A Generic Programming Toolkit for PADS/ML: First-Class Upgrades for Third-Party Developers

Mary Fernández[1], Kathleen Fisher[1], J. Nathan Foster[2], Michael Greenberg[1,2], and Yitzhak Mandelbaum[1]

[1] AT&T Research
[2] University of Pennsylvania

Abstract. Domain-specific languages facilitate solving problems in a targeted domain by providing features particular to the domain. Declarative domain-specific languages have the additional benefit that users specify *what something means* rather than *how to do something*. As a result, the language compiler is free to choose the best implementation strategies and to generate multiple artifacts from a single description. PADS/ML is a declarative data description language designed to facilitate ad hoc data management. From a single description, the compiler generates a myriad of artifacts, including data structures for the in-memory representation of the data and parsers and printers. In this paper, we describe a new generic programming infrastructure for PADS/ML that allows third-party developers to define additional useful artifacts without modifying the compiler. We report on two case studies that use this infrastructure. In the first, we build a version of PADX for PADS/ML, allowing any data source with a PADS/ML description to be queried as if it were XML. In the second, we extend Harmony with the ability to synchronize any data with a PADS/ML description.

1 Introduction

Domain-specific languages provide enormous leverage precisely because they have limited scope, allowing their designers to tailor the abstractions they provide to the targeted domain. Declarative domain-specific languages bring an additional benefit in that they specify *what something means*, rather than *how to do something*. As a result, the compiler is free to decide how to accomplish the task and even what tasks need to be accomplished. This freedom allows the designers of declarative domain-specific languages to generate more than one software artifact from a single specification.

The PADS/ML language is a declarative domain-specific language for specifying the format of ad hoc data [MFW+07]. An *ad hoc* data format is any semi-structured data representation 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 and binary encodings, with both fixed and variable-width records arranged in linear sequences and in tree-shaped hierarchies.

Name	Use	Representation
Gene Ontology (GO) [Con]	Gene Product Information	Variable-width ASCII records
SDSS/Reglens Data [MHS+05]	Weak gravitational lensing analysis	Floating point numbers, et al
Web server logs (CLF)	Measuring web workloads	Fixed-column ASCII records
AT&T Call detail data	Phone call fraud detection	Fixed-width binary records
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

Fig. 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 have little control over the format of the data; it typically 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. Ad hoc data sources frequently contain errors, which poses another challenge. For some applications, like system monitors, erroneous data is more important than error-free data; it may signal, for example, where two systems are failing to communicate. Unfortunately, writing code that reliably handles both error-free and erroneous data is difficult and tedious.

The PADS/ML system, like its close ancestor PADS/C [FG05], solves these problems by providing a declarative data description language. A PADS/ML specification describes the physical layout and semantic properties of an ad hoc data source. The language provides a type-based model: basic types specify atomic data such as integers, strings, dates, *etc.*, while structured types such as tuples, records, and datatypes describe compound data. Leveraging the declarative nature of such descriptions, the PADS/ML compiler generates from each description a suite of useful data structures and tools, including a canonical in-memory representation of the data, a canonical meta-data representation called a *parse descriptor*, a parser, and a printer.

Ideally, a system like PADS/ML would permit third-party developers to build new tools for specifications without modifying the compiler. With that goal in mind, the original PADS/ML compiler generated an OCAML functor for traversing the canonical data structure. Although an improvement over PADS/C, which requires modifying the compiler to generate new tools, the PADS/ML infrastructure was insufficient because it only supported tools that *consume* a PADS data representation in a single depth-first, left-to-right traversal. This limitation precludes many useful tools, *e.g.,* those that require a different traversal strategy or that *produce* a PADS/ML data representation rather than consuming one.

To rectify this deficiency, we redesigned the generic tool infrastructure of PADS/ML, leveraging ideas from type-directed programming [Yan98,Hin04]. Given a PADS/ML description, third-party developers can now build a wide variety of generic tools re-

lating to the description's in-memory representation and parse descriptor. To illustrate the power of this generic infrastructure, we describe two third-party tools that did not require making tool-specific changes to the compiler: an implementation of PADX [FFGM06], a system for querying any PADS data source as though it were XML; and an extension to Harmony [FGK+07,PBF+], a system for synchronizing data.

The contributions of this paper are:

- An extension of PADS/ML with a generic tool infrastructure, which permits third parties to easily build new tools for processing PADS data (Section 3).
- A demonstration of how to implement generic programming constructs in OCAML (Section 3).
- Case studies of two non-trivial ad hoc data tools whose functionality was enhanced by using PADS/ML's generic tool infrastructure (Section 4).

We briefly review the PADS/ML data description language in Section 2. We then describe the generic tool framework in Section 3. In Section 4, we describe how the framework was used to build PADX and Harmony. We survey related work and conclude in Section 5.

2 A Review of PADS/ML

In this section, we briefly describe PADS/ML; a more complete description appears in earlier publications [MFW+07,Man06]. A PADS/ML description specifies the physical layout and semantic properties of an ad hoc data source. These descriptions are formulated using types. Base types describe atomic data, such as ASCII-encoded, 8-bit unsigned integers (`puint8`), binary 32-bit integers (`pbint32`), dates (`pdate`), strings (`pstring`), and the singleton types corresponding to literal values. Certain base types take additional OCAML values as parameters. For example, `pstring(c)` describes strings that are immediately followed by the character `c`. Structured types describe compound data built using standard type constructors such as tuples and records for specifying ordered data, variants for specifying alternatives, and lists for specifying homogeneous sequences of data. Type constraints describe data satisfying arbitrary programmer-specified semantic conditions—*e.g.,* that a string `pstring` has at least ten characters. The following subsections illustrate PADS/ML types further using Cisco router configuration files as an example.

2.1 Example: Cisco router configuration

A configuration file for a Cisco router sets the values of parameters that control the router's behavior. The configuration language contains hundreds of commands, and a typical configuration file has hundreds of commands with thousands of parameters. A configuration file lists commands, one per line, where the first word on the line is the command and the remaining words are parameters. A command may depend on other commands, indicated by indentation. Additionally, configurations may include comments, marked by "`!`". Figure 2 shows an excerpt of such a file.

```
version 12.0
!
hostname anaconda
username viking password 5 AF334003CC2
policy-map mis_policy_90:100_output_12K
  class rt_class
    priority
    police cir percent 90 conf-act tx
end
```

Fig. 2. A tiny excerpt of a Cisco router configuration file.

```
ptype command = cmd_name * ' ' * cmd_args
ptype section (min_indent : int) = {
  indent: [i: pstring_ME("/^ */") | length i >= min_indent];
  start_cmd: command; peol;
  sub_cmds: section(length indent + 1) plist(No_sep,Error_term)
}
ptype config_element =
    Section of section (0)
  | Comment of pre "/ *[!].*$/" * peol
ptype source = config_element plist(No_sep,No_term)
```

Fig. 3. Simplified description of Cisco configuration files.

Figure 3 contains a simplified PADS/ML description of the Cisco configuration file format. The description is a sequence of type definitions. The first definition, `command`, describes a single command consisting of a command name followed by its arguments. The `section` type describes a group of related commands. A command is deemed to be a child of an earlier command if its indentation is greater. To express this constraint, `section` is parameterized by the expected minimum indentation, and its identation is checked against the parameter. The `section` type is a record with three fields. The first field `indent` describes the indentation preceding every command. It detects a decrease in indentation level signals, which signals the end of a command group, using a *constraint*. The second field, `start_cmd`, describes the first command of the section, and the third field, `sub_cmds`, describes the list of its subcommands.

The `plist` constructor defining the subcommand list takes three parameters: on the left, the element type; on the right, an optional *separator* that delimits list elements, and an optional *terminator*. In this example, the list has no separators; it terminates when it encounters an element with an error. Next, `config_element` uses a variant type to indicate that an element of a configuration file can be either a `section` or a comment line. Lastly, the type `source` describes a complete Cisco configuration file as a list of elements with no separator and no special terminator. It is terminated with the default terminator of the end-of-file.

2.2 Compiling PADS/ML

Given a description, the PADS/ML compiler creates an OCAML library containing types for the in-memory representation and for the parse-descriptor body for each type in the

```
type source = Config_element.rep plist
type source_pd_body = Config_element.pd_body plist_pd_body Pads.pd
module Source :
  sig
    type rep = source
    type pd_body = source_pd_body
    type pd = pd_body Pads.pd
    val  parse : Pads.handle -> rep * pd
    val  print : rep -> pd -> Pads.handle -> unit
    module MakeTyrep (GenFunTys:GenFunTys.S) : sig
      val tyrep : ...
    end
    ...
  end
```

Fig. 4. Selected software artifacts generated from the `source` type.

PADS/ML description. It also contains a module with functions for parsing and printing the data, and a functor for creating a runtime representation of the types. Figure 4 shows the signature of the module produced for the type `source` from Figure 3.

For reference, we note that the structure of the parse descriptor reflects that of the representation. Every parse descriptor has a header with meta data describing the entirety of the corresponding representation (error information, *etc.*), and a body, consisting of descriptors for each subcomponent. Therefore, every parse descriptor has the type `pd_header * 'pdb`, for some parse-descriptor body type `'pdb`. We use the abbreviation `'a pd = pd_header * 'a` to express this structure.

3 Generic Programming for Ad hoc Data

In a data-processing pipeline, several steps typically occur between parsing and printing. Some steps may be application specific, but many others can be expressed generically and applied to data of any type. Examples include compression and decompression, pretty printing, flattening, database formatting, cleaning, querying, conversion to and from generic formats such as XML and S-expressions, summarization, data generators (*e.g.,* for testing), and transformations like those described in the "scrap-your-boilerplate" series [LP03,LP04,LP05]. Given the variety and number of generic operations, we wish to provide third-party developers with a mechanism to express such operations, without having to modify the PADS/ML compiler.

We use the term "generic" to mean *type indexed*. A type-indexed function defines a family of functions, with one member of the family for every type in the index. A type-indexed function can be constrasted with a *polymorphic* function, which is a single function that can be used at many different types. Because PADS/ML descriptions consist of types, it is natural to express algorithms that are generic to any description as functions indexed by the types of the in-memory representation and the parse descriptor.

Previously, we introduced a generic-tool framework for PADS/ML to support third-party tool development [MFW+07]. While this framework was sufficient to code a number of useful functions, it had limitations. Specifically, it only supported functions that

```
type summary = ... (* uniform data summary type. *)
type seed =  ... (* seed value used in data generation. *)
('r,'b) pretty_print = 'r -> 'b Pads.pd -> string
('r,'b) flatten      = 'r -> 'b Pads.pd -> (string * string) list
('r,'b) decompress   = in_channel -> 'r * ('b Pads.pd)
('r,'b) summarize    = 'r -> 'b Pads.pd -> summary -> summary
('r,'b) clean        = 'r * 'b Pads.pd -> 'r * 'b Pads.pd
('r,'b) generate     = seed  -> 'r * ('b Pads.pd)
```

Fig. 5. Type constructors for selected generic functions.

could be implemented by consuming the in-memory representation of PADS/ML data in a single depth-first, left-to-right traversal.

In this section, we present a fully redesigned framework that supports a much broader range of generic functions. We begin with an overview of our new framework. Then, we provide two examples of how the system is used, from the perspectives of both the user and the tool developer. Finally, we will describe the implementation of the generic tool framework, including the details of type representations.

3.1 Overview

In our generic-programming architecture, three different "actors" cooperate to build and use each generic function f: the end user, the PADS/ML compiler, and the tool developer. When a user wants to apply a generic function f to data of a particular type τ, she needs to *specialize* f to τ, that is, select the member of f appropriate to τ. We use the notation $f[\tau]$ to denote this member. Note that for every type-indexed function f, there is a type constructor σ that relates the type indices of f to the types of members of f—specifically, $f[\tau] : \sigma(\tau)$. For example, Figure 5 lists type constructors for some useful generic functions.

While, conceptually, specialization involves types τ, in reality, OCAML provides no way to manipulate, or even access, types in code. Therefore, we must encode type indexes as runtime values, which we call *type representations*. A function `specialize`, defined in the PADS/ML runtime, instantiates generic functions to particular types using the type's runtime representation. All type representations are built from a set of combinators, which we will describe in greater detail at the end of this section. In principle, the user can use the combinators to construct type representations by hand. In practice, though, such constructions are tedious boilerplate and therefore best generated automatically. Therefore, the PADS/ML compiler generates the runtime representations for each PADS/ML type, along with all of the other generated software artifacts.

The tool developer is responsible for writing f as a type-indexed function. In OCAML, a natural way to express such functions is by pattern matching on a representation of the type. Therefore, the developer implements f by specifying the generic function's behavior for each PADS/ML type constructor, including base types, records, tuples, variants, and user-defined types. Note that the role of "tool developer" might be played by a range of users, from PADS/ML developers to data analysts. Our goal is that tool developers should not need any expertise in PADS/ML internals to be productive, although we

```
<Left>
  <fst>""</fst>
  <snd>
    <fst><fst>version</fst><snd>12.0</snd></fst>
    <snd/>
  </snd>
</Left>
```

Fig. 6. Cisco config command `version 12.0` encoded in XML using a canonical, sums-of-products schema.

do expect a higher level of programming expertise for tool developers than for average PADS/ML users.

3.2 Example: conversion to XML

We begin with an example use of a generic function `to_xml` that translates any PADS data to a corresponding canonical XML representation.[3] This canonical representation uses one schema to encode all data as anonymous sums of products. We explain our choice of this simple encoding when we dicuss the implementation of `to_xml`.

In the example, the end-user wants to translate Cisco configuration data into XML, so she needs to specialize the generic function `to_xml` to the `source` type from the Cisco description of Figure 3. The user might perform this conversion as follows:

```
module SourceTyrep = Cisco.Source.MakeTyrep(TXTys)
let source_to_xml = specialize to_xml SourceTyrep.tyrep
let r,pd = ... Cisco.Source.parse ... ;;
let source_xml = source_to_xml r pd "Config"
```

In the first line, she creates a representation of the type `source` by applying the functor `MakeTyRep`, generated by the compiler for this purpose, to the module `TXTys` defined by the `to_xml` tool writer to specify the type structure of that tool. In the second line, she specializes the generic function `to_xml` to the type `source`, creating the function `source_to_xml`. She then parses the configuration file to create a data representation `r` and its corresponding parse descriptor `pd`. Finally, she applies the specialized conversion function to `r`, `pd`, and a tag for the resulting XML element, yielding an XML representation of the data. Figure 6 shows the result of converting the command "`version 12.0`" in Figure 2 into XML using the `to_xml` function.

Next, we turn to the tool developer's task of implementing the generic function `to_xml`. Figure 7 shows an excerpt of the code. The first four lines define the type constructor `TXTys`, which describes the types of the specializations of the generic function `to_xml`. The implementation of the generic function follows. It is actually a record with one field for each type constructor that can appear in a type index. Each field defines a function that specifies how the generic function behaves for the corresponding type constructor.

[3] We presume a type `xml` with two constructors: `PCData`, for atomic values, and `Element`, for structured values; and a pretty-printer for such values.

```
module ToXMLTycon  =
struct
  type ('a,'pdb) t = 'a -> 'pdb pd -> string -> xml
end
module TXTys = GenFunTys.MakeGeneric(ToXMLTycon)
let rec to_xml = { TXTys.
  int = (fun i (hdr,()) tag ->
     Element(t,[PCData(string_of_int i)]));
  tuple = (fun a_ty b_ty (a,b) (hdr,(a_pd,b_pd)) tag ->
     let a_xml = specialize to_xml a_ty a a_pd "fst" in
     let b_xml = specialize to_xml b_ty b b_pd "snd" in
       Element(tag,[a_xml;b_xml])
  );
  sum = (fun a_ty b_ty v (hdr,v_pdb) tag ->
     match v,v_pdb with
       Left a, Left a_pd  ->
          Element(tag,[specialize to_xml a_ty a a_pd "Left"])
     | Right b,Right b_pd ->
          Element(tag,[specialize to_xml b_ty b b_pd "Right"])
  );
  defined = (fun a_ty (from_t, to_t) (from_pdb, to_pdb)
                 t (hdr,t_pdb) tag ->
     specialize to_xml a_ty (from_t t) (hdr,(from_pdb t_pdb)) tag
  );
}
```

Fig. 7. Excerpt of a generic converter to XML.

For the sake of brevity, we have simplified the implementation of `to_xml`. In the full implementation, there are cases for most of OCAML's base types and a default case. The cases for sums and products have additional parameters to support n-ary sums and products with field and constructor names, and nullary constructors, thereby fully supporting OCAML's named records and variant types. Additionally, parse descriptor headers are included in the XML when they indicate an error in the data.

The case (*i.e.,* field) for integers, `int`, takes the representation of a parsed integer and its parse descriptor as arguments. It returns a representation of that integer wrapped in the XML constructor `PCData`. More interesting is the field `tuple`, which corresponds to the case for binary products. The first two arguments, `a_ty` and `b_ty`, represent the types of the tuple components. They are used to specialize `to_xml` for use with those components. The next two arguments are the tuple to be converted and its parse descriptor. The last argument is the tag to be used when constructing the XML element. The first two lines of the function body translate the components into XML by specializing `to_xml` to the type of each tuple component and applying the result to the appropriate component. Note that this "appropriateness" is statically checked by the OCAML compiler (that is, unless the components have the same type, inverting the type representations will result in a type error.) Finally, the XML for the components is bundled into a single element with the tag specified by the final argument.

```
module FromXMLTycon =
struct
  type ('a,'pdb) t = xml -> ('a * 'pdb pd)
end
module FXTys = GenFunTys.MakeGeneric(FromXMLTycon)
let rec from_xml = { FXTys.
  int = (fun Element(_,[PCData(s)]) ->
     try int_of_string s, (good_hdr,())
     with Failure "int_of_string" -> 0, (error_hdr,()));
  tuple = (fun pos a_name a_ty b_ty Element(_,[a_xml;b_xml]) ->
     let (a,a_pd) = a_ty from_xml a_xml in
     let (b,b_pd) = b_ty from_xml b_xml in
       (a,b),(valid_hdr,(a_pd,b_pd))
  );
  sum = (fun pos a_name a_ty a_empty b_ty b_empty -> function
       Element(_,[Element("Left",_) as a_xml]) ->
         let a,a_pd = a_ty from_xml a_xml in
           Left a, (valid_hdr, Left a_pd)
     | Element(_,[Element("Right",_) as b_xml]) ->
         let b, b_pd = b_ty from_xml b_xml in
           Right b, (valid_hdr, Right b_pd)
  );
  defined = (fun a_ty (from_t, to_t) (from_pdb, to_pdb) a_xml ->
     let a,(h,a_pdb) = a_ty from_xml a_xml in
       (to_t a),(h,to_pdb a_pdb)
  );
}
```

Fig. 8. Generic converter from XML.

The case for binary sums follows the same pattern as that of binary tuples. For user-defined types, we borrow from Hinze [Hin04], requiring the tool writer to use functions that convert between the user-defined type and a sum-of-products type (similarly for the parse descriptor). These compiler-generated conversions are supplied by the caller of the tool as the second and third arguments of the `defined` field (the first argument is a representation of the sum-of-products type). In our example, we are mapping from a type to XML, so we use the "from" conversion function.

3.3 Example: conversion from XML

A significant improvement in the new generic interface for PADS/ML is that it does not limit developers to writing functions that consume data. To illustrate this point, we define in Figure 8 the implementation of a generic function `from_xml` that *produces* data of a given type from XML input.

The implementation mirrors that of `to_xml`. The first four lines specify the type constructor for the generic function and create the type of the `from_xml` generic function. The field definitions for `from_xml` follow the same pattern as those for `to_xml`, producing data rather than consuming it. One difference relates to parse descriptors.

The type constructor for `from_xml` requires a parse descriptor along with the reconstructed data. But we discarded such descriptors when converting to XML, so we need to recreate them now. In most cases, we simply provide a place-holder `valid_hdr` to indicate the data is error free. For the `int` field, however, we check that the string in the XML is a valid integer and report errors using the parse descriptor.

3.4 Other generic functions

All our example tools are self contained in that they make no reference to other generic functions. Our framework, however, permits generic functions that depend on other generic functions, and even mutually recursive generic functions. The only limitation is that such functions must all share the same generic-function type constructor.

3.5 Type representations

We now discuss the implementation of type representations, reusing the `to_xml` generic function from above for an example. The tool developer implemented `to_xml` as a record with one field for each type constructor. The end user specialized this generic function implementation to a particular PADS/ML type τ by applying the `specialize` function to a representation of the type τ. The expression `specialize to_xml` has the polymorphic type

```
specialize to_xml : ('r,'p) tyrep -> 'r -> 'p pd -> xml
```

Notice that the type constructor σ of `to_xml` is expressed implicitly in this type.

While the runtime provides the definition of the `specialize` function, it is the task of the compiler to produce the representation of the PADS/ML type τ. Following Yang's approach [Yan98], we choose to represent each PADS/ML type τ as a function that takes as an argument a generic function, (*i.e.,* a record of functions, each field specifying the behavior of the generic function for one type constructor) and selects the field of the generic function corresponding to τ. If τ is a simple type, that is all the type representation function need do. If τ is a type constructor, the type representation function then applies the selected function to the representations of the arguments of the type constructor.

For example, the representation of the PADS/ML type `(pint*pint)` is:

```
fun gf -> gf.tuple (fun gf -> gf.int) (fun gf -> gf.int)
```

This function takes a generic function `gf` as an argument and selects the `tuple` field. Because tuples are type constructors with two arguments, the type representation function for the pair then applies this selected function to the type representation of the arguments, `pint`. This representation type function simply selects the `int` field from the generic function `gf`.

With this choice for the representation of types, the definition of the `specialize` function is trivial— it is just function application: `fun gf ty -> ty gf`. This one definition handles all generic functions and all type representations. In contrast, the compiler must generate a different type representation for each PADS/ML type in a description.

```
type to_xml_record = {
  int     : int -> unit pd -> xml
  tuple   : 'a 'b 'p 'q. ('a,'p) type_rep -> ('b,'q) type_rep
            -> ('a * 'b) -> ('p pd * 'q pd) pd -> xml
  sum     : 'a 'b 'p 'q. ('a,'p) type_rep -> ('b,'q) type_rep
            -> ('a,'b) sum -> ('p pd,'q pd) sum pd -> xml
  defined : 'a 'p 'u 'q. ('a,'p) type_rep -> ('a,'u) iso
                    -> ('p,'q) iso -> 'u -> 'q pd -> xml
}
and ('r,'p) type_rep = to_xml_record -> 'r -> 'p pd -> xml
```

Fig. 9. The type of `to_xml`.

The type system of OCAML ensures that the application of a generic function to a type representation will never go wrong, but getting our choice for type representations as functions to typecheck in OCAML takes a bit of engineering. To illustrate, we again turn to the `to_xml` example. Figure 9 defines the type `to_xml_record`, which is the type of the generic function implementation `to_xml`. Notice that the record fields contain first-class polymorphic functions. This flexibility is essential because the representation of a PADS/ML type might need to apply the same field to several distinct types, *e.g.,* for a PADS/ML type containing more than one kind of tuple. Figure 9 also defines the type constructor `type_rep`, which is the type of the representation of all PADS/ML types for the `to_xml` generic function. As an example, the type of the representation of the the PADS/ML type `(pint*pint)` is

```
to_xml_record -> int*int -> (int_pd*int_pd) pd -> xml
```

which is just `(int*int,int_pd*int_pd) type_rep`.

3.6 Tool-independent type representations

The types in Figure 9 describe the `to_xml` generic function very precisely; too precisely, in fact. Those types and the type representations built from them are specific to `to_xml` and could not be used for other generic function, for example, `from_xml`. To support a wide range of different generic functions, we follow Yang's approach and provide tool-*independent* type representations and record types, by abstracting away the pieces that are particular to each generic function. Specifically, we must abstract away the type constructor associated with the generic function.

Here we encounter a problem: abstracting over a type constructor requires support for higher-order polymorphism, a feature not provided in OCAML's core language. Therefore, we turn to OCAML's module system and use a functor to do the abstraction. Figure 10 shows a simplified excerpt of such a functor, `MakeGeneric`, provided by the PADS/ML runtime. This functor defines the type of the representation of PADS/ML types `type_rep` and the type of the generic-function record for all generic functions associated with the type constructor `t`, passed as an argument to the functor. Conceptually, the types we described earlier in this section, `to_xml_type`, *etc.*, result from applying this functor, although in doing the abstraction, we added a parameter to the type

constructor `t` so that a single instance of this functor will be able to express the necessary types for a wider range of generic functions. Note that we have simplified the presentation of this functor in the same way that we simplified `to_xml` and `from_xml` – specifically, we have left out a number of cases and the parameters to `tuple` and `sum` that provide full support for OCAML's records and variant types.

```
type 'a pd = pd_hdr * 'a
type ('a,'t) iso = ('t -> 'a) * ('a -> 't)
type ('l,'r) sum = Left of 'l | Right of 'r
module MakeGeneric(GenFunTycon: sig type ('r,'pdb,'s) t end) :
sig
  type ('r,'pdb,'s) gf_tycon = ('r,'pdb,'s) GenFunTycon.t
  type 's gf_record = {
    int    : (int,    unit, 's) gf_tycon;
    tuple : 'a 'b 'a_pdb 'b_pdb.
       ('a,'a_pdb,'s) type_rep ->
       ('b,'b_pdb,'s) type_rep ->
       ('a * 'b, ('a_pdb pd * 'b_pdb pd) pd, 's) gf_tycon;
    sum : 'a 'b 'a_pdb 'b_pdb.
       ('a,'a_pdb,'s) type_rep ->
       ('b,'b_pdb,'s) type_rep ->
       (('a,'b) sum, ('a_pdb pd,'b_pdb pd) sum pd, 's) gf_tycon;
    defined : 'a 'r 'a_pdb 'r_pdb.
       ('a,'a_pdb,'s) type_rep ->
       ('a,'r) iso -> ('a_pdb,'r_pdb) iso ->
       ('r,'r_pdb,'s) gf_tycon;
  }
  and ('r,'pdb,'s) type_rep =
     's gf_record -> ('r,'pdb,'s) gf_tycon
end
```

Fig. 10. A simplified excerpt of the signature of functor `MakeGeneric` for making generic-function types. This functor is located in the `GenFunTys` module, which is part of the PADS/ML runtime.

The issue of higher-order polymorphism arises in the definition of the representation of PADS/ML types as well because the representations reference the labels of the generic-function record. Hence, the definition of the representation of each PADS/ML type is given in a compiler-generated functor `MakeTyrep`, parameterized by the type of the generic function.

To summarize, the generic function infrastructure provided by PADS/ML has three main components: the function `specialize` and the functor `MakeGeneric`, defined once, and the functor `MakeTyrep`, which is generated for each PADS/ML type in a given description. A tool developer writing a generic function with associated type constructor σ uses the functor `MakeGeneric` to produce the type of the generic function that she must define. The user of the generic function uses the functor `MakeTyrep` to produce a representation of the PADS/ML type suitable for use with that generic function.

```
('a, 'pdb, 's) consumer = 'a -> 'pdb Pads.pd -> 's
('a, 'pdb, 's) producer = 's -> 'a * ('pdb Pads.pd)
('a, 'pdb, 's) updater  = 'a * 'pdb Pads.pd -> 'a * 'pdb Pads.pd
flatten      : ('a,'pdb,(string * string) list) consumer
pretty_print : ('a,'pdb, string) consumer
summarize    : ('a,'pdb, summary -> summary) consumer
to_xml       : ('a,'pdb, xml list) consumer
decompress   : ('a,'pdb,in_channel) producer
generate     : ('a,'pdb, seed) producer
from_xml     : ('a,'pdb, xml list) producer
clean : ('a,'pdb,unit) updater
```

Fig. 11. Classes of generic functions.

An apparent disadvantage of this functorized approach is that a given type representation can only be applied to one generic function – the one corresponding to the type constructor for which it was instantiated. However, this limitation is not as restrictive as it might seem. The type constructor at which a type representation is instantiated can be far more general than a single generic function and can encompass a family of functions using the extra type parameter `'s`. For example, Figure 11 shows how to rewrite the function types in Figure 5 in terms of only three generic function classes: consumers, producers, and updaters.

4 Case Studies

Converting ad hoc data to XML is only one of many possible applications of our generic function framework. In this section, we discuss two other uses of the framework.

4.1 PADX/ML

In previous work [FFGM06], we reported on our experience designing and implementing PADX, a system for querying large-scale PADS data sources with XQuery [Kat04], a standardized query language for XML. PADX synthesizes and extends two existing systems: PADS/C and Galax [FSC+03]. With PADX, an analyst writes a PADS description of her ad hoc data, and the PADS/C compiler produces two software artifacts: an XML Schema that specifies the virtual XML view of the corresponding PADS data and a customized library for viewing it as XML. The resulting library is linked with the Galax query engine, permitting the analyst to query ad hoc data sources using XQuery.

We were pleased with PADX's functionality. The unified system gave us a standard, high-performance engine for querying ad hoc data without having to build one from scratch. We were frustrated, however, by the implementation and its limitations. We made substantial modifications to the PADS/C compiler to generate PADX's software artifacts, which required 1050 lines of Standard-ML, 2117 lines of C, and 350 lines of OCAML. The generated libraries were large, *e.g.,* the library for the Sirius description [FG05] was more than 7000 lines of C and used C macros extensively, making the code hard to understand and debug. Most significantly, the changes only supported PADX and were incomplete: PADX can map from PADS data to XML but not vice versa.

Using the generic tool framework, the implementation of PADX/ML is more complete, simpler, and more flexible than that of PADX. The PADX/ML consumer tool maps PADS/ML representations and parse descriptors into values in Galax's abstract XML data model (XDM) (*i.e.,* sequences of elements and XML scalar values), and the PADX/ML producer tool does the inverse, enabling the output of XQuery expressions to be represented as PADS data. Together, the tools are implemented in only 884 lines of OCAML.

The PADX consumer yields a completely lazy tree, which permits the Galax query engine to cope with large-scale data more efficiently. Each XML element in Galax's XDM roughly corresponds to a node in the consumer's lazy tree. The consumer is lazy "all the way down", that is, the consumer does not parse a PADS element in a data source until its corresponding node in the XDM is forced. This laziness is important to query performance. For some queries, Galax can produce query plans that access a virtual XML source sequentially using memory bounded by the query size, not the data size. This optimization is only possible if the underlying data source is itself lazy.

The PADX producer maps values in the Galax XDM into PADS/ML. Given a producer specialized on a type and an XML value in the Galax XDM, the producer simply performs a pattern match on the XML value to map it into the corresponding PADS/ML value. When a match fails, a parse-descriptor header is returned, indicating a syntax error. To apply a producer, however, requires knowing the correspondence between an XML value and an extant, unparameterized PADS/ML type. This correspondence can be recovered by validating an XML value with respect to any PADS/ML-generated XML Schema, as each XML Schema type corresponds one-to-one with a PADS/ML type. If validation succeeds, the XML value is labelled with its corresponding XML Schema type. The compiler produces a meta-data table that given an XML Schema type name selects a specialized producer for the corresponding PADS/ML type.

We did make one modification to the PADS/ML compiler for PADX to generate the XML Schema for a PADS/ML specification. A generic type-consumer tool would avoid this problem, by permitting computation over any PADS/ML type, just like the generic value-consumer tool permits computation over the representations and parse descriptors of any PADS/ML value. No technical issue prevents us from providing a generic type-consumer tool, but it is not yet implemented.

4.2 Harmony

In our second case study, we use our generic infrastructure with the Harmony synchronization framework [PBF$^+$]. An instance of Harmony takes as inputs two *replicas* containing data to be synchronized, an *archive* representing their last synchronized state, and a *schema* describing the set of well-formed replicas. Harmony's synchronization algorithm merges non-conflicting changes made to each replica relative to the archive and subject to the constraints expressed in the schema, and produces as outputs maximally-synchronized replicas (and an updated archive). Harmony instances exist for synchronizing browser bookmarks, calendars, address books, and structured documents.

To simplify the synchronization algorithm—in particular, the task of aligning and identifying the common data in each replica—Harmony's internal data model is unordered trees and not a richer model like XML. Working with unordered trees makes synchronization simpler, but introduces a "last-mile problem"—most data is not stored

```
{Section={indent={""},
          start_cmd={elt1={version={}},
                     elt2={"12.0"={}}}}}
```

Fig. 12. Cisco config command `version 12.0` encoded as an unordered tree.

as unordered trees. Therefore, before the replicas can be processed using Harmony they need to be parsed, and likewise after synchronization, the updated replicas must be serialized to their original formats. Harmony currently relies on a collection of custom "viewers"—i.e., parsers and corresponding pretty printers—for a variety of on-disk formats (XML, CSV, iCalendar, and Palm Datebook) to bridge this gap. These viewers are not ideal, however, being tedious to write and difficult to maintain. Moreover, every new format requires its own custom viewer. A better solution is to use a generic tool to generate a viewer from a PADS description.

We have implemented generic tools for the unordered tree data model analagous to the `to_xml` and `from_xml` tools for XML. The generic consumer takes a PADS representation of a data value and yields a Harmony tree. The generic producer maps a Harmony tree back to a PADS representation. The representation of a data value as an unordered tree is determined by its type: base type values are represented as trees with a single child whose label encodes the value; records are represented as trees with a child for every field; a value belonging to a variant type is represented as a tree with a single child whose label is the tag; and lists are represented using a cons-cell encoding. Figure 12 shows how the Cisco line from the earlier example is represented as an unordered tree (writing "{" for internal tree nodes and "=" for subtrees).

These generic tools provide effective conduits between arbitrary on-disk representations of ad hoc data described in PADS and Harmony's internal data model. We plan to use them to build Harmony instances for several new data formats in the near future.

5 Discussion

Our generic programming framework combines two existing techniques: Yang's theoretical account of type-indexed values and their encoding in ML-like languages using the ML module system [Yan98], and Hinze's framework for generic programming using Haskell's type classes [Hin04]. We compare our work to these approaches in turn.

We make a number of improvements to Yang's original presentation. First, his theoretical encoding requires first-class polymorphism, which at the time was only available in the ML module system. Now that OCAML provides first-class polymorphism within records, his encoding can be expressed in a significantly more lightweight manner. Second, we have generalized his theoretical encoding to support the definition of generic functions based on other generic functions. Finally, Yang did not support user-defined recursive types, which we address using techniques based on Hinze's work.

While Yang's work showed how to implement generics in OCAML, Hinze's work is the most closely related in terms of the interface it provides to users and generic function developers. The essential difference between Hinze's framework and our own is that Hinze's solution works for Haskell, while ours is for OCAML. This difference manifests itself most notably in that Hinze uses Haskell's type classes to parameterize over type

constructors, and so he manages type representations implicitly as dictionaries. We use OCAML's module system for parameterization and our type representations must be passed explicitly, which provides more control over instantiation at the price of some syntactic overhead. An important practical but less essential difference is that we have adapted our system for use with PADS/ML, incorporating parse descriptors and requiring tools to implement a case for constrained types.

The "scrap-your-boilerplate" series of papers [LP03,LP04,LP05] presents another approach to generic programming in HASKELL. Recently, members of the Gallium project have added support for similar functionality to OCAML with the new camlp4 system [cam]. We refer readers to the SYB papers for a full comparison of SYB to other generic-programming approaches, including this one.

Shortly before this paper was ready for publication, Yallop [Yal07] and Karvonen [Kar07] published works on generic programming in ML. Due to lack of time to fully review their work, we offer only basic comparisons. Yallop's work uses camlp4 – an OCAML preprocessor – to extend OCaml with a deriving construct, similar to that found in Haskell. As Yallop points out in his conclusion, while this approach offers a convenient way to use the generic functions, it does not address the challenge of writing new generic functions, which is exactly the goal of the current work. Karvonen's work is more closely related in that it supports generic programming within ML itself, rather than in a preprocessor. However, Karvonen is working within the confines of Standard ML, which lacks recursive values and first-class polymorphism. Hence, the challenges he faces are somewhat different, as is his resulting solution.

The most direct contributions of the present work are both related to PADS/ML: the extension of PADS/ML's support for third-party development of type-directed tools and the description of two non-trivial tools built using this extension. However, both of these contributions have broader relevance. The latter, because PADX/ML and extended Harmony provide compelling examples of the applicability of generic programming techniques to real-world challenges. The former, because the generic programming framework that we present is relevant to OCAML developers in general, not just those interested in PADS/ML.

Acknowledgments

We would like to thank the anonymous reviewers for their helpful comments on the paper and its organization.

References

[cam] Camlp4 - Gallium. http://brion.inria.fr/gallium/index.php/Camlp4.

[Con] Gene Ontology Consortium. Gene ontology project. http://www.geneontology.org.

[FFGM06] Mary Fernández, Kathleen Fisher, Robert Gruber, and Yitzhak Mandelbaum. PADX: Querying large-scale ad hoc data with XQuery. In *PLAN-X*, 2006.

[FG05] Kathleen Fisher and Robert Gruber. PADS: A domain-specific language for processing ad hoc data. In *PLDI*, 2005.

[FGK+07] J. Nathan Foster, Michael B. Greenwald, Christian Kirkegaard, Benjamin C. Pierce, and Alan Schmitt. Exploiting schemas in data synchronization. *Journal of Computer and System Sciences*, 73(4), June 2007.

[FSC+03] Mary F. Fernández, Jŕôme Siméon, Byron Choi, Amélie Marian, and Gargi Sur. Implementing XQuery 1.0: The Galax experience. In *VLDB*, pages 1077–1080, 2003.

[Hin04] Ralf Hinze. Generics for the masses. In *ICFP*, 2004.

[Kar07] Vesa A.J. Karvonen. Generics for the working ml'er. In *ML Workshop*, 2007.

[Kat04] Howard Katz, editor. *XQuery from the experts*. Addison Wesley, 2004.

[LP03] Ralf Lämmel and Simon Peyton Jones. Scrap your boilerplate: a practical design pattern for generic programming. In *TLDI*, 2003.

[LP04] Ralf Lämmel and Simon Peyton Jones. Scrap more boilerplate: reflection, zips, and generalised casts. In *ICFP*, 2004.

[LP05] Ralf Lämmel and Simon Peyton Jones. Scrap your boilerplate with class: extensible generic functions. In *ICFP*, 2005.

[Man06] Yitzhak Mandelbaum. *The Theory and Practice of Data Description*. PhD thesis, Princeton University, September 2006.

[MFW+07] Yitzhak Mandelbaum, Kathleen Fisher, David Walker, Mary Fernandez, and Artem Gleyzer. PADS/ML: A functional data description language. In *POPL*, 2007.

[MHS+05] R. Mandelbaum, C. M. Hirata, U. Seljak, J. Guzik, N. Padmanabhan, C. Blake, M. R. Blanton, R. Lupton, and J. Brinkmann. Systematic errors in weak lensing: application to SDSS galaxy-galaxy weak lensing. *Mon. Not. R. Astron. Soc.*, 361:1287–1322, August 2005.

[PBF+] Benjamin C. Pierce, Aaron Bohannon, J. Nathan Foster, Michael B. Greenwald, Sanjeev Khanna, Keshav Kunal, and Alan Schmitt. Harmony: A synchronization framework for heterogeneous tree-structured data. http://www.seas.upenn.edu/ harmony/.

[Yal07] Jeremy Yallop. Practical generic programming in ocaml. In *ML Workshop*, 2007.

[Yan98] Zhe Yang. Encoding types in ML-like languages. In *ICFP*, 1998.