The PADS Project: An Overview

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

David Walker
Princeton University
dpw@cs.princeton.edu

ABSTRACT

The goal of the PADS project, which started in 2001, is to make it easier for data analysts to extract useful information from ad hoc data files. This paper does not report new results, but rather gives an overview of the project and how it helps bridge the gap between the unmanaged world of ad hoc data and the managed world of typed programming languages and databases. In particular, the paper reviews the design of PADS data description languages, describes the generated parsing tools and discusses the importance of meta-data. It also sketches the formal semantics, discusses useful tools and how can they can be generated automatically from PADS descriptions, and describes an inferencing system that can learn useful PADS descriptions from positive examples of the data format.

Categories and Subject Descriptors

D.3.m [**Programming languages**]: Miscellaneous

General Terms

Languages, Algorithms

Keywords

Data description languages, ad hoc data, domain-specific languages

1. WHY PADS

Traditional databases and XML-based systems provide rich infrastructure for managing well-behaved data, but provide little support for *ad hoc formats*, which we define to be any semi-structured data format for which parsing, querying, analysis, or transformation tools are not readily available. Vast amounts of useful data are stored and processed in such ad hoc formats, despite the existence of standard formats for semi-structured data. Examples arise from a myriad of domains, including finance, health care, transportation, telecommunications, and the sciences [11].

For a number of reasons, processing ad hoc data is challenging. Ad hoc data typically arrives "as is": analysts are lucky to get the data at all and have little chance of getting suppliers to standardize its format. Documentation for the format is often incomplete, out-of-date, or completely non-existent. Such data frequently contain errors for a variety of reasons, including malfunctioning equipment, non-standard representations of missing values, human error in entering data *etc.* Handling such errors is challenging because of the variety of errors and because the appropriate response is application-dependent. Some applications require the data to be error free and need to halt processing if any errors are detected. Others can simply discard erroneous values, while still others want to study the errors in detail. All of these challenges are exacerbated by the fact that ad hoc data sources often have high volume. For example, AT&T's Altair project accumulates billing data at a rate of 250-300GB/day, with occasional spurts of 750GBs/day.

Finally, someone has to write a parser for a new format before anyone can use the data. People tend to use C, PERL, or PYTHON for this task. This approach is tedious and error-prone, complicated by the lack of documentation, convoluted encodings designed to save space, the need to produce efficient code, and the need to handle errors robustly to avoid corrupting down-stream data. Moreover, the parser writers' hard-won understanding of the data ends up embedded in parsing code, making long-term maintenance difficult for the original writers and sharing the knowledge with others nearly impossible.

The PADS project started with the observation that an appropriately designed, declarative data-description language could help bridge the gap between the *unmanaged* world of ad hoc data and the *managed* world of strongly typed programming languages and databases and thereby help alleviate many of the concerns mentioned above. To this end, the language we have designed permits analysts to describe ad hoc data *as it is*, not how one might want it to be. The descriptions are concise enough to serve as documentation and flexible enough to describe most of the data formats we have seen in practice, including ASCII, binary, Cobol, and mixed data formats. From these descriptions, a compiler can produce data structure declarations for representing the data in the host language of the data description language as well as parsing and printing routines. Because the compiler is generating software artifacts used to manipulate the data, analysts have to keep the data description up to date, ensuring it can serve as living documentation.

The declarative nature of PADS descriptions facilitates the insertion of error handling code. The generated parsing code checks all possible error cases. Because these checks appear only in generated code, they do not clutter the high-level declarative description of the data source. The result of a parse is a pair consisting of a canonical in-memory representation of the data and a parse descriptor. The parse descriptor precisely characterizes both the syntactic and the semantic errors that occurred during parsing. This structure allows analysts to choose how to respond to errors in application-specific ways.

```
171.64.78.97 - kfisher [31/Oct/2010:23:55:36 -0700] "GET /padsproj.org HTTP/1.1" 200 982
quantum.com  - - [31/Oct/2010:24:55:36 -0700] "GET /padsproj.org/logo.png HTTP/1.1" 200 2326
```

Figure 1: A fragment of data in Apache Common Log Format.

Finally, a PADS description gives enough information about the structure of the data that it is possible to generate automatically a wide variety of useful tools customized to the particular format. Examples of such tools include statistical analyses, format converters, data adaptors to connect to query engines like XQuery [9], and visualizers. Using generic programming techniques [8, 19, 21], third party developers can design tools that will work for any PADS description.

This paper does not describe new research, but rather collects and summarizes the work done in the PADS project. The interested reader is invited to read more about individual aspects of the project in the original papers. The project is the work of many people over a long period of time; all contributors are listed in the acknowledgements section of the paper.

The rest of the paper is organized as follows. Section 2 explains what a PADS data description language looks like by working through a simple example. Section 3 describes the output of a PADS compiler. Section 4 describes a formal semantics for data description languages, which serves as a specification for how PADS data description languages should behave, regardless of the host language in which they are embedded. Section 5 describes the kinds of tools we can generate from PADS descriptions as well as how generic programming makes third-party tool generation possible. Section 6 describes how we can leverage large quantities of data to learn PADS descriptions rather than having to write them entirely by hand. Section 7 briefly reviews related work and Section 8 concludes.

2. WHAT YOU SAY

In this section, we briefly describe what a PADS data description language looks like by working through a simple example. Figure 1 gives a small fragment of a web access log in Apache Common Log Format (CLF) [5]. Each time an Apache web server receives a request, it writes such an entry to its access log. The first field denotes the IP address or host name of the client making the request. The next two columns denote the identity of the client user, the first as determined by the `identd` function on the client machine and the second as determined by HTTP authentication. If the information is not available, the server writes a - instead. The next field is the time stamp of the request, in brackets. It is followed by the request itself in double quotes. The request has three pieces: the HTTP method used by the client, the requested url, and the version of HTTP used for the transaction. The format ends with two integers, the first of which is the three-digit response code sent back to the client and the second is the number of bytes returned. If no bytes were returned, this last number will be a dash instead.

PADS uses a type metaphor to describe ad hoc data. Base types describe atomic pieces of data and type constructors describe how to build compound descriptions from simpler ones. Each PADS type plays two roles: it defines a grammar for parsing the data *and* a format-specific data structure in which to store the result of the parse. We have developed versions of PADS for C [11] and ML [21], both of which are available for download on the web with an open-source license [23]. We are currently developing a version for Haskell, which we will use as the example language in this paper. For each language binding, we re-use the types of the host language in the data description language. PADS also uses predicate expressions in the host language to describe semantic properties of the data, such as the range of integer values or correlations between data items.

```
[pads|
  type CLF_t = [Line Entry_t]
  data Entry_t =
    {       host       :: Source_t,
      ' ',  identdID   :: ID_t,
      ' ',  httpID     :: ID_t,
      ' ',  time       :: TimeStamp_t,
      ' ',  request    :: Request_t,
      ' ',  response   :: Response_t,
      ' ',  contentLen :: ContentLength_t
    }

  data Source_t = IP IP_t | Host Host_t

  data ID_t = Missing '-' | Id (Pstring ' ')

  data Request_t =
    { '"',  method  :: Method_t,
      ' ',  url     :: Pstring ' ',
      ' ',  version :: Version_t,
      '"' }

  data Version_t  =
    {"HTTP/", major :: Pint, '.', minor :: Pint }

  data Method_t = GET | PUT | POST | HEAD
                | DELETE | LINK | UNLINK

  type Response_t = constrain r :: Pint
       where <| 100 <= r && r < 600 |>

  data ContentLength_t = NotAvailable '-'
                       | ContentLength Pint
|]
```

Figure 2: PADS/HASKELL description of CLF data.

The PADS/HASKELL description in Figure 2 describes the CLF data format. We embed PADS in Haskell using Haskell's quasi-quoting mechanism [20]. All of the code inside the quasi-quotes `[pads|...|]` is PADS/HASKELL; the identifier `pads` tells the Haskell parser which quasi-quoter to use to process the enclosed code. Any code written outside of the quasi-quotes is ordinary Haskell. Moreover, Haskell declarations written prior to such quotes are in scope and available in the PADS/HASKELL code. At compile time, the Haskell compiler calls the PADS quasi-quoter to convert the PADS/HASKELL code to plain Haskell declarations, which are spliced into the source code at the point of the quasi-quotation. These declarations are subsequently in scope for any Haskell code that follows. This mechanism enables us to completely control the syntax of our data description language while still inter-operating with the host language. Indeed, the ability to alternate back and forth at will between PADS and Haskell declarations provides a smooth, flexible and pleasing programming experience — the two languages act as one.

The first line of the description declares the type `CLF_t`, which describes the entirety of an access log. It says that such a log is a list of lines of type `Entry_t`. The next line declares the type `Entry_t`, which is a record describing a single entry in the log. It says that an entry is a sequence of seven fields; each field is given a

```
newtype CLF_t = CLF_t [Entry_t]
data Entry_t = Entry_t
         {host       :: Source_t,
          identdID   :: ID_t,
          httpID     :: ID_t,
          time       :: TimeStamp_t,
          request    :: Request_t,
          response   :: Response_t,
          contentLen :: ContentLength_t}
data Source_t = IP IP_t | Host Host_t
data ID_t = Missing | Id Pstring
 ...
newtype Response_t = Response_t Pint
```

Figure 3: Generated representation types.

name and an associated type. For example, the `host` field has type `Source_t`, which is also declared in the figure. In addition to the named fields, there are literal characters in the record declaration, which correspond to literals in the data. For example, between the `host` and `identdID` fields, there is a space character (`' '`).

The `Source_t` type is an interesting example of a datatype: a value of this type is *either* an IP address `IP_t` or a `Host_t`, where `IP_t` and `Host_t` are PADS *base types* describing IP addresses and host names, respectively. The branches of a datatype are attempted in order; the parser greedily selects the first branch that matches. The labels `IP` and `Host` tag the parsed value as belonging to the first or second branch, respectively. Type `ID_t` is another datatype. In this case, the first branch corresponds to a data format with the literal `'-'`. The type `Method_t` is a datatype where the branch labels correspond to the data, and so the argument type to the branch label is omitted.

Base types describe atomic pieces of data. Examples of base types include integers (`Pint`) and floats (`Pfloat`), single characters (`Pchar`) and strings (`Pstring ' '`), IP addresses (`IP_t`) and host names (`Host_t`), dates (`Date_t`), and many others. The type `Pstring ' '` is an example of a *parameterized type*. In general, a string could go on forever. The parameter specifies when the string should stop, in this case, when the parser encounters a space. To account for more general stopping conditions, PADS/HASKELL provides the base type `PstringME`, which takes a regular expression as a parameter. This type matches the longest string that matches the argument regular expression.

Finally, the type `Response_t` is an example of a *constrained type*. It specifies that a `Response_t` is a `Pint` between 100 and 599, inclusive. In the declaration, the variable `r` is bound to the result of parsing the input as a `Pint`, after which the predicate given in the `where` clause is evaluated. The type matches if the predicate evaluates to `True`. The brackets `<|...|>` indicate that the enclosed code is pure Haskell code.

3. WHAT YOU GET

From a PADS specification, the compiler generates a pair of data structures: one for the in-memory representation of the parsed data and an isomorphic structure for meta-data such as the number and type of errors. The form of the generated representation type corresponds to the form of the type in the PADS description: PADS lists compile to Haskell lists, records to records, and data types to data types. Constrained types map to the representation of the underlying type. Figure 3 gives a selection of the representation types generated for the CLF description.

The meta-data types, shown in Figure 3, have a similar structure. These declarations make use of the type `Base_md` to describe the number and type of errors detected during parsing:

```
type CLF_t_md   = (Base_md, [Entry_t_md]),
type Entry_t_md = (Base_md,  Entry_t_inner_md),
data Entry_t_inner_md = Entry_t_inner_md
        {host_md       :: Source_t_md,
         identdID_md   :: ID_t_md,
         httpID_md     :: ID_t_md,
         time_md       :: TimeStamp_t_md,
         request_md    :: Request_t_md,
         response_md   :: Response_t_md,
         contentLen_md :: ContentLength_t_md}
type Source_t_md = (Base_md, Source_t_inner_md)
data Source_t_inner_md
       = IP_md IP_t_md | Host_md Host_t_md
data ID_t = Missing | Id Pstring
data ID_t_inner_md
       = Missing_md Base_md
       | Id_md (Base_md, Base_md)
  ...
newtype Response_t = Response_t Pint
```

Figure 4: Generated meta-data types.

```
data Base_md = Base_md
        { numErrors :: Int,
          errInfo   :: Maybe ErrInfo }
data ErrInfo = ErrInfo
       { msg      :: ErrMsg,
         position :: Maybe Position }
```

Each generated meta-data type pairs a generic `Base_md` with a type-specific meta-data structure. The `Base_md` type summarizes the errors that occurred within the corresponding structure while the type-specific meta-data localizes error information. This structure allows analysts to handle errors in application-specific ways. By checking the top-level `Base_md` value, analysts can determine whether there were any errors during parsing. If the error count is zero, they know the data parsed without any errors and all the semantic predicates held. If the error count is non-zero, they can traverse the meta-data structure to determine precisely where the errors occurred and what caused them, allowing the analysts to decide whether to discard, repair, or study the errors. This design also means that the representation is not cluttered with option types indicating that each value could be absent because of an error during parsing. If an error occurs while parsing a base type, the compiler fills in an appropriate default value and marks the meta-data accordingly.

Finally, the compiler generates a function that parses an input file into a pair of a representation and a meta-data structure. In Haskell terms, each generated representation (`rep`), meta-data (`md`) pair belongs to the `Pads` type class and provide a definition for the `parseFile` method:

```
parseFile :: Pads rep md =>
             FilePath -> IO (rep, md)
```

The return type `IO(rep,md)` indicates that the function produces a value of type `(rep,md)` while causing side-effects such as opening and closing file handles. A particular instance of this function parses values for the type `CLF_t`:

```
parseFile :: FilePath -> IO (CLF_t, CLF_t_md)
```

The generated parser is a simple recursive-descent parser. This parsing strategy makes it easy for values early in the parse to affect down-stream choices, for example, to read an integer that determines the length of an upcoming list or a tag that predicts what form the body of a record will take. While generally satisfactory, recursive descent parsers cannot parse left-recursive grammars and can require exponential time (if there is a lot of backtracking). Developing always-efficient parsers suitable for powerful PADS-like grammars is currently an active research question [17].

Although not yet implemented in PADS/HASKELL, PADS/C and PADS/ML both also generate a pretty printing function that takes a pair of a representation and meta-data structure and serializes the representation to a file. In Haskell, this function will have the signature:

```
printFile :: Pads rep md =>
             (rep, md) -> FilePath -> IO ()
```

4. WHAT IT MEANS

The close correspondence between PADS descriptions and the type structure of the host language makes the meaning of PADS descriptions relatively intuitive, but it does not suffice to precisely define their semantics. To address this deficiency, we developed a formal calculus, called DDC$^\alpha$, based on dependent type theory [13].

We defined a denotational semantics for DDC$^\alpha$ that interprets each term in multiple ways. In the first interpretation, each DDC$^\alpha$ term τ is mapped to a type that we call its "representation" type, τ_{rep}. This type describes the data structure that stores the host-language representation of the parsed value. In the second interpretation, each DDC$^\alpha$ term τ is mapped to a type that we call its "meta-data" type, τ_{md}. This type describes the data structure that stores the host-language representation of the meta-data generated during parsing. In the third interpretation, each DDC$^\alpha$ term τ is mapped to a parsing function. This parsing function takes as input a string to be parsed and returns a pair with type (τ_{rep}, τ_{md}).

We precisely characterized the *canonical* relationship between a representation value and a meta-data value for the two to be meaningfully paired. This canonical relation enforces the property that the meta-data structure captures the errors in the representation. We then showed that for every DDC$^\alpha$ term τ, the generated parser returns a representation and a meta-data value that are related via the canonical relation, guaranteeing that the meta-data structure returned by the parser precisely captures the errors detected during parsing.

In addition, we showed how to translate PADS declarations into terms of DDC$^\alpha$ to document precisely the semantics of those declarations. This process allowed us to find several bugs in the PADS/C implementation and guided the design of later versions of PADS including PADS/ML and PADS/HASKELL. Moreover, the DDC$^\alpha$ calculus is general enough that it also allowed us to define formal semantics for interesting elements of other data description languages, including PACKETTYPES [22] and DATASCRIPT [4].

We eventually extended DDC$^\alpha$ to add a fourth semantic function, corresponding to a printing function. We explored under what conditions parsing followed by printing or printing followed by parsing is the identity function [12]. This question is non-trivial because various parsing functions throw away information from the input, such as the number of white space characters between two values. This loss makes it impossible to precisely regenerate the output in all cases. Of course, it would always be possible to change the parser to retain enough information to ensure round-tripping laws for parsing and printing, but it is unclear whether the practical price is worth the theoretical gain.

5. WHAT ELSE YOU GET

A key insight behind the PADS project is that once someone has written a description, it is possible to generate a wide variety of additional tools besides a parser and a printer because the description tells the computer a lot about the data. Each version of PADS can generate a number of such tools fully automatically from any description. We describe a few of the most useful tools below.

Accumulator.

With large data sets, it can be difficult to get a "bird's eye" view of the data, which requires developing a sense of what the data "usually" looks like, what fields have a lot of variation, what the representations for missing values are, *etc.* The accumulator tool is designed to help with this problem. It runs over large volumes of data with the designated format, accumulating a variety of different statistics for each part of the structure. When all the relevant data has been processed, the accumulator generates an informative statistical report. For base types, the accumulator reports information relevant to that type. For example, for integers, the tool reports the minimum, maximum, and average values, as well as a histogram of the most commonly seen values, precisely tracking all values up to a customizable limit. For strings, the accumulator reports the observed lengths of the strings and a histogram of observed values. For structured types, the accumulator tool reports summaries of the components of the types. For lists, it reports the various lengths of the list observed in the input. For datatypes, it reports the relative frequencies of the various branches.

A common use of the accumulator tool is in writing PADS descriptions. It is typical to write an initial description that covers a representative sample of the data source, using string base types to specify poorly-understood portions of the data. From this description, the analyst generates and runs the accumulator tool, which reports both the records in the input that do not match the description and distributions on the place-holder strings. Both pieces of information allow the analyst to refine the description and iterate. This process helps in developing descriptions where the data file is large and has variation throughout the file, making it impossible for human beings to see all the variation without automated assistance.

A demo of a PADS accumulator is available from the website `http://www.padsproj.org/learning-demo.cgi`.

XML Converter.

PADS descriptions typically describe semi-structured data, which makes it natural to represent the same information in XML. Because XML is a standard representation for semi-structured data, there are many tools available to manage XML data. To leverage this infrastructure, we developed a tool to convert any PADS description into a corresponding, format-specific XML Schema and any data matching the PADS description into XML that matches the generated Schema. The PADS website also has a demo of this tool.

Relational Converter.

Although not all PADS descriptions describe essentially relational data, some do, and for such descriptions, it can be useful to convert the raw data into a "cleaned-up" form suitable for loading into a relational database. The common log format we saw in Section 2 is an example of such a data source. Its raw form is difficult to include via a typical database import function because of the extraneous punctuation, but conceptually it is a simple table. The PADS relational converter tool maps the raw data into a delimited column form, where the user can specify the delimiter. We have used this tool at AT&T to import data into the Daytona database system. Again, the PADS website has a demo of this tool.

XQuery Integration.

An obviously useful tool for ad hoc data sources is the ability to query the data. Inventing an entirely new query language for what is essentially semi-structured data seemed like reinventing the wheel. Instead, we decided to develop a tool that would allow analysts to query any data source with a PADS description using XQUERY [18]. An obvious approach to this integration would be to use the XML converter to translate the original data into XML and then run an XQuery implementation on the resulting document. However, the large amount of extra space required to represent the data in XML, typically a factor of eight, led to unacceptable performance with this approach. Instead, we were able to leverage the abstract data model provided by the Galax [10] implementation of XQUERY to enable Galax to query PADS data directly. The original implementation of this tool [9] only allowed query results to be returned in XML, but a subsequent extension allowed results to be mapped back into the original form [8].

Harmony integration.

The Harmony synchronization framework [24] allows two replicas of a document to be synchronized with each other. Internally, Harmony works on unordered trees. Synchronizing particular data formats requires writing viewers to map between the on-disk representation and Harmony's tree model. To avoid having to write such viewers by hand, we wrote a tool to automatically convert any data with a PADS description into the required format and back. Together, PADS and Harmony allow effective, semantics-preserving synchronization of arbitrary ad hoc data sets.

5.1 Implementing tools

The best way to implement these description-specific tools has been the subject of on-going research. In the original implementation of PADS, the compiler generated these tools. This approach gives a lot of flexibility and is fairly straightforward to implement, but it means that only compiler writers can add new tools, a significant limitation.

To address this problem, the PADS/ML compiler generates generic "traversal functions" [8, 21] in addition to the standard type declarations and parsing functions. Using this infrastructure, third-party developers can write tools without having to change the compiler. However, the downside is that the interface to the set of generic functions is extremely complex. The complexities in the interface arose because the host language for PADS/ML, OCAML, does not provide direct support for generic programming, In order to obtain the generic programming facilities we required, we were forced to implement sophisticated type-directed algorithms in the structures and functors supplied by OCAML's powerful module system. The result was a system that is expressive enough to accomplish the desired tasks, but usable only by extreme experts.

One of the motivations for building a version of PADS in Haskell is that Haskell does provide a lot of support for generic programming [19, 25]. We anticipate that writing third party tools in the PADS/HASKELL framework will be significantly easier.

6. SOMETHING FOR FREE

The time and expertise required to write a PADS description from scratch can be a significant impediment to using the system. Depending on the complexity of a data source, it can take hours to days to produce a comprehensive PADS description. To shorten this process, we have developed a system that automatically infers a PADS description from multiple positive examples of the data format [14]. This learning process can be connected to the PADS tool infrastructure to automatically produce tools to generate accumulator reports or XML representations of ad hoc data sources without any human intervention. A demo of this capability is available on the web `http://www.padsproj.org/learning-demo.cgi`.

The inference system works in a series of stages. In the first stage, the input data is broken into chunks, each of which is a positive instance of the data format to be learned. We require the user to tell us how to do this division. Typical examples include breaking a file on newline boundaries or treating each file in a collection of files as an instance.

In the second stage, we convert each chunk into a sequence of tokens, where the collection of possible tokens is specified using regular expressions. By default, the system provides tokens for integers, floats, various kinds of strings, white space, and punctuation. It also provides domain-specific tokens for systems-like data, such as IP addresses, MAC addresses, email addresses, dates, and times. The intuition is that this collection should include atomic pieces of data that a human would glance at and know what it means with 100% confidence. The system is parameterized so users can provide their own set of regular expressions.

In the third stage, the system computes a histogram for each token, counting the number of records in which the token appears zero times, one time, two times, *etc.* Tokens with similar histograms are clustered, based on a similarity metric. The cluster that "best describes" the data is selected, using a heuristic that rewards high coverage, meaning the cluster appears in almost every record, and narrowness, meaning that the tokens in the cluster appear with the same frequency in almost all records. For example, if every record had exactly one comma and two quotation characters, then the comma and two quotation tokens would be clustered and that cluster would be selected. The system next partitions the input based on the selected cluster, with one partition for each observed order for the cluster tokens and an extra partition for the records that do not contain all the tokens in the cluster. This collection of partitions will correspond to a datatype in the eventual description, with one branch for each non-empty partition. (In the case where there is only one partition, this datatype is omitted from the inferred description.) Within a partition, all records have all the tokens in the cluster in the same order. Each such partition will correspond to a record type declaration in the generated description. This record type contains each of the tokens in the appropriate order. To infer the description for the data between these tokens, the system divides each input record into the tokens before the first cluster token, between the first and second cluster token, *etc.* Each of these groups is then recursively analysed to produce a description that is slotted into the top-level record declaration.

In the fourth stage, we greedily search for the best possible description in the nearby area. This search is executed by successively applying rewriting rules to the inferred description and scoring the results of the rewrites. The search continues until it is no longer possible to rewrite the description in such a way as to obtain a new description with a superior score. The scoring function itself uses an information-theoretic measure called the Minimum Description Length (MDL) [16] to evaluate the quality of any description. This measure counts the complexity of the description and the complexity of the data given the description, thereby penalizing both simplistic descriptions (like `String`) that cover the data without adding any information as well as overly complex descriptions, such as the description that specifies each character in order.

6.1 Learning tokenizations

The learning algorithm is very sensitive to how the input is tokenized. For certain basic types, like filepaths, the regular expression that defines legal file paths is very general. Almost every string of characters is a file path, but that does not mean it is *likely* that every string of characters is a file path. It is fairly easy for human beings to look at a data set and identify the filepaths. We explored whether machine-learning techniques could help us develop a tokenizer that could capture this concept effectively [26]. The result of this study was that the learning system required a lot of data and the inference process was slower, but the quality of the inferred descriptions improved. Still, more research in this area may well improve the quality of descriptions relative to the time required to learn them.

6.2 Incremental Inference

The original inferencing algorithm produces a description exclusively from a relatively small, single input data set. Unfortunately, this original design made it impossible to use inference to improve an existing description, to process streaming data or to process large data sets. To address these weaknesses, we are in the process of developing an incremental version of the learning system [28]. In this version, the learning system optionally takes an existing description as input in addition to the data. The output of the system is a new description that covers all the new data while diverging from the original description as little as possible.

This architecture allows us to scale to larger data sets by iterating the inference process. We can either start with a supplied description or use the original learning algorithm to produce a description from a subset of the supplied data. We then divide the remaining data into groups of appropriate size and iteratively apply the incremental algorithm to these groups, eventually returning a description that covers the entirety of the original data set. We are in the process of evaluating the effectiveness of this approach and how well it scales.

6.3 Putting humans in the loop

Even if format inference were perfect, we would still need human involvement to produce the high-quality descriptions. At the very least, it is not possible for the computer to infer meaningful labels for fields in records. For example, the computer cannot tell if a given IP address is a source or a destination. In the end, it is likely that the best system for inferring descriptions will use a combination of machine learning techniques and a high-power user interface that lets humans explore the data and edit descriptions effectively.

One kind of human-friendly interface we have explored in detail is a new sort of *markup language* for raw text [27]. This markup language, called ANNE, helps users interactively generate PADS descriptions with human-readable names and the exact structure desired. Given a new ad hoc data source, an ANNE programmer edits the document to add annotations that are somewhat akin to XML tags, yet contain bits of grammatical information that serve to specify the syntactic structure of the document. These 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, smart defaults, and the raw data itself to extract a PADS description from the document. This PADS description 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. The description can also be saved as documentation or used to generate any of the other PADS tools. Like other languages in the PADS family, ANNE has a formal semantics. This time, the semantics is based on concepts drawn from Relevance Logic [2] as opposed to type theory. We used the semantics to prove a number of interesting properties concerning the expressiveness of ANNE and the conditions under which it is able to extract a desired context-free grammar from a document.

One of the inspirations for ANNE was an earlier system called LAUNCHPADS [6]. This tool used a graphical user interface to help human beings write descriptions. The tool presents users with sample input and provided a tool palette for introducing structure such as lists, datatypes, and records. An integration of a visual tool like LAUNCHPADS or a text-based annotation language like ANNE and the incremental inferencing algorithm is likely the best approach to producing high-quality descriptions quickly.

7. WHAT OTHERS HAVE DONE

There is a vast literature on parsing, so here we only briefly review related work on data description languages. Our paper defining the semantics of PADS contains a detailed discussion of a large body of related work [13]. The interested reader is invited to consult that paper for a more detailed discussion. In addition, each of the PADS papers mentioned here discusses the relevant related work. For more information on particular aspects of the PADS project, consult the relevant paper.

There are many data description languages for designing data formats, including ASN.1 [7] and ASDL [1], or, more recently, Google Protobufs [15] and Apache Avro [3]. These declarative languages allow programmers to describe the *logical representation* of data and then automatically generate a *physical representation* and functions to map between the two representations. Although useful for many purposes, such tools are of little use when the physical representation is already fixed, which is the domain that PADS targets.

Traditional parsing systems such as YACC generate parsers from declarative specifications; however, they are not particularly well suited for writing data descriptions. In particular, such systems generally require users to write a separate lexer and construct in-memory data structures by hand. They typically only work on ASCII data and do not allow data-dependent parsing.

The languages and systems that are most closely related to PADS are PACKETTYPES [22] and DATASCRIPT [4]. Each of these systems allow declarative descriptions of physical data, motivated respectively by the goals of parsing TCP/IP packets and JAVA jar-files. As with PADS, these languages all use a type-directed approach to describing physical data formats and permit the user to specify semantic constraints in a host language. These systems differ from PADS in focusing only on binary data and assuming that the data is error free, halting if an error is detected. In addition, these systems focus on the parsing problem, rather than also providing a body of auxiliary tools. None of these systems attempt to infer descriptions from raw data.

8. WHERE WE GO FROM HERE

Various problems remain open. Developing new tools is still difficult. We anticipate that working in the context of Haskell, which has an active research community in generic and type-directed programming, will help in this area. Format inferencing is reasonably successful, but we believe the inferencing process can be further improved by incorporating more advanced machine-learning techniques and by including the user in strategic decisions. Finally, to get widespread adoption of the technology, it is likely necessary to integrate a version of PADS more tightly into standard tools for manipulating data.

Acknowledgments

The PADS project has been a collaboration involving a large number of people from a variety of institutions, including: Andy Adams-Moran, David Burke, Mark Daly, Zach DeVito, Pamela Dragosh, Mary Fernández, Andrew Forrest, J. Nathan Foster, Artem Gleyzer, Joel Gottlieb, Michael Greenberg, Robert Gruber, Vikas Kedia, John Launchbury, Yitzhak Mandelbaum, Ricardo Medel, Frances Spalding, Peter White, Qian Xi, Xuan Zheng, and Kenny Q. Zhu.

Portions of this material are based upon work supported under NSF grants CCF-1016937, 0238328, 0612147, and 0615062 and supported by DARPA under grant FA8750-07-C-0014. 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 DARPA.

9. REFERENCES

[1] Abstract syntax description language. `http://sourceforge.net/projects/asdl`.

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

[3] Apache. Apache Avro. `http://avro.apache.org/docs/current/`, 2009.

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

[5] Apache Common Log Format. `http://httpd.apache.org/docs/1.3/logs.html`, 2010.

[6] M. Daly, M. Fernández, K. Fisher, Y. Mandelbaum, and D. Walker. LaunchPads: A system for processing ad hoc data. In *Demo at the ACM SIGPLAN Workshop on Programming Language Technologies for XML*, 2006.

[7] O. Dubuisson. *ASN.1: Communication between heterogeneous systems*. 2001.

[8] M. F. Fernández, K. Fisher, J. N. Foster, M. Greenberg, and Y. Mandelbaum. A generic programming toolkit for PADS/ML: First-class upgrades for third-party developers. In *ACM Symposium on Practical Aspects of Declarative Programming*, Jan. 2008.

[9] M. F. Fernández, K. Fisher, R. Gruber, and Y. Mandelbaum. PADX: Querying large-scale ad hoc data with XQuery. In *ACM SIGPLAN Workshop on Programming Language Technologies for XML*, Jan. 2006.

[10] M. F. Fernández, J. Siméon, B. Choi, A. Marian, and G. Sur. Implementing XQuery 1.0: The Galax experience. In *International Conference on Very Large Data Bases*, pages 1077–1080, 2003.

[11] K. Fisher and R. Gruber. PADS: A domain specific language for processing ad hoc data. In *ACM Conference on Programming Language Design and Implementation*, pages 295–304, June 2005.

[12] K. Fisher, Y. Mandelbaum, and D. Walker. A dual semantics for the data description calculus. Available from `http://www.cs.princeton.edu/~dpw/papers/tfp07.pdf`, June 2007.

[13] K. Fisher, Y. Mandelbaum, and D. Walker. The next 700 data description languages. *Journal of the ACM*, 57:10:1–10:51, February 2010.

[14] K. Fisher, D. Walker, K. Zhu, and P. White. From dirt to shovels: Fully automatic tool generation from ad hoc data. In *ACM Symposium on Principles of Programming Languages*, Jan. 2008.

[15] Google. Protocol buffers. `http://code.google.com/p/protobuf/`, 2010.

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

[17] T. Jim, Y. Mandelbaum, and D. Walker. Semantics and algorithms for data-dependent grammars. In *ACM Symposium on Principles of Programming Languages*, pages 417–430, New York, NY, USA, 2010. ACM.

[18] H. Katz, editor. *XQuery from the experts*. Addison Wesley, 2004.

[19] R. Lämmel and S. P. Jones. Scrap your boilerplate: A practical design pattern for generic programming. In *ACM International Workshop on Types in Language Design and Implementation*, pages 26–37, New York, NY, USA, 2003. ACM.

[20] G. Mainland. Why it's nice to be quoted: Quasiquoting for Haskell. In *ACM Workshop on Haskell*, pages 73–82, New York, NY, USA, 2007. ACM.

[21] Y. Mandelbaum, K. Fisher, D. Walker, M. Fernández, and A. Gleyzer. PADS/ML: A functional data description language. In *ACM Symposium on Principles of Programming Languages*, Jan. 2007.

[22] P. McCann and S. Chandra. PacketTypes: Abstract specificationa of network protocol messages. In *ACM Conference of Special Interest Group on Data Communications*, pages 321–333, August 2000.

[23] PADS project. `http://www.padsproj.org/`, 2010.

[24] B. C. Pierce, A. Bohannon, J. N. Foster, M. B. Greenwald, S. Khanna, K. Kunal, and A. Schmitt. Harmony: A synchronization framework for heterogeneous tree-structured data. `http://www.seas.upenn.edu/~harmony/`.

[25] P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad hoc. In *ACM Symposium on Principles of Programming Languages*, pages 60–76, New York, NY, USA, 1989. ACM.

[26] Q. Xi, K. Fisher, D. Walker, and K. Q. Zhu. Ad hoc data and the token ambiguity problem. In *ACM Symposium on Practical Aspects of Declarative Programming*, pages 91–106, Berlin, Heidelberg, 2009. Springer-Verlag.

[27] Q. Xi and D. Walker. A context-free markup language for semi-structured text. In *ACM Conference on Programming Language Design and Implementation*, pages 221–232, 2010.

[28] K. Q. Zhu, K. Fisher, and D. Walker. Incremental learning of system log formats. *SIGOPS Operating System Review*, 44:85–90, March 2010.