Theor. Comput. Sci.*, 313(2):267–294, 2004.

[7] M. Fernandez, K. Fisher, J. Foster, M. Greenberg, and Y. Mandelbaum. A generic programming toolkit for PADS/ML: First-class upgrades for third-party developers. In *PADL*, pages 133–149, 2008.

[8] M. F. Fernández, K. Fisher, R. Gruber, and Y. Mandelbaum. PADX: Querying large-scale ad hoc data with xquery. In *PLANX*, Jan. 2006.

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

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

[11] K. Fisher, D. Walker, K. Q. Zhu, and P. White. From dirt to shovels: Fully automatic tool generation from ad hoc data. In *POPL*, pages 421–434, Jan. 2008.

[12] P. García and E. Vidal. Inference of k-testable languages in the strict sense and application to syntactic pattern recognition. *IEEE Trans. Pattern Anal. Mach. Intell.*, 12(9):920–925, 1990.

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

[14] P. Gustafsson and K. Sagonas. Adaptive pattern matching on binary data. In *ESOP*, pages 124–139. Springer, Mar. 2004.

[15] T. W. Hong and K. L. Clark. Using grammatical inference to automate information extraction from the Web. *Lecture Notes in Computer Science*, 2168:216+, 2001.

[16] N. Kushmerick. *Wrapper induction for information extraction*. PhD thesis, University of Washington, 1997. Department of Computer Science and Engineering.

[17] K. Lerman, L. Getoor, S. Minton, and C. Knoblock. Using the structure of web sites for automatic segmentation of tables. pages 119–130, New York, NY, USA, 2004.

[18] K. J. Lieberherr and A. J. Riel. Demeter: A CASE study of software growth through parameterized classes. 1(3):8–22, August 1988.

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

[20] W. Martens, F. Neven, T. Schwentick, and G. J. Bex. Expressiveness and complexity of XML schema. *ACM Trans. Database Syst.*, 31(3):770–813, 2006.

[21] P. McCann and S. Chandra. PacketTypes: Abstract specification of network protocol messages. In *SIGCOM*, pages 321–333. ACM Press, August 2000.

[22] H. T. Ng, C. Y. Lim, and J. L. T. Koo. Learning to recognize tables in free text. pages 443–450, Morristown, NJ, USA, 1999.

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

[24] PADS project learning demo. `http://www.padsproj.org/learning-demo.cgi`, 2007.

[25] R. Pang, V. Paxson, R. Sommer, and L. Peterson. binpac: a yacc for writing application protocol parsers. In *IMC '06: Proceedings of the 6th ACM SIGCOMM conference on Internet measurement*, pages 289–300, New York, NY, USA, 2006. ACM.

[26] D. Pinto, A. McCallum, X. Wei, and W. B. Croft. Table extraction using conditional random fields. In *SIGIR*, pages 235–242, New York, NY, USA, 2003.

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

[28] S. Soderland. Learning information extraction rules for semi-structured and free text. *Machine Learning*, 34(1-3):233–272, 1999.

[29] A. Stolcke and S. Omohundro. Inducing probabilistic grammars by bayesian model merging. In *ICGI*, 1994.

[30] C. Wikström and T. Rogvall. Protocol programming in Erlang using binaries. In *Fifth International Erlang/OTP User Conference*, Oct. 1999.