PADS/T:
A Language for Describing and Transforming Ad Hoc Data

Mary Fernandez and Kathleen Fisher, AT&T Research
Yitzhak Mandelbaum and David Walker, Princeton University

September 16, 2005

Abstract

PADS/T is a declarative data description language paired with an error-aware transformation language. This pairing provides rich support for programming with ad hoc data sources. Such data is common in a wide range of domains including networking, financial analysis, biology, and physics. PADS/T's data description language is based on polymorphic, recursive, and dependent data types. From such descriptions, the compiler generates robust parsing and pretty printing code. The transformation language supports *error-aware computation* by automatically maintaining an association between data and a description of the errors in that data.

1 Introduction

An *ad hoc data format* is any non-standard data format for which parsing, querying, analysis, or transformation tools are not readily available. Despite the increasing use of standard data formats such as XML, ad hoc data sources continue to arise in numerous industries such as finance, health care, transportation, and telecommunications as well as in scientific domains, such as computational biology and chemistry. The absence of tools for processing ad hoc data formats complicates the daily data-management tasks of data analysts, who may have to cope with numerous ad hoc formats even within a single application.

Common characteristics of ad hoc data complicate the building of tools to perform even basic data-processing tasks. Documentation of ad hoc formats, for example, is often incomplete or inaccurate, making it difficult to define a database schema for the data or to build a reliable data parser. The data itself often contains numerous kinds of errors, which can thwart standard database loaders. Surprisingly, few meta-language tools, such as data-description languages or parser generators, exist to assist in management of ad hoc data. And although ad hoc data sources are among the richest for database and data mining researchers, they often ignore such sources as the work necessary to clean and vet the data is prohibitively expensive.

The variety of application domains, format characteristics, sources of errors, and volume of data makes processing ad hoc data both interesting and challenging. Figure 1 summarizes several ad hoc sources from the networking and telecommunication domains at AT&T and from computational biology applications at Princeton. Formats include ASCII, binary, and Cobol, with both fixed and variable-width records arranged in linear sequences and in tree-shaped or DAG-shaped hierarchies. Even within one format, there can be a great deal of syntactic variability. For example, Figure 2 contains two records from the network-monitoring application. Note that each record has different number of fields (delimited by '|') and that individual fields contain structured values (e.g., attribute-value pairs separated by '=' and delimited by ';').

Name: Use	Record Format (Size)	Common Errors
Web server logs (CLF): Measuring Web workloads	Fixed-column ASCII ($\leq$12GB/week)	Race conditions on log entry Unexpected values
AT&T provisioning data (Sirius): Monitoring service activation	Variable-width ASCII (2.2GB/week)	Unexpected values Corrupted data feeds
Call detail: Fraud detection	Fixed-width binary (˜7GB/day)	Undocumented data
AT&T billing data (Altair): Monitoring billing process	Cobol (>250GB/day)	Unexpected values Corrupted data feeds
IP backbone data (Regulus): Network Monitoring	ASCII ($\geq$ 15 sources,˜15 GB/day)	Multiple representations of missing values Undocumented data
Netflow: Network Monitoring	Data-dependent number of fixed-width binary records ($\geq$1Gigabit/second)	Missed packets
Gene Ontology data: Gene-gene correlations in Magic	Variable-width ASCII in DAG-shaped hiearchy	
Newick data Immune system response simulation	Fixed-width ASCII in tree-shaped hierarchy	Manual entry errors

Figure 1: Selected ad hoc data sources.

```
2:3004092508||5001|dns1=abc.com;dns2=xyz.com|c=slow link;w=lost packets|INTERNATIONAL
3:|3004097201|5074|dns1=bob.com;dns2=alice.com|src_addr=192.168.0.10;
dst_addr=192.168.23.10;start_time=1234567890;end_time=1234568000;cycle_time=17412|SPECIAL
```

Figure 2: Simplified network-monitoring data. Newlines inserted for legibility.

Unfortunately, data analysts have little control of the format of ad hoc data at its source nor at its final destination, for example, in a database. The data arrives "as is", and the analyst who receives it can only thank the supplier, not request a more convenient format. Once an analyst has the data, a common task is converting the source format to a standard database loading format. This transformation proceeds in three stages. First, the analyst writes a parser for the ad hoc format, using whatever (in)accurate documentation may be available. Second, he writes a program that detects and handles erroneous data records, selects records of interest, and possibly normalizes records into a standard format, for example, by reordering, removing, or transforming fields. Unfortunately, the parsing, error handling, and transformation code is often tightly interleaved. This interleaving hides the knowledge of the ad hoc format obtained by an analyst and severely limits the parser's reuse in other applications.

Another challenge is the variety of errors and the variety of application-dependent strategies for handling errors in ad hoc data. Some common errors, listed in Figure 1, include undocumented data, corrupted data, missing data, and multiple representations for missing values. Some sources of errors that we have encountered in ad hoc sources include malfunctioning equipment, race conditions on log entry [16], presence of non-standard values to indicate "no data available," human error when entering data, and unexpected data values. A wide range of responses are possible when errors are detected, and they are highly application dependent. Possible responses range from halting processing and alerting a human operator, to partitioning erroneous from valid records for examination off-line, to simply discarding erroneous or unexpected values. One of the most challenging aspects of processing ad hoc data is that erroneous data is often more important than error-free data, because it may indicate, for example, that two systems are failing to communicate. Writing code that is reliably *error aware*, however, is difficult and tedious.

The high volume of ad hoc data sources is another challenge. Figure 1 gives the volume of several sources.

AT&T's call-detail stream, for example, contains roughly 300 million calls per day requiring approximately 7GBs of storage space. Although this data is eventually archived in a database, data analysts mine it profitably before such archiving [6, 7]. More challenging, the Altair project at AT&T accumulates billing data at a rate of 250-300GB/day, with occasional spurts of 750GBs/day, and netflow data arrives from Cisco routers at rates over a Gigabit per second [8]! Such volumes require that the data be processed without loading it into memory all at once. Not surprisingly, flexible error-response strategies are especially critical with high-volume sources, so that error detection does not halt or delay normal processing.

Commercial data-management products for ad hoc data address some of these problems, but to our knowledge, none can handle all the variability in formats that we have encountered, nor do they support error-aware processing of high volume sources. Without tools adequate to the task, analysts often write custom programs in C or PERL. Unfortunately, writing parsers, transformers, and printers by hand is tedious and error-prone. These tasks are complicated by 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.

1.1 PADS/T

Our answer to processing ad hoc data is PADS/T, a functional programming language with extensive support for describing and transforming ad hoc data. PADS/T's core includes standard functional features such as pattern matching and higher-order functions, which we view as critical to supporting data-driven transformation. PADS/T's novel data-description language is based on a rich system of polymorphic, recursive, and dependent types. In addition, PADS/T allows programmers to enforce semantic constraints on data by using PADS/T descriptions as a special form of runtime contracts.

Most importantly, PADS/T is not merely a combination of two language elements—data description and transformation—but a synthesis. The meta-data acquired during parsing of data is not lost once parsing is finished, but instead plays a prominent role in the programmatic elements of the language. As we are particularly concerned with error-related meta-data, we call this synthesis *error-aware computing*, the central contribution of this work.

PADS/T has evolved from PADS [12], a data-description language for ad hoc data with a C-like syntax, and PADS/T's implementation is based on the PADS system [1]. Given a PADS description, the PADS compiler generates a parsing library that can be called from a C program. The parsing functions can detect and report errors in data records and can recover from non-fatal errors, but the programmer himself must write C code for partitioning erroneous from error-free records, for correcting errors, and for transforming records. In contrast, PADS/T is a high-level language with an elegant and convenient syntax for data-driven programming and intrinsic support for handling errors.

Figure 1.1 depicts a common use of the PADS/T architecture[2]. On the upper left, the format of an ad hoc input source is specified by a user in the PADS/T data description language. From this description, the PADS/T description compiler produces a custom parser for instances of the ad hoc input. Symmetrically, on the upper-right, the data description of the output data is compiled into a custom printer for instances of the ouput. A PADS/T transformation program refers to the types defined by the input and output descriptions, and the PADS/T runtime environment for the transformation program calls the corresponding parsers and printers.

[1]PADS can be downloaded from `http://www.padsproj.org`

[2]The green boxes denote code written by users.

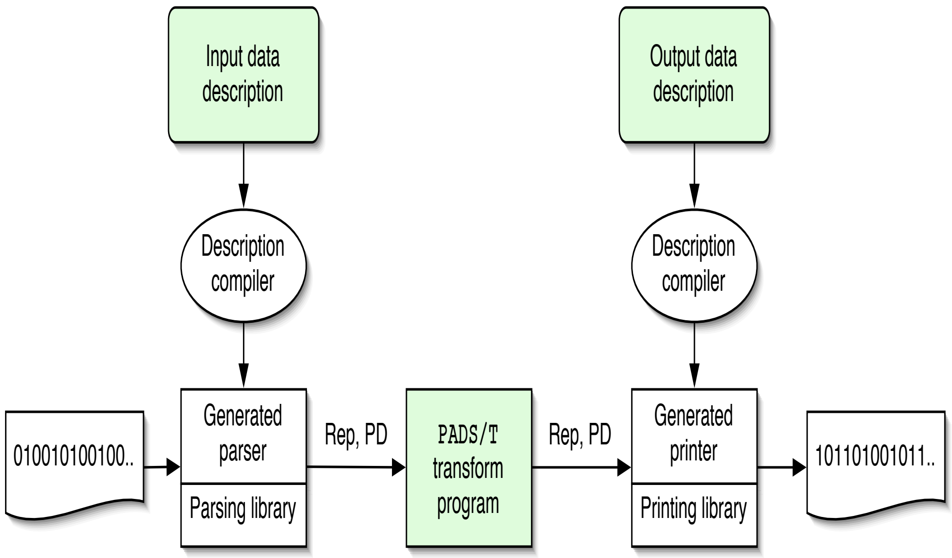

Figure 3: Common use of PADS/T architecture

The key abstraction in PADS/T that supports error-aware computing is the *fused value*, which pairs a data item with its corresponding meta data. Meta data includes, among other things, descriptions of the error content of the data item. In Figure 1.1, a fused value consists of the data item's *representation* (Rep) and its *parse descriptor* (PD), which contains the meta data.

PADS/T's type system ensures that data items and their associated meta data are always consistent. This error-aware computation enables three important language features:

1. Safe, error-transparent transformations that need only specify how to handle error-free data.
2. Selective querying of errors, which supports easy extraction of detailed error profiles.
3. Flexible, programmatic repair of erroneous data, which supports disciplined and identifiable correction of errors.

In Section 2, we introduce PADS/T's data description language using examples from the applications described in Figure 1 and in Section 3, we do the same for PADS/T's data-transformation constructs. Section 4 briefly surveys related work, and in Section 5, we enumerate only a few of the numerous open problems that await us in PADS/T's future.

2 Describing Data in PADS/T

A PADS/T description specifies the physical layout and semantic properties of an ad hoc data source. These descriptions are composed of types: base types describe atomic data, while structured types describe compound data built from simpler pieces. Examples of base types include 8-bit unsigned integers (`Puint8`),

```
0|1005022800
9152|9152|1|9735551212|0||9085551212|07988|no_ii152272|EDTF_6|0|APRL1|DUO|
9153|9153|1|0|0|0|0||152268|LOC_6|0|FRDW1|DUO|
```

Figure 4: Miniscule example of Sirius data.

32-bit integers (`Pint32`), binary 32-bit integers (`Pbint32`), dates (`Pdate`), strings (`Pstring`), and IP addresses (`Pip`). Semantic conditions for such base types include checking that the resulting number fits in the indicated space, *i.e.*, 16-bits for `Pint16`.

Base types (and other types) may be parameterized by values. This mechanism serves both to reduce the number of base types and to permit the format and properties of later portions of the data to depend upon earlier portions. For example, the base type `Puint16_FW(3)` specifies an unsigned two byte integer physically represented by exactly three characters. The base type `Pstring` takes a string indicating the set of acceptable terminator characters. Hence, `Pstring(";|")` can be terminated by either a semicolon or a vertical bar.

To describe more complex data, PADS/T provides a collection of type constructors derived from the type structure of functional programming languages such as Haskell and ML. The following subsections will explain these structured types through a series of real-world examples. Readers eager to see the complete syntax of types should flip forward to Appendix A.1.

2.1 Simple Structured Descriptions

The bread and butter of any PADS/T description are the simple structured types: tuple, record and array types for expressing sequences of data; sum types, realized here as datatypes, for expressing alternatives; and singleton types for expressing the placement of particular characters in the data. In this section, we explain each of these items by building a description of the Sirius data presented in Figure 5.

The Sirius data source is used to record summaries of phone orders produced at AT&T. Each summary involves a date and one record per order. Each order record contains a header followed by a sequence of events. The header has 13 pipe separated fields: the order number, AT&T's internal order number, the order version, four different telephone numbers associated with the order, the zip code of the order, a billing identifier, the order type, a measure of the complexity of the order, an unused field, and the source of the order data. Many of these fields are optional, in which case nothing appears between the pipe characters. The billing identifier may not be available at the time of processing, in which case the system generates a unique identifier, and prefixes this value with the string "no_ii" to indicate the number was generated. The event sequence represents the various states a service order goes through; it is represented as a new-line terminated, pipe separated list of state, timestamp pairs. There are over 400 distinct states that an order may go through during provisioning. The sequence is sorted in order of increasing timestamps.

Figure 5 gives a PADS/T description for the Sirius phone order summaries in our syntax. Overall, the description is a sequence of type definitions. It is probably easiest to understand the data source by reading these definitions bottom up.

The last type definition `source` is intended to be a definition of an entire Sirius data source. It states that a `source` is a `summary_header` followed by a sequence of objects made up of an `order_header` followed by `events`. The tuple type constructor (`T1 ** T2`) and the array type constructor (`T Parray(sep,term)`) both specify sequences of objects in a data source. The `Parray` type depends upon two value parameters, (`sep` and `term`). The first parameter describes the syntactic separators that may be found between elements

```
type summary_header = "0|" ** Puint32 ** Peor

datatype dib_ramp =
  Ramp of Puint64
| GenRamp of "no_ii" ** Puint64

type order_header = {
       order_num       : Puint32;
 '|';  att_order_num  : Puint32;
 '|';  ord_version     : Puint32;
 '|';  service_tn      : pn_t Popt;
 '|';  billing_tn      : pn_t Popt;
 '|';  nlp_service_tn : pn_t Popt;
 '|';  nlp_billing_tn : pn_t Popt;
 '|';  zip_code        : Pzip Popt;
 '|';  ramp             : dib_ramp;
 '|';  order_sort      : Pstring("|");
 '|';  order_details   : Puint32;
 '|';  unused           : Pstring("|");
 '|';  stream           : Pstring("|");
 '|'
}

type event  = Pstring('|') **  '|' ** Puint32
type events = event Parray('|',Peor)
type source = summary_header **
                  (order_header ** events) Parray(Peor,Peof)
```

Figure 5: PADS/T description for Sirius provisioning data.

of the array. In this case `Peor`, the *end-of-record* character, may be found between each element of the array. In this case, the end-of-record character should be set to a newline character as there is one record per line in the file. The second parameter is the terminator for the array. In this case, the terminator is `Peof`, the end-of-file marker. It is often necessary to have additional termination conditions for arrays, such as termination dependent on the number of elements read so far, but we omitted this additional option for expository purposes.

The definition of `events` indicates that this part of the Sirius data will contain a sequence of `events` separated by vertical bars and terminated by an end-of-record character. Each `event` is a string terminated by a vertical bar, followed by a vertical bar and ending with an unsigned 32-bit integer. The interesting part of this sequence is the presence of the type `'|'`. In type-theoretic terms, this is a *singleton type*. It states that one should expect exactly the character `'|'` in the input stream at this point. Other singletons appear in the summary header type as `"0|"` and `Peor`. Any expression that appears embedded in a type as a singleton must be effect-free.

The type `order_header` is a record type that indicates the data format involves the sequence of items described by the fields of the record. Notice that there are two different sorts of fields: anonymous fields (a second form of singleton type) containing directives to parse a particular character (`'|'`) or string (`"0|"`) and fields with names.

The last interesting feature in the Sirius example is the datatype definition of `dib_ramp`. It describes two alternatives for a portion of data, either an integer alone or the fixed string `"no_ii"` followed by an integer. To parse data in this format, the parser will attempt to parse the first branch and only if it fails will it attempt to parse the second branch. Notice how this semantics differs from similar constructs in regular expressions and context-free grammars, which non-deterministically choose between alternatives. Fortunately, we have yet to come across an ad hoc data source where we wish we had nondeterministic choice.[3]

2.2 Recursive Descriptions

The original PADS design contained analogues of all the simple type constructors described in the previous section. Hence, although the syntax of types we just saw is more compact and more elegant than the C-style syntax of PADS, we have yet to add any expressive power. The first real step forward for PADS/T is the introduction of recursion through the data type mechanism. While recursion never appeared in the telecommunications and networking data sources PADS was initially designed for, it appears essential for describing a number of biological data sources we have encountered.

A representative example of such data comes courtesy of Steven Kleinstein, program coordinator of Princeton's Picasso project for interdisciplinary research in computational sciences. Kleinstein is in the process of building a simulator to study the proliferation of B lymphocytes during an immune response. Data needed for his simulations is represented in a variant of the Newick format, which is a flat representation of trees used by many biologists [3]. In Newick, leaves of the tree are string labels followed by a colon and a number. A parent node in the tree introduces a collection of children by placing a sequence of trees within parens. Following the parens is a colon and a number, as is the case for the leaf node. The numbers represent the "distance" that separates the child from the parent. In Kleinstein's case, the distance is the number of mutations that occur in the antibody receptor genes of B lymphocytes. Each line of a file may contain a different tree, terminated by a semi-colon.

Figure 6 gives a description of Newick and a short fragment of example data. Despite the relative complexity of the structure of the data, the description is remarkably concise and elegant. Notice that the

[3] PADS can recognize string data based on regular expressions. Non-determinism here has been useful, but as it has been confined to parsing elements of the `Pstring` base type, it has had no impact on the overall parsing algorithm.

```
val COMMA  = ','
val COLON  = ':'
val SEMI   = ';'
val LPAREN = '('
val RPAREN = ')'

type entry = Pstring(COLON) ** COLON ** Pfloat32

datatype tree =
    Tree of LPAREN ** tree Parray(COMMA,RPAREN) ** "):" ** Puint32
  | Tip of entry

type source = (tree ** SEMI) Parray(Peor,Peof)
```

Tiny fragment of Newick data:

```
(((erHomoC:0.28006,erCaelC:0.22089):0.40998,(erHomoA:0.32304,
(erpCaelC:0.58815,((erHomoB:0.5807,erCaelB:0.23569):0.03586,
erCaelA:0.38272):0.06516):0.03492):0.14265):0.63594,(TRXHomo:0.65866,
TRXSacch:0.38791):0.32147,TRXEcoli:0.57336);
```

Figure 6: Simplified tree-shaped Newick data

```
2:3004092508||5001|dns1=abc.com;dns2=xyz.com|c=slow link;w=lost packets|INTERNATIONAL
3:|3004097201|5074|dns1=bob.com;dns2=alice.com|src_addr=192.168.0.10;
dst_addr=192.168.23.10;start_time=1234567890;end_time=1234568000;cycle_time=17412|SPECIAL
```

Figure 7: Simplified network monitoring data. This data containts two alarm records. Extra newlines were inserted mid-record so the data would fit on a page.

data type definition of `tree` is recursive — we have not been able to find any effective description of this data source without it.

2.3 Parameterized Descriptions

The last key feature of PADS/T is the ability to define parameterized descriptions. In PADS, description parameters were limited to values. In PADS/T, descriptions maybe parameterized by both values and types. Type parameterization allows descriptions to be reused at a variety of different types. This feature helps make data descriptions more concise and allows programmers to define convenient libraries of reusable description components.

Our final example, shown in Figure 7, will illustrate the usefulness of descriptions parameterized by both types and values and introduce a couple of other useful features as well. This example displays alarm data recorded by a network link monitor used by the Regulus project at AT&T. Each alarm signals some problem with a network link.

Now, at this point, we could write out an English description of the Regulus data, and indeed, in an earlier version of this paper we did so. However, English is a terrible specification language for ad hoc data, and every description we have attempted to give has not only been imprecise, it has also made for some very boring reading. Hence, we dispense with the informal English and head straight for the PADS/T description, which is shown in Figure 8.

One of the interesting facets of this data source is the fact that it contains multiple different types of name-value pairs. In the second definition from the top, `pnvp`, we take advantage of both the type and value parameterization of types to encode all the different variations.[4] The type parameters are specified to the left of the type name, as is customary in ML. To the right of the type name in parentheses, we give the value parameter and its type. In the case of `pnvp`, there is a type parameter name `'a` and a value parameter named `f`. Informally, `'a pnvp(f)` is a name-value pair where the value has type `'a` and the name must satisfy the predicate `f`. More precisely, `pnvp` is defined to be a record with three fields. The first field of the record has been given a constrained type. In general, constrained types have the form `{x:T | M}` where `M` is an arbitrary pure boolean expression. Data `d` satisfies this description if it satisfies `T` and boolean `M` evaluates to true when the fused value `d` is substituted for `x`. If the boolean evaluates to false, the data contains a semantic error.

The `pnvp` type is used in both of the following type definitions. In the case of the `nvp` definition, the predicate is instantiated with a test for a specific string but the type parameter remains abstract. In the `nvp_a` definition, the name can be arbitrary, but the value must have type string. Later in the description, `nvp`'s are used with ip addresses, timestamps and integers.

The Regulus description also introduces two new forms of datatypes. The first new form appears in the `info` type. This datatype is parameterized by an `alarm_code`. Rather than determine the branch of the datatype to choose based on the data about to be parsed, the decision is made based on the `alarm_code`, data which has likely been extracted from some previous point in the source. More specifically, if the alarm code is `5074`, the format specification given by the `Details` constructor will be used to parse the current data. Otherwise, the format given by the `Generic` constructor will be used to parse the current data.

The second new form of datatype appears in the `service` type. Here, the constructors specify no types for their arguments. Our convention in this case is to look for string data that matches the name of the constructor. For example, the constructor `DOMESTIC` is associated with the string "DOMESTIC." This type could have been specified (more verbosely) using the ordinary form for datatypes together with singletons:

```
datatype service =
    DOMESTIC      of "DOMESTIC"
  | INTERNATIONAL of "INTERNATIONAL"
  | SPECIAL       of "SPECIAL"
```

2.4 Complete Syntax

Appendix A.1 presents the complete syntax of PADS/T data descriptions.

3 Transforming Data in PADS/T

As we have seen, PADS/T descriptions make it easy to specify the format of ad hoc data sources and then to have the PADS/T compiler generate parsers for the data. Once a data source has been parsed,

[4]This fragment of code contains both types `Pstring` and `string`. These types are subtly different as the first is the type of a fused value containing both a string representation and a parse descriptor whereas the second is an ordinary string. The reader can safely ignore the distinction for now.

```
type timestamp = Ptimestamp_explicit_FW(10, "%S", GMT)

type 'a pnvp(f:Pstring -> bool) =
      { name : {name : Pstring("=") | f name};
        '=';
        val : 'a }
type 'a nvp(name:string) = 'a pnvp(fn s = (s = name))
type nvp_a = Pstring(";|") pnvp(fn s = true)

type details = {
      source      : Pip nvp("src_addr");
';';  dest        : Pip nvp("dest_addr");
';';  start_time  : timestamp nvp("start_time");
';';  end_time    : timestamp nvp("end_time");
';';  cycle_time  : Puint32 nvp("cycle_time")
}

datatype info(alarm_code : Puint32) =
  case alarm_code of
    5074 => Details of details
  | _    => Generic of nvp_a Parray(';', '|')

datatype service = DOMESTIC | INTERNATIONAL | SPECIAL

type raw_alarm = {
       alarm    : { alarm : Puint32 | alarm = 2 orelse alarm = 3};
 ':';  start    : timestamp Popt;
 '|';  clear    : timestamp Popt;
 '|';  code     : Puint32;
 '|';  src_dns  : Pstring(";|") nvp("dns1");
 ';';  dest_dns : Pstring(";|") nvp("dns2");
 '|';  info     : info(code);
 '|';  service  : service
}

fun checkCorr(ra:raw_alarm):bool =
  case ra of
    {alarm=2; start=SOME(_); clear=NONE;...} => true
  | {alarm=3; start=NONE;    clear=SOME(_);...} => true
  |  _ => false

type alarm = {x:raw_alarm | checkCorr x}

type source = alarm Parray(Peor, Peof)
```

Figure 8: Description of Regulus data.

a natural desire is to transform such data to make it more amenable to further analysis. For example, analysts often need to convert ad hoc data into a form suitable for loading into an existing system, such as a relational database or statistical analysis pacakge. Desired transformations include removing extraneous literals, inserting delimiters, dropping or reordering fields, and normalizing the values of fields (*e.g.* converting all times into a specified time zone). Because relational databases typically cannot store unions directly, another important transformation is to convert data with variation (*i.e.*, datatypes) into a form that such systems can handle. Typically, there are two choices for such a transformation. The first is to chop the data into a number of relational tables: one table for each variation. This approach is called *shredding*. The second is to create an "uber" table, with one "column" for each field in any variation. If a given field is not in a particular variation, it is marked as missing.

Typically, these transformations are fairly straightforward, except they must all be done in the context of errors. If analysts choose to ignore possible errors in the data, then the transformations may be very simple to code, but the analysts run the risk of having those errors corrupt valuable data. If they insert code to handle errors, that error processing code can swamp the semantically interesting transformations.

Errors themselves provide the second major motivation to support data transformation, as error analysis, repair, and removal are important tasks in understanding and using an ad hoc data source.

In designing the transformation language PADS/T, our goal is to allow data analysts to complete their transformation tasks as concisely as possible while preserving the integrity of their data. This goal leads to a collection of principles:

Obliviousness For transformations not concerned with errors, we should be able to specify the transformation without reference to errors. The language should ensure, however, that error values are appropriately propagated "under the covers," so that later code may determine that a given piece of data is erroneous.

Reification For transformations concerned with errors, it should be easy to examine the error characteristics of a piece of data and to make choices depending upon those characteristics.

Soundness Programmers cannot accidentally (or maliciously) associate the wrong error characterization with a piece of data. For instance, if data contains a syntactic error, the associated description must say so.

Together, these principles define what we call *error-aware* computing.

3.1 Error-aware computing

PADS/T supports error-aware computing by leveraging the idea of *descriptors*. The result of a parse is a pair of an in-memory representation of the parsed data and a descriptor, which characterizes the error structure of the data. Instead of throwing this descriptor away immediately after parsing, the language preserves it during transformation. At any point in the computation, the descriptor associated with a data item describes the errors in that data item. We use the term *fused value* to denote such a pair. Note that this integrated value pairs a descriptor with a data element at every level of the data structure. That is, the subcomponents of a data structure (if any) are themselves fused values. PADS/T carefully controls the use of fused values to maintain the invariant that a fused value's descriptor accurately describes the representation. Critical to this last feature is PADS/T's type system, which we designed with this invariant in mind. To manipulate fused values, PADS/T provides constructors and destructors (*i.e.*, patterns) designed to facilitate programming with fused values in a natural way.

To preserve soundness, we do not permit programmers to pair descriptors with representations to create fused values. However, we do provide a generic pattern for all fused values, `pat<<pdpat>>`, to extract and read a value's descriptor. In this pattern, the right subpattern matches only the descriptor, while the left subpattern matches the entire fused value. There are currently five possible values for a descriptor: `G`, `B`,`N`, `S`, and `U`. The description `G` describes data that is error free; `B` describes data that is entirely invalid; `N` describes data with an error in one or more subcomponents; `S` describes data with a semantic error, *i.e.* data that violates a constraint written in a constrained type. The final description `U` describes data for which a top-level descriptor is incomplete. This case arises when parsing a potentially infinite stream, for example. In this case, the user must query the descriptor of individual elements rather than of the stream itself. Note that in the actual language, descriptors will carry more information including the location in the source file (or a default location if generated by any user), but for the purposes of this paper we have simplified the representation to the one described.

In the remainder of this section, we present our data transformation language through a series of examples. For reference, we present the grammar for the remaining types and the terms, patterns, and programs of the language in Appendices A.2, A.3, and A.4. In the examples, we assume the existence of a `Stream` library with standard operations `fold`, `map`, and `mapPartial`.

3.2 Example: Data transformation

In this section, we illustrate the "pay as you go" nature of error handling in PADS/T using a transformation designed to prepare Regulus data for loading into a relational database. In particular, we shred the data into two different tables based on the `info` field. The code in Figure 9 shows such a transformation (It also reorders the fields, putting the shared `service` field into the common `header`.) Our syntax for record pattern matching resembles that of SML. Note however, that unlike SML, the order of fields in PADS/T records is significant (because we have to be able to serialize the data to an output stream in a canonical form eventually). Note also that we omit the literal fields from PADS/T record patterns for conciseness. They are redundant because the information is available in the type of the expression.

This code fragment provides an example of our obliviousness principle. Despite the fact that the argument `ra` may contain errors, the transformation code did not need to examine the descriptor associated with `ra` to perform the transformation. Errors in the argument descriptor will simply be mapped into errors in the result descriptors automatically.

Two features of the language enable this obliviousness: first, the fact that programmers can deconstruct fused values without using the pattern `pat<<pdpat>>`, and second, the fact that the language propagates base type errors. If a given transform does not need to examine the descriptor, programmers can use a pattern form that matches only against the underlying value, as illustrated in Figure 9. If the transform applies an operator to a base value marked as a syntactic error (in which case the representation is `BOT`) the operator simply returns `BOT` and the resulting descriptor is set appropriately. For example, addition of one or more `BOT` values results in `BOT`. We use this property in the example in Figure 10, where we normalize timestamp-timezone pairs into simple timestamps in GMT time.

3.3 Example: Data cleaning

Instead of simply ignoring errors as in the previous section, programmers might want to clean their data, *i.e.*, filter out data containing errors. In this case, the direct access to descriptors suggested by the reification principle facilitates this process.

```
type header = {
       alarm : { alarm : Puint32 | alarm = 2
                                    orelse alarm = 3};
 ':';  start :  timestamp Popt;
 '|';  clear :  timestamp Popt;
 '|';  code: Puint32;
 '|';  src_dns  :  nsp("dns1");
 ';';  dest_dns :  nsp("dns2");
 '|';  service  : service
}

type d_alarm = {
       header   : header;
 '|';  details  : details
 }

type g_alarm = {
       header   : header;
 '|';  generic  : nsp_a Parray(';', '|')
}

fun splitAlarm (ra : raw_alarm): d_alarm opt * g_alarm opt
  = case ra of
        {alarm=a; start=s; clear=c; code=cd; src_dns=sd;
         dest_dns=dd; info=Details(d); service=s}
        =>
        (SOME { header = {alarm=a; start=s; clear=c;
                          code=cd; src_dns=sd;
                          dest_dns=dd; service=s};
                details = d},
         NONE)
      | {alarm=a; start=s; clear=c; code=cd; src_dns=sd;
         dest_dns=dd; info=Generic(g); service=s}
        =>
        (NONE,
         SOME { header = {alarm=a; start=s; clear=c;
                          code=cd; src_dns=sd;
                          dest_dns=dd; service=s};
                generic = g})
```

Figure 9: Shredding Regulus data based on the `info` field.

```
type time =
  {time: Ptimestamp_explicit_FW(8, "%H:%M:%S", GMT);
   ':'; timezone: Pstring_FW(3)}

fun normalizeTimeToGMT(t : time): Ptimestamp_FW(8) =
    case t of
      {time=t;timezone="GMT"} => t
    | {time=t;timezone="EST"} => t + (5 * 60 * 60)
    | {time=t;timezone="PST"} => t + (8 * 60 * 60)
    | ...
```

Figure 10: Normalizing timestamps

```
type streams = entry stream * entry stream
type entry_array = entry Parray(Peor,Peof)

fun splitEntry (e:entry * streams) : streams =
  case e of
    (entry<<G>> * (good * bad)) => (entry::good * bad)
  | (entry<<_>> * (good * bad)) => (good * entry::bad)

fun splitSource (s:source) : (entry_array * entry_array) =
    let (hdr ** Parray(entries,_,_)) = s in
    let (good*bad) = Stream.fold splitEntry (nil * nil) entries in
    (Parray(good,Peor,Peof) * Parray(bad,Peor,Peof))
```

Figure 11: Error filter for Sirius data

```
fun checkAlarm (a : alarm): alarm opt =
    case a of
        {{alarm=_<<S>>;...} |_} => SOME(a)
      | _ => NONE

fun collectAlarms(as : alarm stream): alarm stream =
    Stream.mapPartial (Stream.map checkAlarm as)
```

Figure 12: Querying for Erroneous Alarm Values

Figure 11 gives such an example. It examines all entries in a Sirius data source, placing good entries into one output array and bad entries into another. The good entries may then be further processed or loaded into a database without corrupting the valuable data therein. A human might examine the bad entries off-line to determine the cause of errors or to figure out how to fix the corrupted entries.

The `splitEntry` function checks its argument `e` to determine whether it is `Good` (syntactically and semantically valid). Because the function needs to access `e`'s descriptor for filtering, it uses the `pat<<pdpat>>` pattern to extract the descriptor from the fused value. In this pattern, `pdpat` is the pattern for descriptors and `pat` is a pattern for the entire fused value.

The `splitSource` function pattern matches against the input source, extracting the stream of entries, and iteratively applies the `splitEntry` function. It then packages the resulting two streams as a pair of arrays. In this example, `Parray(entry,_,_)` is a pattern for array values. An array value is a stream (in this case `entry`) coupled with a separator and a terminator (and of course, a descriptor).

3.4 Example: Error querying

Understanding the errors in an ad hoc data source can be immensely important, either to learn how to repair the data or simply to understand the errors themselves. It is not uncommon for errors in ad hoc data to be the most interesting aspect of the data. For example, errors in log files often indicate that the monitored system may be in need of repair. By reifying the error structure of the data, PADS/T enables programmers to write rich query programs to help them in such tasks. For example, such programs might walk over the data and collect information about the number, type, frequency, or location of errors.

The code in Figure 12 queries a Regulus data description to collect the alarms in that source that contain semantic errors. (Recall that semantic errors correspond to violations of user constraints). The function `checkAlarm` takes an alarm constrained type, extracts the underlying raw alarm and checks its `alarm` field for a semantic error. If the `alarm` field contained a semantic error, we return the entire alarm (wrapped in `SOME`). Otherwise, we return `NONE`. The pattern `{pat | cpat}` for constrained types matches `pat` against the underlying value, and `cpat` against a boolean indicating whether the constraint was satisfied. As we don't care about the constraint on the alarm itself, we specify a wild-card pattern. The function `collectAlarms` collects all alarm records in the alarm stream `as`, whose `alarm` field has a semantic error.

3.5 Example: Error repair

A final advantage of reifying descriptors is that it supports error repair for situation in which programmers have enough domain knowledge to make such repairs.

Figure 13 illustrates such a transformation. Here, the data analyst has discovered that some process

```
fun fixAlarmVal (al : {x:Puint32 | x=2 orelse x=3}):
  {x:Puint32 | x=2 orelse x=3} =
    case al of
      {20 | _} => {x = Puint32(2) | x = 2 orelse x = 3}
    | {30 | _} => {x = Puint32(3) | x = 2 orelse x = 3}
    | _ => al

fun fixAlarm (a : alarm): alarm opt =
    case a of
        {{alarm=al<<S>>; start=s; clear=c;
          code=cd; src_dns=sd; dest_dns=dd;
          info=i; service=s} |_}
          => {x={alarm=fixAlarmVal al;
                 start=s; clear=c;
                 code=cd; src_dns=sd; dest_dns=dd;
                 info=i; service=s}
              | checkCorr x}
      | _ => a

fun fixAlarms(as : alarm stream): alarm stream =
    Stream.map checkAlarm as
```

Figure 13: Repairing erroneous alarms

producing alarm records was erroneously using the values 20 and 30 for the alarm classes, instead of the required 2 and 3. The program finds all alarm records with semantic errors in the alarm field and corrects those with value 20 or 30, leaving the other parts of the data untouched.

We highlight two new PADS/T features illustrated in this example. In function `fixAlarmVal`, the bodies of the case branches use the constructor for constrained types (`{x=M | M'}`) and the constructor for base types (`Pbase(M)`). The base type constructor takes the underlying value and pairs it with a fresh descriptor, one that indicates no error has occurred. The constrained-type constructor evaluates the associated constraint and sets the descriptor for the value as appropriate based on the result.

3.6 A note on semantic checks

The many advantages of accurate descriptors highlight the importance of maintaining accurate meta-data. Therefore, we need language support for pairing newly created data with an accurate parse descriptor, based on the type of the pair. Furthermore, we would like to be able to check whether an existing value matches a particular data description (i.e. type), for example, in a cast operation. However, data descriptions can contain rich constraints that can't necessarily be checked at compile time.

To this end, we include a mechanism for translating the constraints found in the data descriptions into dynamic checks in the style of contracts [11]. For example, in Figure 13, the compiler will insert a dynamic check at the end of function `fixAlarmVal` to ensure that the constraint on the constrained type is satisfied. However, unlike previous work on contracts, our system does not raise an exception when a dynamic check fails, but, instead, records the failure in the parse descriptor of the datum being checked.

4 Related Work

As PADS/T supports both data description and transformation, we will divide related work between these two functions. It is interesting to note that, to the best of our knowledge, there are no other languages that synthesize these two functions as PADS/T does.

4.1 Data Description

Regular expressions and context-free grammars, while excellent formalisms for describing programming language syntax, are not ideal for describing the sort of ad hoc data we have discussed in this paper. The main reason for this is that regular expressions and context free grammars do not support polymorphism, dependency or semantic constraints — key features for describing many ad hoc data formats.

ASN.1 [9] and related systems [1] allow the user to specify the *logical* in-memory representation and automatically generate some *physical* on-disk format. Unfortunately, this doesn't help in the slightest when the user is given a fixed, physical on-disk format and needs to parse or transform that specific format. PADS/T helps solve the latter problem.

More closely related work includes ERLANG's bit syntax [2] as well as languages like PACKETTYPES [17], DATASCRIPT [4], and BLT [10]. All these systems allow users to write declarative descriptions of physical data. These projects were motivated by parsing networking protocols, TCP/IP packets, and JAVA jar-files. Like PADS/T, these languages have a type-directed approach to describing ad hoc data and permit the user to define semantic constraints. In contrast to our work, these systems do not have recursion or polymorphism, handle only binary data and assume the data is error-free. In addition, they are designed for imperative or object-oriented host languages, while we have focused here on data descriptions appropriate for a functional setting.

4.2 Data Transformation

There are a number of languages that focus on transforming XML data including XDuce [15], Cduce [5], and Xtatic [14], to name just a few. The closest work to our own is the XDuce language [15] as it considers XML-processing in the context of a statically typed, functional language with pattern matching. However, the types needed for describing XML are quite different from the types needed to describe ad hoc data. Moreover, these languages simply reject ill-formed XML. On the whole, we view PADS/T as completely complementary to this work — one can easily imagine a system in which PADS/T is used to translate ad hoc data into XML and then one of these other tools takes over.

The Harmony project [13] is also engaged in data transformation. This time for the purpose of synchronizing disparate views of the same logical data. However, Harmony operates at a higher level of abstraction than PADS/T. Once again, the relationship with Harmony appears more cooperative than competitive: One can imagine using PADS/T as a front end that translates data into a format Harmony can understand whenceforth Harmony uses its technology for synchronization.

5 Conclusions and Future Work

In this paper, we have described the importance and pervasiveness of ad hoc data, as well as the many difficulties in dealing with it. We have enumerated a set of principles for any language targeted at the problem of transforming such data. These principles include obliviousness, reification, and soundness. We term languages that support these properties "error-aware." Based on these principles, we have sketched the design of PADS/T, an ML-like language with a declarative data description sublanguage and an error-aware transformation language. We have illustrated the design of each through a number of practically-motivated examples.

The work described in this paper is very much "work in progress." Consequently, much work remains, including formally specifying the type system and semantics for the language and building a prototype implementation. We intend to leverage the existing PADS system to implement our parsers, and to define our transform language as an extension of SML so that we may "compile" it by translation into SML proper.

References

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

[2] Erlang bit syntax. `http://www.erlang.se/euc/99/binaries.ps`.

[3] Newick file format. Workshop on molecular evolution. `http://workshop.molecularevolution.org/resources/fileformats/tree_forma%ts.php`.

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

[5] V. Benzaken, G. Castagna, and A. Frisch. CDuce: An XML-centric general-purpose language. In *ACM SIGPLAN ICFP*, Aug. 2003.

[6] C. Cortes and D. Pregibon. Giga mining. In *KDD*, 1998.

[7] C. Cortes and D. Pregibon. Information mining platform: An infrastructure for KDD rapid deployment. In *KDD*, 1999.

[8] C. Cranor, Y. Gao, T. Johnson, V. Shkapenyuk, and O. Spatscheck. Gigascope: High performance network monitoring with an SQL interface. In *SIGMOD*. ACM, 2002.

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

[10] D. Eger. Bit level types. Unpublished manuscript, Mar. 2005.

[11] R. B. Findler and M. Felleisen. Contracts for higher-order functions. In *ICFP*, pages 48–59, Pittsburgh, Oct. 2002. ACM Press.

[12] K. Fisher and R. Gruber. PADS: A domain-specific language for processing ad hoc data sources. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming language design and implementation*, June 2005. To appear.

[13] J. N. Foster, M. B. Greenwald, J. T. Moore, B. C. Pierce, and A. Schmitt. Combinators for bi-directional tree transformations: A linguistic approach to the view update problem. In *POPL*, 2005.

[14] V. Gapeyev, M. Y. Levin, B. C. Pierce, and A. Schmitt. XML goes native: Run-time representations for Xtatic. In *14th International Conference on Compiler Construction*, Apr. 2005.

[15] H. Hosoya and B. C. Pierce. XDuce: A statically typed XML processing language. *ACM Transactions on Internet Technology*, 3(2):117–148, May 2003.

[16] B. Krishnamurthy and J. Rexford. *Web Protocols and Practice*. Addison Wesley, 2001.

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

A Language Syntax

A.1 Syntax of data descriptions

Constants	k	$::=$	$\texttt{true} \mid \texttt{false} \mid () \mid ...$
Type Variables	α		
Type Names	t		
Fused Types	T	$::=$	$\alpha \mid \mathit{Pbase} \mid M$
		$\mid$	$x{:}T_1 ** T_2 \mid \{\mathit{ffts}\} \mid \mathit{tas}\ t(M)$
		$\mid$	$\{x{:}T\ '\vert'\ M\}$
		$\mid$	$T\ \texttt{Parray}(M_{sep}, M_{term})$
Fused Datatypes	D	$::=$	$\texttt{datatype}\ \mathit{tps}\ t(x{:}F) = b$
		$\mid$	$\texttt{type}\ \mathit{tps}\ t(x{:}F) = T$
Type Parameters	tps	$::=$	$\cdot \mid \alpha \mid (\mathit{tvs})$
	tvs	$::=$	$\alpha \mid \alpha,\ \mathit{tvs}$
Type Arguments	tas	$::=$	$\cdot \mid T \mid (\mathit{ts})$
	ts	$::=$	$T \mid T,\ \mathit{tss}$
	b	$::=$	$\mathit{cs} \mid \texttt{case}\ M\ \texttt{of}\ \mathit{ccs}$
	cs	$::=$	$c \mid c\ \texttt{of}\ T\ '\vert'\ \mathit{cs}$
	ccs	$::=$	$\mathit{pat} \Rightarrow c$
		$\mid$	$\mathit{pat} \Rightarrow c\ \texttt{of}\ T\ '\vert'\ \mathit{ccs}$
Fused Field Types	ffts	$::=$	$\mathit{fft} \mid \mathit{fft};\ \mathit{ffts}$
Fused Field Type	fft	$::=$	$T \mid x = T$

A.2 Syntax of Transform Terms and their Types

Types	F	$::=$	T	// type of fused value
		$\mid$	base	// values of ordinary base types
		$\mid$	$\texttt{PD}$	// PD type
		$\mid$	$F * F$	// ordinary pairs
		$\mid$	$F \rightarrow F$	// functions
		$\mid$	$F\ \texttt{stream}$	// streams
Field Types	fts	$::=$	$x = F \mid x = F,\ \mathit{fts}$	

Parse Descriptors	pd	$::=$	$G \mid B \mid N \mid S \mid U$	
Fused Terms	N	$::=$	$Pbase[M_1](M_2)$	// base type constructor
		$\mid$	$\langle M \rangle$	// unit value (with singleton type M)
		$\mid$	$(x{=}M_1 ** M_2)$	// pair
		$\mid$	$\{fs\}$	// record
		$\mid$	$c[M_1](M_2)$	// data type constructor
		$\mid$	$\{x = M_1\ '\vert'\ M_2\}$	// constrained type, with M_2 the constraint
		$\mid$	$\texttt{Parray}(M, M_{sep}, M_{term})$	// array; first element is stream
Terms	M	$::=$	x	// variable
		$\mid$	N	// fused terms
		$\mid$	k	// constants
		$\mid$	pd	// pd value
		$\mid$	$(M_1 * M_2)$	// ordinary pair
		$\mid$	$\texttt{fun}\ x_1(x_2{:}F_1) : F_2 = M$	// recursive function x1 with arg x2
		$\mid$	$\texttt{nil}$	// empty stream
		$\mid$	$M_1 :: M_2$	// cons
		$\mid$	$\texttt{case}\ M\ \texttt{of}\ ms$	// deconstructors
		$\mid$	$M_1\ (M_2)$	// function application
		$\mid$	$\texttt{op}\ M$	// additional uninteresting operations
		$\mid$	$\texttt{let}\ pat = M_1\ \texttt{in}\ M_2$	// computation in host language
		$\mid$	$\texttt{cast}\ (M : T)$	// type annot/dependent cast
Fields	fs	$::=$	$x = M \mid x = M; fs$	
Matches	ms	$::=$	$pat \Rightarrow M \mid pat \Rightarrow M\ '\vert'\ ms$	

A.3 Syntax of Patterns

Fused Patterns	$fpat$	$::=$	$x \mid Pbase(pat)$	
		$\mid$	$\langle pat \rangle$	// singleton
		$\mid$	$(fpat ** fpat)$	// fused pair
		$\mid$	$\{ffps\}$	// fused record
		$\mid$	c	// const constructor
		$\mid$	$c(fpat)$	// constructor
		$\mid$	$\{fpat\ '\vert'\ cpat\}$	// type constaint
		$\mid$	$\texttt{Parray}(pat, x_{sep}, x_{term})$	// array with stream, sep and term.
		$\mid$	$fpat\langle\langle pdpat \rangle\rangle$	
Patterns	pat	$::=$	$fpat$	// fused pattern
		$\mid$	$k \mid pdpat$	// constants and parse descriptors
		$\mid$	$(pat * pat)$	// normal pair
		$\mid$	$\texttt{nil} \mid pat_1 :: pat_2$	// stream
Fused Field Pattern	$ffps$	$::=$	$x = fpat \mid x = fpat;\ ffps$	
Constraint Pattern	$cpat$	$::=$	$x \mid \texttt{true} \mid \texttt{false}$	
PD Pattern	$pdpat$	$::=$	$x \mid pd$	
Field Pattern	fps	$::=$	$x = pat \mid x = pat;\ fps$	

A.4 Syntax of Programs

Program	*prog*	::=	M	
		|	D *prog*	// type declaration
		|	`val` $x = M$ `prog`	// value declaration