PADS: Processing Arbitrary Data Streams

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

Robert E. Gruber
AT&T Labs — Research
gruber@research.att.com

August 7, 2003

1 Introduction

Transactional data streams, such as sequences of stock-market buy/sell orders, credit-card purchase records, web server entries, and electronic fund transfer orders, can be mined very profitably. As an example, researchers at AT&T have built customer profiles from streams of call-detail records to significant financial effect [CP98, CP99, CFP+00].

Often such streams are high-volume: AT&T's call-detail stream contains roughly 300 million calls per day requiring approximately 7GBs of storage space. Typically, such stream data arrives "as is" in *ad hoc* formats with poor documentation. In addition, the data frequently contains errors. The appropriate response to such errors is application-specific. Some applications can simply discard unexpected or erroneous values and continue processing. For other applications, however, errors in the data can be the most interesting part of the data.

Understanding a new data stream and producing a suitable parser are crucial first steps in any use of stream data. Unfortunately, writing parsers for such data is a difficult task, both tedious and error-prone. It is complicated by lack of documentation, convoluted encodings designed to save space, the need to handle errors robustly, and the need to produce efficient code to cope with the scale of the stream. Often, the hard-won understanding of the data ends up embedded in parsing code, making long-term maintenance difficult for the original writer and sharing the knowledge with others nearly impossible.

The goal of the PADS project is to provide languages and tools for simplifying data stream analysis. We have a preliminary design of a declarative data-description language, PADSL, expressive enough to describe the data feeds we see at AT&T in practice, including ASCII, binary, EBCDIC, Cobol, and mixed data formats. From PADSL we generate a tunable C library with functions for parsing, manipulating, and summarizing the data.

2 PADS language

Intuitively, a PADSL description specifies complete information about the physical layout and semantic constraints for the associated data stream. Most type declarations in PADSL are analogous to type declarations in C. PADSL has an extensible set of base types that specify how to read and verify atomic pieces of data such as ASCII 32-bit integers (`Pa_int32`) and binary bytes (`Pb_int8`). Verification conditions for such base types include checking that the resulting number fits in the indicated space, *i.e.*, 16-bits for `Pa_int16`. PADSL has **`Pstruct`**s, **`Punion`**s, and **`Parray`**s to describe record-like structures, alternatives, and sequences, respectively. Each of these types can have an associated predicate that indicates whether a value calculated from the physical specification is indeed a legal value for the type. For example, a predicate might require that two fields of a **`Pstruct`** are related or that the elements of a sequence are in increasing order. Programmers can specify such predicates using PADSL expressions or functions. PADSL **`Ptypedef`**s can be used to define new types that add further constraints to existing types.

In addition, PADSL types can be parameterized by values. This mechanism serves both to reduce the number of base types and to permit the format of later portions of the data to depend upon earlier portions. For example, the base type `Pa_uint32_FW(:3:)` specifies an unsigned integer physically represented by exactly 3 ASCII characters, while the type `Pa_string(:' ':)` describes an ASCII string terminated by a space. Parameters can be used with compound types to specify the size of an array or which branch of a union should be taken.

As an example, consider the common log format for Web server logs. A typical record looks like the following:

```
207.136.97.49 - - [15/Oct/1997:18:46:51 -0700] "GET /tk/p.txt HTTP/1.0" 200 30
```

recording the IP address of the requester; either a dash or the owner of the TCP session; either a dash or the login of the requester; the date; the actual request, which consists of the HTTP method, the requested URL, the HTTP version number; a response code; and the number of bytes returned. A PADSL type describing the request portion is

```
Pstruct http_request_t {
  '\"'; http_method_t    meth;           /- Method used during request
  ' ';  Pa_string(:' ':) req_uri;        /- Requested uri.
  ' ';  http_v_t         version : checkVersion(version, meth);
                                         /- HTTP version number of request
  '\"';
};
```

This **Pstruct** uses (omitted) auxiliary types `http_method_t` and `http_v_t` to describe the HTTP method and version formats, respectively. It uses character literals (`'\"'` and `' '`) to consume the quotes and spaces from the physical representation. The `version` field has a constraint predicate `checkVersion` which ensures that obsolete HTTP methods `LINK` and `UNLINK` are only used with HTTP version `1.0`.

3 Generated library

From each type in a PADSL description, we generate C declarations for (1) an in-memory representation, (2) a *check-set*-mask, which allows users to specify which portions of the data are relevant to their applications, (3) an error-description, which we use to describe physical and semantic errors detected during parsing, (4) a parse function, and (5) utility functions. The parse function takes as arguments pointers to a checkset-mask, an in-memory representation, and an error-description. The generated library maintains the invariant that if the checkset-mask requests that a data item be verified and set, and if the error description indicates no error, then the in-memory representation satisfies the semantic constraints on the data.

The checkset-mask allows the user to specify with fine granularity which constraints the parser should check and which portions of the in-memory representation it should fill in. This control allows the description-writer to specify all known constraints about the data without worrying about the run-time cost of verifying potentially expensive constraints for time-critical applications.

Appropriate error-handling can be as important as processing error-free data. The error descriptor marks which portions of the data contain errors and characterizes the detected errors. Depending upon the nature of the errors and the desired application, programmers can take the appropriate action: halting the program, discarding parts of the data, or repairing the errors.

By supporting multiple entry-points, we accommodate larger-scale data. For a small file, programmers can define a PADSL type that describes the entire file and use that type's parsing function to read the whole file with one call. For larger-scale data, programmers can sequence calls to parsing functions that read manageable portions of the file, *e.g.*, reading a record at a time in a loop.

4 Related work

There are many tools for describing data formats. For example, ASN.1 [Dub01] and ASDL [asd] are both systems for declaratively describing data and then generating libraries for manipulating that data. In contrast to PADS, however, both these systems specify the *logical* representation and automatically generate a *physical* representation. Although useful for many purposes, this technology does not help process data that arrives in predetermined, *ad hoc* formats.

More closely related work allows declarative descriptions of physical data [MC98, erl, Bac02], motivated by parsing TCP/IP packets and JAVA jar-files. In contrast to our work, these systems only handle binary data and assume the data is error-free or halt parsing if an error is detected.

5 Conclusion

Further information and a source-code distribution of PADS is available from:

`http://www.research.att.com/projects/pads`

References

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

[Bac02] Back, G. DataScript - A specification and scripting language for binary data. In *Proceedings of Generative Programming and Component Engineering*, vol. 2487. LNCS, 2002, pp. 66–77.

[CFP+00] Cortes, C., K. Fisher, D. Pregibon, A. Rogers, and F. Smith. Hancock: A language for extracting signatures from data streams. In *Proceedings of the Sixth International Conference on Knowledge Discovery and Data Mining*, 2000, pp. 9–17.

[CP98] Cortes, C. and D. Pregibon. Giga mining. In *Proceedings of the Fourth International Conference on Knowledge Discovery and Data Mining*, 1998.

[CP99] Cortes, C. and D. Pregibon. Information mining platform: An infrastructure for KDD rapid deployment. In *Proceedings of the Fifth International Conference on Knowledge Discovery and Data Mining*, 1999.

[Dub01] Dubuisson, O. *ASN.1: Communication between heterogeneous systems*. Morgan Kaufmann, 2001.

[erl] Proposals for and experiments with an Erlang bit syntax. http://lambda.weblogs.com/discuss/msgReader$5185.

[MC98] McCann, P. and S. Chandra. PacketTypes: Abstract specification of network protocol messages. In *ACM Conference of Special Interest Group on Data Communications (SIGCOMM)*, August 1998, pp. 321–333.