PADS/ML: A Functional Data Description Language

Yitzhak Mandelbaum*, Kathleen Fisher†, David Walker*, Mary Fernandez†, Artem Gleyzer*

*Princeton University †AT&T Labs Research

yitzhakm,dpw,agleyzer@CS.Princeton.EDU kfisher,mff@research.att.com

Abstract

Massive amounts of useful data are stored and processed in *ad hoc* formats for which common tools like parsers, printers, query engines and format converters are not readily available. In this paper, we explain the design and implementation of PADS/ML, a new language and system that facilitates the generation of data processing tools for ad hoc formats. The PADS/ML design includes features such as dependent, polymorphic and recursive datatypes, which allow programmers to describe the syntax and semantics of ad hoc data in a concise, easy-to-read notation. The PADS/ML implementation compiles these descriptions into ML structures and functors that include types for parsed data, functions for parsing and printing, and auxiliary support for user-specified, format-dependent and format-independent tool generation.

Categories and Subject Descriptors D.3.2 [*Language Classifications*]: Applicative (functional) languages

General Terms Languages

Keywords Data description languages, domain-specific languages, functional programming, dependent types, ML, modules, parsing, printing

1. Introduction

An *ad hoc* data format is any semi-structured data format for which parsing, querying, analysis, or transformation tools are not readily available. Despite the existence of standard formats like XML, ad hoc data sources are ubiquitous, arising in industries as diverse as finance, health care, transportation, and telecommunications as well as in scientific domains, such as computational biology and physics. Figure 1 summarizes a variety of such formats, including ASCII, binary, and Cobol encodings, with both fixed and variable-width records arranged in linear sequences and in tree-shaped hierarchies. Snippets of some of these data formats appear in Figure 2. Note that even a single format can exhibit a great deal of syntactic variability. For example, Figure 2(c) contains two records from a network-monitoring application. Each record has a different number of fields (delimited by '|') and individual fields contain structured values (*e.g.*, attribute-value pairs separated by '=' and delimited by ';').

Name: Use	Representation
Gene Ontology (GO): Gene Product Information	Variable-width ASCII records
SDSS/Reglens Data: Weak gravitational lensing analysis	Floating point numbers, among others
Web server logs (CLF): Measuring web workloads	Fixed-column ASCII records
AT&T Call detail data: Phone call fraud detection	Fixed-width binary records
AT&T billing data: Monitoring billing process	Cobol
Newick: Immune system response simulation	Fixed-width ASCII records in tree-shaped hierarchy
OPRA: Options-market transactions	Mixed binary & ASCII records with data-dependent unions
Palm PDA: Device synchronization	Mixed binary & character with data-dependent constraints

Figure 1. Selected ad hoc data sources.

Common characteristics of ad hoc data make it difficult to perform even basic data-processing tasks. To start, data analysts typically have little control over the format of the data; it arrives "as is," and the analysts can only thank the supplier, not request a more convenient format. The documentation accompanying ad hoc data is often incomplete, inaccurate, or missing entirely, which makes understanding the data format more difficult. Managing the errors that frequently occur poses another challenge. Common errors include undocumented fields, corrupted or missing data, and multiple representations for missing values. Sources of errors include malfunctioning equipment, race conditions on log entry, the presence of non-standard values to indicate "no data available," and human error when entering data. How to respond to errors is highly application-specific: Some need to halt processing and alert a human operator; others can repair errors by consulting auxiliary sources; still others simply filter out erroneous values. In some cases, erroneous data is more important than error-free data; for example, it may signal where two systems are failing to communicate. Unfortunately, writing code that reliably handles both error-free and erroneous data is difficult and tedious.

1.1 PADS/ML

PADS/ML is a domain-specific language designed to improve the productivity of data analysts, be they computational biologists, physicists, network administrators, healthcare providers or financial analysts. To use the system, analysts describe their data in the PADS/ML language, capturing both the physical format of the data and any expected semantic constraints. In return for this investment, analysts reap substantial rewards. First of all, the description serves as clear, compact, and formally-specified documentation of the data's structure and properties. In addition, the PADS/ML compiler can convert the description into a suite of robust, end-to-end data processing tools and libraries specialized to the format. As the

```
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
```

(a) Simplified Regulus network-monitoring data.

```
0|1005022800
9153|9153|1|0|0|0|0||152268|LOC_6|0|FRDW1|DUO|LOC_CRTE|1001476800|LOC_OS_10|1001649601
9152|9151|1|9735551212|0||9085551212|07988|no_ii152272|EDTF_6|0|APRL1|DUO|10|1000295291
```

(b) Sirius data used to monitor billing in telecommunications industry.

```
(((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)
```

(c) Newick data used to study immune system responses.

Figure 2. Snippets of a variety of ad hoc data formats. Each '\' denotes a newline we inserted to improve readability.

analysts' data sources evolve over time, they can simply update the high-level descriptions and recompile to produce updated tools.

The type structure of modern functional programming languages inspired the design of the PADS/ML language. Specifically, PADS/ML provides dependent, polymorphic recursive datatypes, layered on top of a rich collection of base types, to specify the syntactic structure and semantic properties of data formats. Together, these features enable analysts to write concise, complete, and reusable descriptions of their data. We describe the PADS/ML language using examples from several domains in Section 2.

We have implemented PADS/ML by compiling descriptions into O'CAML code. We use a "types as modules" implementation strategy in which each PADS/ML type becomes a module and each PADS/ML type constructor becomes a functor. We chose ML as the host language because we believe that functional languages lend themselves to data processing tasks more readily than imperative languages such as C or JAVA. In particular, constructs such as pattern matching and higher-order functions make expressing data transformations particularly convenient. Section 3 describes our "types as modules" strategy and shows how PADS/ML-generated modules together with functional O'CAML code can concisely express common data-processing tasks such as filtering errors and format transformation.

A key benefit of our approach is the high return-on-investment that analysts can derive from describing their data in PADS/ML. In particular, PADS/ML makes it possible to produce automatically a collection of data analysis and processing tools from each description. As a start, the PADS/ML compiler generates from each description a parser and a printer for the associated data source. The parser maps raw data into two data structures: a canonical *representation* of the parsed data and a *parse descriptor*, a meta-data object detailing properties of the corresponding data representation. Parse descriptors provide applications with programmatic access to errors detected during parsing. The printer inverts the process, mapping internal data structures and their corresponding parse descriptors back into raw data.

In addition to generating parsers and printers, our framework permits developers to add *format-independent* tools without modifying the PADS/ML compiler by specifying *tool generators*. Such generators need only match a generic interface, specified as an ML signature. Correspondingly, for each PADS/ML description, the PADS/ML compiler generates a meta-tool (a functor) that takes a tool generator and specializes it for use with the particular description. Section 4 describes the tool framework and gives examples of three format-independent tools that we have implemented: a data printer useful for description debugging, an accumulator that keeps track of error information for each type in a data source, and a formatter that maps data into XML.

To define the semantics of PADS/ML, we extended our earlier work on the Data Description Calculus (DDC) [3] to account for PADS/ML's polymorphic types. In the process, we simplified the original presentation of the parser semantics substantially, particularly for recursive types. In addition, we extended the theory to give a printing semantics. We used this new semantics to guide the PADS/ML implementation of printing. We also proved a *canonical forms* theorem, which states that the generated parsers produce well-typed, well-behaved canonical results, and, conversely, that printers operate correctly on the appropriate canonical inputs. A full treatment of the extended calculus appears in Mandelbaum's Ph.D. thesis [4], while an overview of the calculus and printing semantics, as well as the associated metatheory, can be found in our companion technical report [5].

PADS/ML has evolved from previous work on PADS/C [1] [2], but PADS/ML differs from PADS/C in three significant ways. First, it is targeted at the ML family of languages. Using ML as the host language simplifies many data processing tasks, such as filtering and normalization, which benefit from ML's pattern matching constructs and high level of abstraction. Second, unlike PADS/C types, PADS/ML types may be parameterized by other types, resulting in more concise and elegant descriptions through code reuse. ML-style datatypes and anonymous nested tuples also help improve readability by making descriptions more compact. Third, PADS/ML provides significantly better support for the development of new tool generators. In particular, PADS/ML provides a generic interface against which tool generators can be written. In PADS/C, the compiler itself generates all tools, and, therefore, developing a new tool generator requires understanding and modifying the compiler. Mandelbaum's Ph.D. thesis [4] contains a full discussion of related work.

In summary, this research makes the following contributions:

- We have designed and implemented PADS/ML, a novel data-description language that includes dependent, polymorphic, recursive datatypes. This design allows data analysts to express the syntactic structure and semantic properties of data formats from numerous application domains in a concise, elegant, and easy-to-read notation.
- Our PADS/ML implementation employs an effective and general "types as modules" compilation strategy that produces robust parser and printer functions as well as auxiliary support for user-specified tool generation. Our implementation is available at `http://www.padsproj.org/padsml/`.

[1] We henceforth call the original PADS language PADS/C.

2. Describing Data in PADS/ML

A PADS/ML 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 ASCII-encoded, 8-bit unsigned integers (`Puint8`) and 32-bit signed integers (`Pint32`), binary 32-bit integers (`Pbint32`), dates (`Pdate`), strings (`Pstring`), zip codes (`Pzip`), phone numbers (`Pphone`), 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 may be parameterized by ML values. This mechanism reduces the number of built-in base types and permits base types to depend on values in the parsed data. For example, the base type `Puint16_FW(3)` specifies an unsigned two byte integer physically represented by exactly three characters, and the base type `Pstring` takes an argument indicating the *terminator character*, *i.e.*, the character in the source that follows the string.

To describe more complex data, PADS/ML provides a collection of type constructors derived from the type structure of functional programming languages like Haskell and ML. We explain these structured types in the following subsections using examples drawn from data sources we have encountered in practice.

2.1 Simple Structured Types

The bread and butter of a PADS/ML description are the simple structured types: tuples and records for specifying ordered data, lists for specifying homogeneous sequences of data, sum types for specifying alternatives, and singletons for specifying the occurrence of literal characters in the data. We describe each of these constructs as applied to the Sirius data presented in Figure 2(b).

Sirius data summarizes orders for phone service placed with AT&T. Each Sirius data file starts with a timestamp followed by one record per phone service order. Each order consists of a header and 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 that the number was generated. The event sequence represents the various states a service order goes through; it is represented as a newline-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. Clearly English is a poor language for describing data formats!

Figure 3 contains the PADS/ML description for the Sirius data format. The description is a sequence of type definitions. Type definitions precede uses, therefore the description should be read bottom up. The type `Source` describes a complete Sirius data file and denotes an ordered tuple containing a `Summary_header` value followed by an `Orders` value.

The type `Orders` uses the list type constructor `Plist` to describe a homogenous sequence of values in a data source. The `Plist` constructor takes three parameters: on the left, the type of elements in the list; on the right, a literal *separator* that separates elements in the list and a literal *terminator* that marks the end of the list. In this example, the type `Orders` is a list of `Order` elements, separated by a newline, and terminated by `peof`, a special literal that describes the *end-of-file marker*. Similarly, the `Events` type

```
ptype Summary_header = "0|" * Ptimestamp * '\n'

pdatatype Dib_ramp =
  Ramp of Pint
| GenRamp of "no_ii" * Pint

ptype Order_header = {
     order_num : Pint;
'|'; att_order_num : [i:Pint | i < order_num];
'|'; ord_version : Pint;
'|'; service_tn : Pphone Popt;
'|'; billing_tn : Pphone Popt;
'|'; nlp_service_tn : Pphone Popt;
'|'; nlp_billing_tn : Pphone Popt;
'|'; zip_code : Pzip Popt;
'|'; ramp : Dib_ramp;
'|'; order_sort : Pstring('|');
'|'; order_details : Pint;
'|'; unused : Pstring('|');
'|'; stream : Pstring('|');
'|'
}

ptype Event  = Pstring('|') * '|' * Ptimestamp
ptype Events = Event Plist('|', '\n')

ptype Order  = Order_header * Events
ptype Orders = Order Plist('\n', peof)

ptype Source = Summary_header * Orders
```

Figure 3. PADS/ML description for Sirius provisioning data.

denotes a sequence of `Event` values separated by vertical bars and terminated by a newline.

String, character, and integer literals can be embedded in a description and are interpreted as singleton types. For example, the `Event` type is a string terminated by a vertical bar, followed by a vertical bar, followed by a timestamp. The singleton type `'|'` means that the data source must contain the character `'|'` at this point in the input stream. Correspondingly, the generated parser reads `'|'` and the generated printer writes `'|'`. These literals do not appear in the generated data representations.

The type `Order_header` is a record type, *i.e.*, a tuple type in which each field may have an associated name. The named field `att_order_num` illustrates two other features of PADS/ML: dependencies and constraints. Here, `att_order_num` depends on the previous field `order_num` and is constrained to be less than that value. In practice, constraints may be complex, have multiple dependencies, and can specify, for example, the sorted order of records in a sequence. Constrained types have the form `[x:T | e]` where `e` is an arbitrary pure boolean expression. Data satisfies this description if it satisfies `T` and `e` evaluates to `true` when the parsed representation of the data is substituted for `x`. If the boolean expression evaluates to false, the data contains a *semantic* error.

The datatype `Dib_ramp` specifies two alternatives for a data fragment: either one integer or the fixed string `"no_ii"` followed by one integer. The order of alternatives is significant, that is, the parser attempts to parse the first alternative and only if it fails, it attempts to parse the second alternative. This semantics differs from similar constructs in regular expressions and context-free grammars, which non-deterministically choose between alternatives.

2.2 Recursive Types

PADS/ML can describe data sources with recursive structure. An example of such data is the Newick Standard format, a flat representation of trees used by biologists [6]. Example Newick Standard data provided by Steven Kleinstein appears in Figure 2(c). The format uses properly nested parentheses to specify a tree hierarchy. A leaf node is a string label followed by a colon and a number. An interior node contains a sequence of children nodes, delimited by parentheses, followed by a colon and a number. The numbers represent the "distance" that separates a child node from its parent. In this example, the string labels are gene names and the distances denotes the number of mutations that occur in the antibody receptor genes of B lymphocytes. The following PADS/ML code describes this format:

```
ptype Entry = {name: Pstring(':'); ':'; dist: Pfloat32}

pdatatype Tree =
  Interior of '(' * Tree Plist(';',')')  * ')'
| Leaf of Entry
```

2.3 Polymorphic Types and Advanced Datatypes

Polymorphic types enable more concise descriptions and allow programmers to define convenient libraries of reusable descriptions. The description in Figure 4 illustrates types parameterized by both types and values. It specifies the format of alarm data recorded by a network-link monitor used in the Regulus project at AT&T. Figure 2(a) contains corresponding example data. We describe the format in tandem with describing its PADS/ML description.

This data format has several variants of name-value pairs. The PADS/C description of this format [5] must define a different type for each variant. In contrast, the polymorphic types of PADS/ML allow us to define the type `Pnvp`, which takes both type and value parameters to encode all the variants. As is customary in ML, type parameters appear to the left of the type name, while value parameters and their ML types appear to the right. The type `Pnvp` has one type parameter named `Alpha` and one value parameter named `p`. Informally, `Alpha Pnvp(p)` is a name-value pair where the value is described by `Alpha` and the name must satisfy the predicate `p`.

The `Nvp` type reuses the `Pnvp` type to define a name-value pair whose name must match the argument string `name` but whose value can have any type. The `Nvp_a` type also uses the type `Pnvp`. It defines a name-value pair that permits any name, but requires the value to have type `SVString` (a string terminated by a semicolon or vertical bar). Later in the description, the type parameter to `Nvp` is instantiated with IP addresses, timestamps, and integers.

The Regulus description also illustrates the use of *switched* datatypes. A switched datatype selects a variant based on the value of a user-specified O'CAML expression, which typically references parsed data from earlier in the data source. For example, the switched datatype `Info` chooses a variant based on the value of its `alarm_code` parameter. 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.

```
(* Pstring terminated by ';' or '|'. *)
ptype SVString = Pstring_SE("/;|\\|/")

(* Generic name value pair. Accepts predicate
   to validate name as argument. *)
ptype (Alpha) Pnvp(p : string -> bool) =
      { name : [name : Pstring('=') | p name];
            '=';
        value : Alpha }

(* Name value pair with name specified. *)
ptype (Alpha) Nvp(name:string) =
   Alpha Pnvp(fun s -> s = name)

(* Name value pair with any name. *)
ptype Nvp_a = SVString Pnvp(fun _ -> true)

ptype Details = {
      source      : Pip Nvp("src_addr");
';';  dest        : Pip Nvp("dest_addr");
';';  start_time  : Ptimestamp Nvp("start_time");
';';  end_time    : Ptimestamp Nvp("end_time");
';';  cycle_time  : Puint32 Nvp("cycle_time")
}

pdatatype Info(alarm_code : int) =
  match alarm_code with
    5074 -> Details of Details
  | _    -> Generic of Nvp_a Plist(';','|')

pdatatype Service =
    DOMESTIC      of "DOMESTIC"
  | INTERNATIONAL of "INTERNATIONAL"
  | SPECIAL       of "SPECIAL"

ptype Alarm = {
       alarm    : [i : Puint32 | i = 2 or i = 3];
 ':';  start    : Ptimestamp Popt;
 '|';  clear    : Ptimestamp Popt;
 '|';  code     : Puint32;
 '|';  src_dns  : SVString Nvp("dns1");
 ';';  dest_dns : SVString Nvp("dns2");
 '|';  info     : Info(code);
 '|';  service  : Service
}

ptype Source = Alarm Plist('\n',peof)
```

Figure 4. Description of Regulus data.

3. From PADS/ML to O'CAML

The PADS/ML compiler takes descriptions and generates O'CAML modules that can be used by any O'CAML program. In this section, we describe the generated modules and illustrate their use.

3.1 Types as Modules

We use the O'CAML module system to structure the libraries generated by the PADS/ML compiler. Each PADS/ML base type is implemented as an O'CAML module. For each PADS/ML type in a description, the PADS/ML compiler generates an O'CAML module containing the types, functions, and nested modules that implement the PADS/ML type. All the generated modules are grouped into one module that implements the complete description. For example, a PADS/ML description named `sirius.pml`, which contains three named types, will result in the O'CAML file `sirius.ml` defining the module `Sirius`, which will contain three submodules, each corresponding to one named type.

Namespace management alone is sufficient motivation to employ a "types as modules" approach, but the power of the ML module system provides substantially more. We implement polymorphic PADS/ML types as functors from (type) modules to (type) modules. Ideally, we would like to map recursive PADS/ML types into recursive modules. Unfortunately, this approach currently is not possible, because O'CAML prohibits the use of functors within recursive modules, and the output of the PADS/ML compiler includes a functor for each type. Instead, we implement recursive types as modules containing recursive datatypes and functions. As there is no theoretical reason to prevent recursive modules from containing functors [1], we pose our system as a challenge to implementers of module systems.

The module generated for any monomorphic PADS/ML type matches the signature `S`:

```
module type S = sig
  type rep
  type pd_body
  type pd = Pads.pd_header * pd_body
  val  parse : Pads.handle -> rep * pd
  val  print : rep -> pd -> Pads.handle -> unit
  (* Functor for tool generator ... *)
  module Traverse ...
end
```

The *representation* (rep) type describes the in-memory representation of parsed data, while the *parse-descriptor* (pd) type describes meta-data collected during parsing. The parsing function converts the raw data into an in-memory representation and parse descriptor for the representation. The printing function performs the reverse operation. The module also contains a generic tool generator implemented as a functor; we defer a description of this functor to Section 4. The module Pads contains the built-in types and functions that occur in base-type and generated modules. The type Pads.pd_header is the type of all parse-descriptor headers and Pads.handle is an abstract type containing the private data structures PADS/ML uses to manage data sources.

The structure of the representation and parse-descriptor types resembles the structure of the corresponding PADS/ML type, making it easy to see the correspondence between parsed data, its internal representation, and the corresponding meta-data. For example, given the PADS/ML type Pair describing a character and integer separated by a vertical bar:

```
  ptype Pair = Pchar * '|' * Pint
```

the compiler generates a module with the signature:

```
module type Pair_sig = sig
  type rep     = Pchar.rep * Pint.rep
  type pd_body = Pchar.pd  * Pint.pd
  type pd      = Pads.pd_header * pd_body
  val  parse   : Pads.handle -> rep * pd
  val  print   : rep -> pd -> Pads.handle -> unit
  ...
end
```

The parse-descriptor header reports on the parsing process that produced the corresponding representation. It includes the location of the data in the source, an error code describing the first error encountered, and the number of subcomponents with errors. The body contains the parse descriptors for subcomponents. Parse descriptors for base types have a body of type unit.

The signature for a polymorphic PADS/ML type uses the signature S for monomorphic types, defined above. Given the polymorphic PADS/ML type ABPair:

```
ptype (Alpha,Beta) ABPair = Alpha * '|' * Beta
```

the compiler generates a module with the signature:

```
module type ABPair_sig (Alpha : S) (Beta : S) =
sig
  type rep      = Alpha.rep * Beta.rep
  type pd_body = Alpha.pd * Beta.pd
  type pd       = Pads.pd_header * pd_body
  val  parse    : Pads.handle -> rep * pd
  val  print    : rep -> pd -> Pads.handle -> unit
  ...
end
```

3.2 Using the Generated Libraries

Common data management tasks like filtering and normalization are easy to express in O'CAML. In the remainder of this section, we illustrate this point by giving O'CAML programs to compute properties of ad hoc data, to filter it, and to transform it.

3.2.1 Example: Computing Properties

Given the PADS/ML type:

```
ptype IntTriple = Pint * '|' * Pint * '|' * Pint
```

the following O'CAML expression computes the average of the three integers in the file input.data:

```
let ((i1,i2,i3), (pd_hdr, pd_body)) =
 Pads.parse_source IntTriple.parse "input.data" in
match pd_hdr with
  {error_code = Pads.Good} -> (i1 + i2 + i3)/3
| _ -> raise Pads.Bad_file
```

The parse_source function takes a parsing function and a file name, applies the parsing function to the data in the specified file, and returns the resulting representation and parse descriptor. To ensure the data is valid, the program examines the error code in the parse-descriptor header. The error code Good indicates that the data is syntactically and semantically valid. Other error codes include Nest, indicating an error in a subcomponent, Syn, indicating that a syntactic error occurred during parsing, and Sem, indicating that the data violates a semantic constraint. The expression above raises an exception if it encounters any of these error codes.

Checking the top-level parse descriptor for errors is sufficient to guarantee that there are no errors in any of the subcomponents. This property holds for all representations and corresponding parse descriptors. This design supports a "pay-as-you-go" approach to error handling. The parse descriptor for valid data need only be consulted once, no matter the size of the corresponding data. User code only needs to traverse nested parse descriptors if information about an error is required.

3.2.2 Example: Filtering

Data analysts often need to "clean" their data (*i.e.*, remove or repair data containing errors) before loading the data into a database or other application. O'CAML's pattern matching and higher-order functions can simplify these tasks. For example, the expression in Figure 5 partitions Sirius data into valid orders and invalid orders.

```
open Pads

let classify_order order (pd_hdr, pd_body) (good, bad)=
   match pd_hdr with
    {error_code = Good} -> (order::good, bad)
   | _                  -> (good, order::bad)

let split_orders orders (orders_pd_hdr,order_pds) =
   List.fold_right2 classify_order orders order_pds ([],[])

let ((header, orders),(header_pd, orders_pd)) =
   parse_source Sirius.parse "input.txt"

let (good,bad) = split_orders orders orders_pd
```

Figure 5. Error filtering of Sirius data

3.2.3 Example: Transformation

Once a data source has been parsed and cleaned, a common task is to transform the data into formats required by other tools, like a relational database or a statistical analysis package. 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, one common transformation is to convert data with variants (*i.e.*, datatypes) into a form that such systems can handle. One option is to partition or "shred" the data into several relational tables, one for each variant. A second option is to create an universal table, with one column for each field in any variant. If a given field does not occur in a particular variant, its value is marked as missing.

Figure 6 shows a partial listing of RegulusNormal.pml, a normalized version of the Regulus description from Section 2.

```
...
ptype Header = {
       alarm : [ a : Puint32 | a = 2 or a = 3];
 ':';  start :  Ptimestamp Popt;
 '|';  clear :  Ptimestamp Popt;
 '|';  code  :  Puint32;
 '|';  src_dns  :  Nvp("dns1");
 ';';  dest_dns :  Nvp("dns2");
 '|';  service  : Service
}

ptype D_alarm = {
       header : Header;
 '|';  info   : Details
 }

ptype G_alarm = {
       header : Header;
 '|';  info   : Nvp_a Plist(';','|')
}
```

Figure 6. Listing of `RegulusNormal.pml`, a normalized format for Regulus data. All named types not explicitly included in this figure are unchanged from the original Regulus description.

```
open Regulus
open RegulusNormal
module A = Alarm
module DA = D_alarm
module GA = G_alarm
module Header = H

type ('a,'b) Sum = Left of 'a | Right of 'b

let split_alarm ra =
  let h =
    {H.alarm=ra.A.alarm; H.start=ra.A.start;
     H.clear=ra.A.clear; H.code=ra.A.code;
     H.src_dns=ra.A.src_dns; H.dest_dns=ra.A.dest_dns;
     H.service=ra.A.service}
  in match ra with
      {info=Details(d)} ->
      Left {DA.header = h; DA.info = d}
    | {info=Generic(g)} ->
      Right {GA.header = h; GA.info = g}

let split_alarm_pd pd = ... (* mirrors split_alarm *)

let process_alarm pads [pads_D; pads_G] =
  let a,a_pd = Alarm.parse pads in
    match (split_alarm a, split_alarm_pd a_pd) with
     (Left  da, Left  da_p) -> DA.print da da_p pads_D
    |(Right ga, Right ga_p) -> GA.print ga ga_p pads_G
    | _ -> ... (* Bug! *)

let _ = process_source process_alarm
              "input.data" ["d_out.data";"g_out.data"]
```

Figure 7. Shredding Regulus data based on the `info` field.

In this shredded version, `Alarm` has been split into two top-level types `D_alarm` and `G_alarm`. The type `D_alarm` contains all the information concerning alarms with the detailed payload, while `G_alarm` contains the information for generic payloads. In the original description, the `info` field identified the type of its payload. In the shredded version, the two different types of records appear in two different data files. Since neither of these formats contains a union, they can be easily loaded into a relational database.

The code fragment in Figure 7 shreds Regulus data in the format described by `Regulus.pml` into the formats described in `RegulusNormal.pml`. It uses the `info` field of `Alarm` records to partition the data. Notice that the code invokes the `print` functions generated for the `G_alarm` and `D_alarm` types to output the shredded data.

4. The Generic Tool Framework

An essential benefit of PADS/ML is that it can provide users with a high return-on-investment for describing their data. While the generated parser and printer alone are enough to justify the user's effort, we aim to increase the return by enabling users to easily construct data analysis tools. To this end, we provide a simple framework for others to develop format-independent tools.

The techniques of type-directed programming, known variously as *generic* or *polytypic* programming, provide a convenient conceptual starting point in designing a tool framework. In essence, any format-independent tool is a function from a description to a concrete realization of that tool. As PADS/ML descriptions are (dependent) types, a format-independent tool is a type-directed function.

Some modern functional programming languages, Generic Haskell [**?**], in particular, have many features that support type-directed programming, and hence would support development of format-independent tools quite nicely. O'CAML, however, lacks any specific, built-in generic programming facility. Fortunately, we can still achieve many of the benefits of generic programming idioms by having the PADS/ML compiler generate well-designed, format-specific libraries at compile time and then linking those libraries to format-independent routines.

To be specific, for each format description, PADS/ML generates a format-dependent traversal mechanism that implements a generalized fold over the representations and parse descriptors that correspond to the description. Independently, tool developers write format-independent routines that specify the behaviour of a tool over each PADS/ML type constructor. When users need a *specific tool* for a *specific format*, they link the format-dependent traversal to the format-independent routines via functor application.

In principle, different tools might require different sorts of traversals. However, many of the tools we have encountered in practice so far, both in implementing PADS/ML and PADS, perform their computations in a single pass over the representation and corresponding parse descriptor, visiting each value in the data with a left-to-right, pre-, post-, or in-order traversal. This paradigm arises naturally as it scales to very large data sets. Hence, the PADS/ML compiler generates an implementation of a such traversal for each data description.

4.1 The Generic-Tool Interface

The interface between format-specific traversals and generic tools is specified as an O'CAML signature. For every type constructor in PADS/ML, the signature describes a sub-module that implements the generic tool for that type constructor. In addition, it specifies an (abstract) type for auxiliary state that is threaded through the traversal. Figure 8 contains an excerpt of the signature that includes the signatures of the `Record` and `Datatype` modules. The signatures of other modules are quite similar.

The `Record` module includes a type `partial_state` that allows tools to represent intermediate state in a different form than the general state. The `init` function forms the state of the record from the state of its fields. The `start` function receives the PD header for the data element being traversed and begins processing the element. Function `project` takes a record's state and the name of a field and returns that field's state. Function `process_field` updates the intermediate state of the record based on the name and state of a field, and `finish` converts the finished intermediate state into general tool state. Note that any of these functions could have side effects.

Although the `Datatype` module is similar to the `Record` module, there are some important differences. The `Datatype` `init` function does not start with the state of all the variants. Instead, a variant's state is added during processing so that only variants that have been encountered will have corresponding state.

```
module type S = sig
 type state
 ...
 module Record : sig
   type partial_state
   val  init          : (string * state) list -> state
   val  start         : state -> Pads.pd_header
                        -> partial_state
   val  project       : state -> string -> state
   val  process_field : partial_state -> string
                        -> state -> partial_state
   val  finish        : partial_state -> state
 end

 module Datatype : sig
   type partial_state
   val  init            : unit -> state
   val  start           : state -> Pads.pd_header
                          -> partial_state
   val  project         : state -> string -> state option
   val  process_variant : partial_state -> string
                          -> state -> partial_state
   val  finish          : partial_state -> state
 end
  ...
end
```

Figure 8. Excerpt of generic-tool interface `Generic_tool.S`.

For this reason, `project` returns a `state option`, rather than a `state`. This design is essential for supporting recursive datatypes as trying to initialize the state for all possible variants of the datatype would cause the `init` function to loop infinitely.

The following code snippet gives the signature of the traversal functor as it would appear in the signature `S` from Section 3.

```
module Traverse (Tool : Generic_tool.S) :
sig
  val init : unit -> Tool.state
  val traverse : rep -> pd -> Tool.state -> Tool.state
end
```

The functor takes a generic tool generator and produces a format-specific tool with two functions: `init`, to create the initial state for the tool, and `traverse`, which traverses the representation and parse descriptor for the type and updates the given tool state.

4.2 Example Tools

We have used this framework to implement a variety of tools useful for processing ad hoc data, including an XML formatter, an accumulator tool for generating statistical overviews of the data, and a data printer for debugging. We briefly describe these tools to illustrate the flexibility of the framework.

The XML formatter converts any data with a PADS/ML description into a canonical XML format. This conversion is useful because it allows analysts to exploit the many useful tools that exist for manipulating data in XML.

The accumulator tool provides a statistical summary of data. Such summaries are useful for developing a quick understanding of data quality. In particular, after receiving a new batch of data, analysts might want to know the frequency of errors, or which fields are the most corrupted. The accumulator tool tracks the distribution of the top n distinct legal values and the percentage of errors. It operates over data sources whose basic structure is a series of records of the same type, providing a summary based on viewing many records in the data source. More complex accumulator programs and a number of other statistical algorithms can easily be implemented using the tool generation infrastructure.

Finally, as an aid in debugging PADS/ML descriptions, we have implemented a simple printing tool. In contrast to the printer generated by the PADS/ML compiler, the output of this tool corresponds to the in-memory representation of the data rather than its original format, which may have delimiters or literals that are not present in the representation. This format is often more readable than the raw data.

5. Conclusions

PADS/ML is a high-level, domain-specific language and system designed to help improve the productivity of the legions of data analysts who work with ad hoc data on a regular basis. Inspired by the type structure of functional programming languages, PADS/ML uses dependent, polymorphic and recursive data types to describe the syntax and the semantic properties of ad hoc data sources. The language is compact and expressive, capable of describing data from diverse domains including networking, computational biology, finance, and physics. The PADS/ML compiler uses a "types as modules" compilation strategy in which every PADS/ML type definition is compiled into an O'CAML module containing types for data representations and functions for data processing. Functional programmers can use the generated modules to write clear and concise *format-dependent* data processing programs. Furthermore, our system design allows external tool developers to write new *format-independent* tools simply by supplying a module that matches the appropriate generic signature. The latest release of our implementation is available at `http://www.padsproj.org/padsml/`.

The next step in our long-term agenda is to build a new generation of format-independent data analysis tools. While our current tools perform some simple syntactic analysis and transformation, we intend our next generation toolkit to perform deeper semantic analysis and more sophisticated transformations. For example, we may explore specification-driven, content-based search, clustering, test data generation, machine learning, security, and data visualization. We believe that if we can automatically generate stand-alone, end-to-end tools that perform these functions over arbitrary data, we can have a substantial impact on the productivity of researchers in a broad array of scientific fields ranging from computational biology through computer science to cosmology and beyond.

Acknowledgments. We would like to thank Derek Dreyer for discussions and advice on advanced module systems. Some of this material is based upon work supported by NSF awards 0615062 and 0612147. Any opinions, findings, and conclusions or recommendations expressed in the material are those of the authors and do not necessarily reflect the views of the NSF.

References

[1] D. Dreyer. *Understanding and Evolving the ML Module System*. PhD thesis, CMU, May 2005.

[2] 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. ACM Press, June 2005.

[3] K. Fisher, Y. Mandelbaum, and D. Walker. The next 700 data description languages. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 2 – 15, Jan. 2006.

[4] Y. Mandelbaum. *The Theory and Practice of Data Description*. PhD thesis, Princeton University, September 2006.

[5] Y. Mandelbaum, K. Fisher, D. Walker, M. Fernandez, and A. Gleyzer. PADS/ML: A functional data description language. Technical Report TR-761-06, Princeton University, July 2006.

[6] Tree formats. Workshop on molecular evolution. `http://workshop.molecularevolution.org/resources/fileformats/tree_formats.php`.