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*, *implementation* and *theory* of PADS/ML, a new language and system that facilitates 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. Finally, the PADS/ML theory gives a precise formal meaning to the descriptions in terms of the semantics of parsing, the semantics of printing, and the types of data structures that represent parsed data.

Name: Use	Representation
Gene Ontology (GO) [6]: Gene Product Information	Variable-width ASCII records
SDSS/Reglens Data [22]: 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.

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 ';').

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, while 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, financial analysts, *etc.* 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 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

```
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.

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 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) [12] 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. Section 5 presents the extended DDC^{α} calculus, focusing on the semantics of polymorphic types for parsing and the key elements of the printing semantics.

PADS/ML has evolved from previous work on PADS/C [1] [11], 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 though 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.

In summary, this work makes the following key 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.
- We have defined the formal semantics of both PADS/ML parsers and printers and proven our generated code is type safe and well-behaved as defined by a canonical forms theorem.

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.

[1] We refer to the original PADS language as PADS/C to distinguish it from PADS/ML.

```
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.

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

Literal characters in type expressions denote 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. String, character, and integer literals can be embedded in a description and are interpreted as singleton types, *e.g.*, the singleton type `"0|"` in the `Summary_header` type denotes the string literal `"0|"`.

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 boolean `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 format, a flat representation of trees used by biologists [25]. Example Newick 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.

```
(* 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 omit "DOMESTIC"
  | INTERNATIONAL of omit "INTERNATIONAL"
  | SPECIAL       of omit "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.

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 (shown in Appendix A) 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` 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.

The last construct in the Regulus description is the type qualifier `omit`. In the `Service` datatype, `omit` specifies that the parsed string literal should be omitted in the internal data representation because the literal can be determined by the datatype constructor.

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 generated O'CAML 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` containing three named types results in the O'CAML file `sirius.ml` defining the module `Sirius`, which contains 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 [8], 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 = (Pads.pd_header * Alpha.pd_body) *
                 (Pads.pd_header * Beta.pd_body)
  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.

```
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 _ = split_orders orders orders_pd
```

Figure 5. Error filtering of Sirius data

```
...
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.

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, and user code only needs to traverse nested parse descriptors if more precise information about the 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.

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

```
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 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.

"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. 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 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. However, there is a limit, both in resources and expertise, to the range of tool generators that we can develop. Indeed, new and interesting data analysis tools are constantly being developed, and we have no hope of integrating even a fraction of them into the PADS/ML system ourselves. Therefore, it is essential that we provide a simple framework for others to develop tool generators.

The techniques of type-directed programming, known variously as *generic* [16] or *polytypic* [19] programming, provide a convenient conceptual starting point in designing a tool framework. In essence, any tool generator is a function from a description to the corresponding tool. As PADS/ML descriptions are types, a tool generator is a type-directed program.

```
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`.

Support for some form of generic programming over data representations and parse descriptors is an essential first step in supporting the development of tool generators. While a full-blown generic programming system like Generic Haskell [17] would be useful in this context, O'CAML lacks a generic programming facility. All is not lost, however, as a number of useful data processing tools share a common computational paradigm, and we can support that paradigm without full generic programming support.

In particular, many of the tools we have encountered perform their computations in a single pass over the representation and corresponding parse descriptor, visiting each value in the data with a pre-, post-, or in-order traversal. This paradigm arises naturally as it scales to very large data sets. It can be abstracted in a manner similar to the generic functions of Lammel and Peyton-Jones [20]. For each format description, we generate a format-dependent traversal mechanism that implements a generalized fold over the representation and parse descriptor corresponding to that format. Then, tool developers can write a format-independent, *generic tool* by specifying the behaviour of the tool for each PADS/ML type constructor. The traversal mechanism interacts with generic tools through a signature that every generic tool must match.

The generic tool architecture of PADS/ML delivers a number of benefits over the fixed architecture of PADS/C. In PADS/C, all tools are generated from within the compiler. Therefore, developing a new tool generator requires understanding and modifying the compiler. Furthermore, the user selects the set of tools to generate when compiling the description. In PADS/ML, tool generators can be developed independent of the compiler and they can be developed more rapidly because the "boilerplate" code to traverse data need not be replicated for each tool generator. In addition, the user controls which tools to "generate" for a given data format, and the choice can differ on a program-by-program basis.

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 traver-

```
<Order_header size="13" status="GOOD">
 <order_num><val>9153</val></order_num>
 <att_order_num><val>9153</val></att_order_num>
 <ord_version><val>1</val></ord_version>
 <service_tn>
    <Something><val>0</val></Something>
 </service_tn>
 <billing_tn>
    <Something><val>0</val></Something>
 </billing_tn>
 <nlp_service_tn>
    <Something><val>0</val></Something>
 </nlp_service_tn>
 <nlp_billing_tn>
    <Something><val>0</val></Something>
 </nlp_billing_tn>
 <zip_code><Nothing><val></val></Nothing></zip_code>
 <ramp><Ramp><val>152268</val></Ramp></ramp>
 <order_sort><val>LOC_6</val></order_sort>
 <order_details><val>0</val></order_details>
 <unused><val>FRDW1</val></unused>
 <stream><val>DUO</val></stream>
</Order_header>
```

Figure 9. A fragment of the XML output for Sirius.

sal. 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 `Datatypeinit` 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. 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. Figure 9 shows a sample portion of the output of this tool when run on the Sirius data in Figure 2(b).

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 *etc.* that are not present in the representation. This format is often more readable than the raw data.

$$
\begin{array}{llll}
\text{Kinds} & \kappa & ::= & \mathsf{T} \mid \mathsf{T} \to \kappa \mid \sigma \to \kappa \\
\text{Types} & \tau & ::= & C(e) \mid \lambda x.\tau \mid \tau\, e \mid \Sigma\, x{:}\tau.\tau \mid \tau + \tau \\
 & & \mid & \{x{:}\tau \mid e\} \mid \alpha \mid \mu\alpha.\tau \mid \lambda\alpha.\tau \mid \tau\,\tau \mid \ldots
\end{array}
$$

Figure 10. DDC$^\alpha$ syntax, selected constructs

5. The Semantics of PADS/ML

In this section, we introduce DDC$^\alpha$, a calculus of simple, orthogonal type constructors, which serves to give a semantics to the main features of PADS/ML. DDC$^\alpha$ is an extension and revision of our previous work on DDC [12]. The main new feature is the ability to define functions from types to types, which are needed to model PADS/ML's polymorphic data types. In the process of adding these new functions, which we call *type abstractions* (as opposed to *value abstractions*, which are functions from values to types), we simplified our overall semantics by making a couple of subtle technical changes. For example, we were able to eliminate the complicated "contractiveness" constraint from our earlier work. We have also added a new interpretation of DDC$^\alpha$ types as printers.

The main practical benefit of the calculus has been as a guide for our implementation. Before working through the formal semantics, we struggled to disentangle the invariants related to polymorphism. After we had defined the calculus, we were able to implement type abstractions as O'CAML functors in approximately a week. Our new printing semantics was also very important for helping us define and check the correctness of our printer implementation. We hope the calculus will serve as a guide for implementations of PADS in other host languages. In the remainder of this section, we give an overview of the calculus. Appendix B contains a complete formal specification.

5.1 DDC$^\alpha$ Syntax

Figure 10 summarizes the syntax of the DDC$^\alpha$. The interpretation of a type with kind T is a parser that maps data from an external form into an internal one. A type with kind $\mathsf{T} \to \kappa$ is a function mapping a parser to the interpretation of a type with kind κ. Finally, types with kind $\sigma \to \kappa$ map values with host language type σ to the interpretation of types with kind κ. For concreteness, we adopt F_ω as our host language.

The simplest description is a base type $C(e)$. The base type's parameter e is drawn from the host language. The PADS/ML type `Pstring` is an example of such a base type. Structured types include value abstraction $\lambda x.\tau$ and application $\tau\, e$, which allow us to parameterize types by host language values. The dependent sum type, $\Sigma\, x{:}\tau.\tau$, describes a pair of values, where the value of the first element of the pair can be referenced when describing the second element. Variation in a data source can be described with the sum type $\tau + \tau$, which deterministically describes a data source that either matches the first type, or fails to match the first branch but does match the second one. We specify semantic constraints over a data source with type $\{x{:}\tau \mid e\}$, which describes any value x that satisfies the description τ and the predicate e. Type variables α are abstract descriptions; they are introduced by recursive types and type abstractions. Recursive types $\mu\alpha.\tau$ describe recursive formats, like lists and trees. Type abstraction $\lambda\alpha.\tau$ and application $\tau\,\tau$ allow us to parameterize types by other types. Type variables α always have kind T.

To specify the well-formedness of types, we use a kinding judgment of the form $\Delta; \Gamma \vdash \tau : \kappa$, where Δ maps type variables to kinds and Γ maps host language value variables to host language types. In our original work [12], these kinding rules were somewhat unorthodox, but we have since simplifed them. Details appear in Appendix B.

5.2 Host Language

The host language of DDC^α is a straightforward extension of F_ω with recursion and a variety of useful constants and operators. For reference, the grammar appears in Appendix C. The constants include bitstrings B; offsets ω, representing locations in bitstrings; and error codes `ok`, `err`, and `fail`, indicating success, success with errors, and failure, respectively. We use the constant `none` to indicate a failed parse. Because of its specific meaning, we forbid its use in user-specified expressions appearing in DDC^α types. We use the notation $\texttt{bs}_1 \mathbin{@} \texttt{bs}_2$ to append bit string $\texttt{bs}_1$ to $\texttt{bs}_2$. Our base types include the type `none`, the singleton type of the constant `none`, and types `errcode` and `offset`, which classify error codes and bit string offsets, respectively.

We extend the formal syntax with some syntactic sugar for use in the rest of this section: anonymous functions $\lambda x.e$ for $\texttt{fun}\ f\ x = e$, with $f \notin \mathrm{FV}(e)$; `span` for $\texttt{offset} * \texttt{offset}$. We often use pattern-matching syntax for pairs in place of explicit projections, as in $\lambda(\texttt{B},\omega).\texttt{e}$ and $\texttt{let}\ (\omega,\texttt{r},\texttt{p}) = e\ \texttt{in}\ e'$. Although we have no formal records with named fields, we use a dot notation for commonly occuring projections. For example, for a pair `x` of rep and PD, we use `x.rep` and `x.pd` for the left and right projections of `x`, respectively. Also, sums and products are right-associative. Finally, we only specify type abstraction over terms and application when we feel it will clarify the presentation. Otherwise, the polymorphism is implicit. We also omit the usual type and kind annotations on λ, with the expectation the reader can construct them from context.

The static semantics ($\Delta; \Gamma \vdash e : \sigma$), operational semantics ($e \to e'$), and type equality ($\sigma \equiv \sigma'$) are those of F_ω extended with recursive functions and recursive types and are entirely standard. See Pierce's text [28] for details.

5.3 DDC^α Semantics

The primitives of DDC^α each have four interpretations: two types in the host language, one for the data representation itself and one for its parse descriptor, and two functions, one for parsing and one for printing. We therefore specify the semantics of DDC^α types using four semantic functions, each of which precisely conveys a particular facet of the meaning of a type. The functions $[\![\cdot]\!]_{\text{rep}}$ and $[\![\cdot]\!]_{\text{PD}}$ describe the type of the data's in-memory representation and parse descriptor, respectively. The semantic functions $[\![\cdot]\!]_{\text{P}}$ and $[\![\cdot]\!]_{\text{PP}}$ define the parsing and printing functions generated from DDC^α descriptions.

$$\boxed{[\![\tau]\!]_{\text{rep}} = \sigma}$$

$$\begin{array}{lcl}
[\![C(e)]\!]_{\text{rep}} & = & \mathcal{B}_{\text{type}}(C) + \texttt{none} \\
[\![\lambda x.\tau]\!]_{\text{rep}} & = & [\![\tau]\!]_{\text{rep}} \\
[\![\tau\, e]\!]_{\text{rep}} & = & [\![\tau]\!]_{\text{rep}} \\
[\![\Sigma\, x{:}\tau_1.\tau_2]\!]_{\text{rep}} & = & [\![\tau_1]\!]_{\text{rep}} * [\![\tau_2]\!]_{\text{rep}} \\
[\![\tau_1 + \tau_2]\!]_{\text{rep}} & = & [\![\tau_1]\!]_{\text{rep}} + [\![\tau_2]\!]_{\text{rep}} \\
[\![\{x{:}\tau \mid e\}]\!]_{\text{rep}} & = & [\![\tau]\!]_{\text{rep}} + [\![\tau]\!]_{\text{rep}} \\
[\![\alpha]\!]_{\text{rep}} & = & \alpha_{\texttt{rep}} \\
[\![\mu\alpha.\tau]\!]_{\text{rep}} & = & \mu\alpha_{\texttt{rep}}.[\![\tau]\!]_{\text{rep}} \\
[\![\lambda\alpha.\tau]\!]_{\text{rep}} & = & \lambda\alpha_{\texttt{rep}}.[\![\tau]\!]_{\text{rep}} \\
[\![\tau_1\tau_2]\!]_{\text{rep}} & = & [\![\tau_1]\!]_{\text{rep}}[\![\tau_2]\!]_{\text{rep}}
\end{array}$$

Figure 11. Representation type translation, selected constructs

$$\boxed{[\![\tau]\!]_{\text{PD}} = \sigma}$$

$$\begin{array}{lcl}
[\![C(e)]\!]_{\text{PD}} & = & \texttt{pd\_hdr} * \texttt{unit} \\
[\![\lambda x.\tau]\!]_{\text{PD}} & = & [\![\tau]\!]_{\text{PD}} \\
[\![\tau\, e]\!]_{\text{PD}} & = & [\![\tau]\!]_{\text{PD}} \\
[\![\Sigma\, x{:}\tau_1.\tau_2]\!]_{\text{PD}} & = & \texttt{pd\_hdr} * [\![\tau_1]\!]_{\text{PD}} * [\![\tau_2]\!]_{\text{PD}} \\
[\![\tau_1 + \tau_2]\!]_{\text{PD}} & = & \texttt{pd\_hdr} * ([\![\tau_1]\!]_{\text{PD}} + [\![\tau_2]\!]_{\text{PD}}) \\
[\![\{x{:}\tau \mid e\}]\!]_{\text{PD}} & = & \texttt{pd\_hdr} * [\![\tau]\!]_{\text{PD}} \\
[\![\alpha]\!]_{\text{PD}} & = & \texttt{pd\_hdr} * \alpha_{\texttt{PDb}} \\
[\![\mu\alpha.\tau]\!]_{\text{PD}} & = & \texttt{pd\_hdr} * \mu\alpha_{\texttt{PDb}}.[\![\tau]\!]_{\text{PD}} \\
[\![\lambda\alpha.\tau]\!]_{\text{PD}} & = & \lambda\alpha_{\texttt{PDb}}.[\![\tau]\!]_{\text{PD}} \\
[\![\tau_1\tau_2]\!]_{\text{PD}} & = & [\![\tau_1]\!]_{\text{PD}}[\![\tau_2]\!]_{\text{PDb}}
\end{array}$$

$$\boxed{[\![\tau]\!]_{\text{PDb}} = \sigma}$$

$$[\![\tau]\!]_{\text{PDb}} \quad = \quad \sigma \ \text{ where } [\![\tau]\!]_{\text{PD}} \equiv \texttt{pd\_hdr} * \sigma$$

Figure 12. Parse-descriptor type translation, selected constructs

DDC^α *representation types.* In Figure 11, we present the representation type of selected DDC^α primitives. While the primitives are dependent types, the mapping to the host language erases the dependency because the host language does not have dependent types. This involves erasing all host language expressions that appear in types as well as value abstractions and applications. A type variable α in DDC^α is mapped to a corresponding type variable $\alpha_{\texttt{rep}}$ in F_ω. Recursive types generate recursive representation types with the type variable named appropriately. Polymorphic types and their application become F_ω type constructors and type application, respectively.

DDC^α *parse descriptor types.* Figure 12 gives the types of the parse descriptors corresponding to selected DDC^α types. The translation reveals that all parse descriptors share a common structure, consisting of two components, a header and a body. The header reports on the corresponding representation as a whole. It stores the number of errors encountered during parsing, an error code indicating the degree of success of the parse—success, success with errors, or failure—and the span of data (location in the source) described by the descriptor. To be precise, the type of the header (`pd_hdr`) is $\texttt{int} * \texttt{errcode} * \texttt{span}$. The body contains parse descriptors for the subcomponents of the representation. For types without subcomponents, we use `unit` as the body type. As with the representation types, dependency is uniformaly erased.

Like other types, DDC^α type variables α are translated into a pair of header and a body. The body has abstract type $\alpha_{\texttt{PDb}}$. This translation makes it possible for polymorphic parsing code

$$\boxed{\llbracket \tau{:}\kappa \rrbracket_{\mathrm{PT}} = \sigma}$$

$$\begin{aligned}
\llbracket \tau{:}\mathsf{T} \rrbracket_{\mathrm{PT}} &= \texttt{bits} * \texttt{offset} \to \texttt{offset} * \llbracket \tau \rrbracket_{\mathrm{rep}} * \llbracket \tau \rrbracket_{\mathrm{PD}} \\
\llbracket \tau{:}\sigma \to \kappa \rrbracket_{\mathrm{PT}} &= \sigma \to \llbracket \tau\ e{:}\kappa \rrbracket_{\mathrm{PT}},\ \text{for any e.} \\
\llbracket \tau{:}\mathsf{T} \to \kappa \rrbracket_{\mathrm{PT}} &= \forall \alpha_{\texttt{rep}}.\forall \alpha_{\texttt{PDb}}.\llbracket \alpha{:}\mathsf{T} \rrbracket_{\mathrm{PT}} \to \llbracket \tau\alpha{:}\kappa \rrbracket_{\mathrm{PT}} \\
&\ (\alpha_{\texttt{rep}}, \alpha_{\texttt{PDb}} \notin \mathsf{FTV}(\kappa) \cup \mathsf{FTV}(\tau))
\end{aligned}$$

Figure 13. Host language types for parsing functions

$$\boxed{\llbracket \tau \rrbracket_{\mathrm{P}} = e}$$

$$\begin{array}{l}
\llbracket C(e) \rrbracket_{\mathrm{P}} = \lambda(\texttt{B},\omega).\mathcal{B}_{\mathrm{imp}}(C)\ (e)\ (\texttt{B},\omega) \\
\llbracket \lambda x.\tau \rrbracket_{\mathrm{P}} = \lambda x.\llbracket \tau \rrbracket_{\mathrm{P}} \\
\llbracket \tau\ e \rrbracket_{\mathrm{P}} = \llbracket \tau \rrbracket_{\mathrm{P}}\ e \\
\llbracket \Sigma\, x{:}\tau.\tau' \rrbracket_{\mathrm{P}} = \\
\quad \lambda(\texttt{B},\omega). \\
\qquad \texttt{let}\ (\omega',\texttt{r},\texttt{p}) = \llbracket \tau \rrbracket_{\mathrm{P}}\ (\texttt{B},\omega)\ \texttt{in} \\
\qquad \texttt{let x} = (\texttt{r},\texttt{p})\ \texttt{in} \\
\qquad \texttt{let}\ (\omega'',\texttt{r}',\texttt{p}') = \llbracket \tau' \rrbracket_{\mathrm{P}}\ (\texttt{B},\omega')\ \texttt{in} \\
\qquad (\omega'', \texttt{R}_\Sigma(\texttt{r},\texttt{r}'), \texttt{P}_\Sigma(\texttt{p},\texttt{p}')) \\
\llbracket \tau + \tau' \rrbracket_{\mathrm{P}} = \\
\quad \lambda(\texttt{B},\omega). \\
\qquad \texttt{let}\ (\omega',\texttt{r},\texttt{p}) = \llbracket \tau \rrbracket_{\mathrm{P}}\ (\texttt{B},\omega)\ \texttt{in} \\
\qquad \texttt{if isOk(p) then}\ (\omega', \texttt{R}_{+\texttt{left}}(\texttt{r}), \texttt{P}_{+\texttt{left}}(\texttt{p})) \\
\qquad \texttt{else let}\ (\omega',\texttt{r},\texttt{p}) = \llbracket \tau' \rrbracket_{\mathrm{P}}\ (\texttt{B},\omega)\ \texttt{in} \\
\qquad (\omega', \texttt{R}_{+\texttt{right}}(\texttt{r}), \texttt{P}_{+\texttt{right}}(\texttt{p})) \\
\llbracket \{x{:}\tau \mid e\} \rrbracket_{\mathrm{P}} = \\
\quad \lambda(\texttt{B},\omega). \\
\qquad \texttt{let}\ (\omega',\texttt{r},\texttt{p}) = \llbracket \tau \rrbracket_{\mathrm{P}}\ (\texttt{B},\omega)\ \texttt{in} \\
\qquad \texttt{let x} = (\texttt{r},\texttt{p})\ \texttt{in} \\
\qquad \texttt{let c} = e\ \texttt{in} \\
\qquad (\omega', \texttt{R}_{\texttt{con}}(\texttt{c},\texttt{r}), \texttt{P}_{\texttt{con}}(\texttt{c},\texttt{p})) \\
\llbracket \alpha \rrbracket_{\mathrm{P}} = \texttt{parse}_\alpha \\
\llbracket \mu\alpha.\tau \rrbracket_{\mathrm{P}} = \\
\quad \texttt{fun parse}_\alpha\ (\texttt{B}{:}\texttt{bits}, \omega{:}\texttt{offset}) : \texttt{offset} * \llbracket \mu\alpha.\tau \rrbracket_{\mathrm{rep}} * \llbracket \mu\alpha.\tau \rrbracket_{\mathrm{PD}} = \\
\qquad \texttt{let}\ (\omega',\texttt{r},\texttt{p}) = \llbracket \tau \rrbracket_{\mathrm{P}}[\llbracket \mu\alpha.\tau \rrbracket_{\mathrm{rep}}/\alpha_{\texttt{rep}}][\llbracket \mu\alpha.\tau \rrbracket_{\mathrm{PDb}}/\alpha_{\texttt{PDb}}]\ (\texttt{B},\omega)\ \texttt{in} \\
\qquad\quad (\omega', \texttt{fold}[\llbracket \mu\alpha.\tau \rrbracket_{\mathrm{rep}}]\,\texttt{r}, (\texttt{p.h}, \texttt{fold}[\llbracket \mu\alpha.\tau \rrbracket_{\mathrm{PDb}}]\,\texttt{p})) \\
\llbracket \lambda\alpha.\tau \rrbracket_{\mathrm{P}} = \Lambda\alpha_{\texttt{rep}}.\Lambda\alpha_{\texttt{PDb}}.\lambda\texttt{parse}_\alpha.\llbracket \tau \rrbracket_{\mathrm{P}} \\
\llbracket \tau_1 \tau_2 \rrbracket_{\mathrm{P}} = \llbracket \tau_1 \rrbracket_{\mathrm{P}}\ [\llbracket \tau_2 \rrbracket_{\mathrm{rep}}]\ [\llbracket \tau_2 \rrbracket_{\mathrm{PDb}}]\ \llbracket \tau_2 \rrbracket_{\mathrm{P}}
\end{array}$$

Figure 14. DDC$^\alpha$ parsing semantics, selected constructs

to examine the header of a PD, even though it does not know the DDC$^\alpha$ type it is parsing. DDC$^\alpha$ abstractions are translated into F_ω type constructors that abstract the body of the PD (as opposed to the entire PD) and DDC$^\alpha$ applications are translated into F_ω type applications where the argument type is the PD body type.

DDC$^\alpha$ *parsing semantics.* The parsing semantics of a type τ with kind T is a function that transforms some amount of input into a pair of a representation and a parse descriptor, the types of which are determined by τ. The parsing semantics for types with higher kind are functions that construct parsers, or functions that construct functions that construct parsers, *etc.* Figure 13 specifies the host-language types of the functions generated from well-kinded DDC$^\alpha$ types.

For each (unparameterized) type, the input to the corresponding parser is a bit string to parse and an offset at which to begin parsing. The output is a new offset, a representation of the parsed data, and a parse descriptor.

For any type, there are three steps to parsing: parse the subcomponents of the type (if any), assemble the resultant representation, and tabulate meta-data based on subcomponent meta-data (if any).

$$\boxed{\llbracket \tau{:}\kappa \rrbracket_{\mathrm{PPT}} = \sigma}$$

$$\begin{aligned}
\llbracket \tau{:}\mathsf{T} \rrbracket_{\mathrm{PPT}} &= \llbracket \tau \rrbracket_{\mathrm{rep}} * \llbracket \tau \rrbracket_{\mathrm{PD}} \to \texttt{bits} \\
\llbracket \tau{:}\sigma \to \kappa \rrbracket_{\mathrm{PPT}} &= \sigma \to \llbracket \tau\ e{:}\kappa \rrbracket_{\mathrm{PPT}},\ \text{for any e.} \\
\llbracket \tau{:}\mathsf{T} \to \kappa \rrbracket_{\mathrm{PPT}} &= \forall \alpha_{\texttt{rep}}.\forall \alpha_{\texttt{PDb}}.\llbracket \alpha{:}\mathsf{T} \rrbracket_{\mathrm{PPT}} \to \llbracket \tau\alpha{:}\kappa \rrbracket_{\mathrm{PPT}} \\
&\ (\alpha_{\texttt{rep}}, \alpha_{\texttt{PDb}} \notin \mathsf{FTV}(\kappa) \cup \mathsf{FTV}(\tau))
\end{aligned}$$

Figure 15. Host language types for printing functions

For the sake of clarity, we have factored the latter two steps into separate representation and PD constructor functions which we define for each type. For example, the representation and PD constructors for the dependent sums are $\texttt{R}_\Sigma$ and $\texttt{P}_\Sigma$, respectively. We have also factored out some commonly occuring code into auxiliary functions. These constructors and functions appear in Appendix D.

The PD constructors determine the error code and calculate the error count. There are three possible error codes: `ok`, `err`, and `fail`, corresponding to the three possible results of a parse: it can succeed, parsing the data without errors; it can succeed, but discover errors in the process; or, it can find an unrecoverable error and fail. The error count is determined by subcomponent error counts and any errors associated directly with the type itself.

Figure 14 specifies the parsing semantics of a selected portion of DDC$^\alpha$. We explain the interpretations of select types, from which the interpretation of the remaining types may be understood. The full semantics appears in Appendix B. A dependent sum parses the data according to the first type, binding the resulting representation and PD to x before parsing the remaining data according to the second type. It then bundles the results using the dependent sum constructor functions.

A type variable translates to an expression variable whose name corresponds directly to the name of the type variable. These expression variables are bound in the interpretations of recursive types and type abstractions. We interpret each recursive type as a recursive function whose name corresponds to the name of the recursive type variable. For clarity, we annotate the recursive function with its type.

We interpret type abstraction as a function over other parsing functions. Because those parsing functions can have arbitrary DDC$^\alpha$ types (of kind T), the interpretation must be a polymorphic function, parameterized by the representation and PD-body type of the DDC$^\alpha$ type parameter. For clarity, we present this type parameterization explicitly. Type application $\tau_1\,\tau_2$ simply becomes the application of the interpretation of τ_1 to the representation-type, PD-type, and parsing interpretations of τ_2.

DDC$^\alpha$ *printing semantics* The definition of the printing semantics for a DDC$^\alpha$ description uses a similar set of concepts as the parsing semantics. To begin, the semantic function $\llbracket \tau{:}\kappa \rrbracket_{\mathrm{PPT}} = \sigma$ gives the host language type σ for the printer generated from type τ with kind κ. As shown in Figure 15, the printing semantics for descriptions with higher kind are functions that construct printers, while the printing semantics for descriptions with base kind are simple first-order functions that map a representation and a parse descriptor into a string of bits.

Figure 16 presents the printing semantics of selected DDC$^\alpha$ constructs. Base types $C(e)$ are printed in various ways according to the definition $\mathcal{B}_{\mathrm{pp}}$, which is a parameter to the semantics. The base type printer $\mathcal{B}_{\mathrm{pp}}$ accepts the parse descriptor as a parameter, and in the case of an error, prints nothing. Dependent sums print one component and then the next in order. An ordinary sum prints the underlying tagged value. Notice that the structure of the parse descriptor and the representation should be isomorphic – both should

$\boxed{[\![\tau]\!]_{\mathrm{PP}} = e}$

$$
\begin{array}{l}
[\![C(e)]\!]_{\mathrm{PP}} = \lambda(\mathtt{r},\mathtt{pd}).\mathcal{B}_{\mathrm{pp}}(C)\ (e)\ (\mathtt{r},\mathtt{pd}) \\
[\![\lambda x.\tau]\!]_{\mathrm{PP}} = \lambda x.[\![\tau]\!]_{\mathrm{PP}} \\
[\![\tau\, e]\!]_{\mathrm{PP}} = [\![\tau]\!]_{\mathrm{PP}}\ e \\
[\![\Sigma\, x{:}\tau_1.\tau_2]\!]_{\mathrm{PP}} = \\
\quad \lambda(\mathtt{r},\mathtt{pd}). \\
\qquad \mathtt{let\ x\ =}\ (\mathtt{r.1},\mathtt{pd.2.1})\ \mathtt{in} \\
\qquad \mathtt{let\ bs_1\ =}\ [\![\tau_1]\!]_{\mathrm{PP}}\ \mathtt{x\ in} \\
\qquad \mathtt{let\ bs_2\ =}\ [\![\tau_2]\!]_{\mathrm{PP}}\ (\mathtt{r.2},\mathtt{pd.2.2})\ \mathtt{in} \\
\qquad \mathtt{bs_1}\ @\ \mathtt{bs_2} \\
[\![\tau_1 + \tau_2]\!]_{\mathrm{PP}} = \\
\quad \lambda(\mathtt{r},\mathtt{pd}). \\
\qquad \mathtt{case}\ (\mathtt{r},\mathtt{pd.2})\ \mathtt{of} \\
\qquad \mid (\mathtt{inl\ r_1},\mathtt{inl\ p_1}) \Rightarrow [\![\tau_1]\!]_{\mathrm{PP}}\ (\mathtt{r_1},\mathtt{p_1}) \\
\qquad \mid (\mathtt{inr\ r_2},\mathtt{inr\ p_2}) \Rightarrow [\![\tau_2]\!]_{\mathrm{PP}}\ (\mathtt{r_2},\mathtt{p_2}) \\
\qquad \mid \_ \Rightarrow \mathtt{badInput}() \\
[\![\{x{:}\tau \mid e\}]\!]_{\mathrm{PP}} = \\
\quad \lambda(\mathtt{r},\mathtt{pd}). \\
\qquad \mathtt{case}\ (\mathtt{r},\mathtt{pd.2})\ \mathtt{of} \\
\qquad \mid (\mathtt{inl\ r_1},\mathtt{p_1}) \Rightarrow [\![\tau]\!]_{\mathrm{PP}}\ (\mathtt{r_1},\mathtt{p_1}) \\
\qquad \mid (\mathtt{inr\ r_2},\mathtt{p_2}) \Rightarrow [\![\tau]\!]_{\mathrm{PP}}\ (\mathtt{r_2},\mathtt{p_2}) \\
[\![\alpha]\!]_{\mathrm{PP}} = \mathtt{print}_\alpha \\
[\![\mu\alpha.\tau]\!]_{\mathrm{PP}} = \\
\quad \mathtt{fun\ print}_\alpha\ (\mathtt{r} : [\![\mu\alpha.\tau]\!]_{\mathrm{rep}}, \mathtt{pd} : [\![\mu\alpha.\tau]\!]_{\mathrm{PD}}) : \mathtt{bits\ =} \\
\qquad [\![\tau]\!]_{\mathrm{PP}}[[\![\mu\alpha.\tau]\!]_{\mathrm{rep}}/\alpha_{\mathtt{rep}}][[\![\mu\alpha.\tau]\!]_{\mathrm{PDb}}/\alpha_{\mathtt{PDb}}] \\
\qquad\quad (\mathtt{unfold}[[\![\mu\alpha.\tau]\!]_{\mathrm{rep}}]\ \mathtt{r}, \mathtt{unfold}[[\![\mu\alpha.\tau]\!]_{\mathrm{PDb}}]\ \mathtt{pd.2}) \\
[\![\lambda\alpha.\tau]\!]_{\mathrm{PP}} = \Lambda\alpha_{\mathtt{rep}}.\Lambda\alpha_{\mathtt{PDb}}.\lambda\mathtt{print}_\alpha.[\![\tau]\!]_{\mathrm{PP}} \\
[\![\tau_1\tau_2]\!]_{\mathrm{PP}} = [\![\tau_1]\!]_{\mathrm{PP}}\ [[\![\tau_2]\!]_{\mathrm{rep}}]\ [[\![\tau_2]\!]_{\mathrm{PDb}}]\ [\![\tau_2]\!]_{\mathrm{PP}}
\end{array}
$$

Figure 16. DDC$^\alpha$ printing semantics, selected constructs

be left injections or both should be right injections. Any pair of structures generated by the parser are guaranteed to satisfy this invariant. If the pair do not match, then the programmer is using the printer incorrectly. In this case, the printer calls an unspecified error routine named `badInput()`.

The semantics of printing recursive and parameterized types follows similar lines to the semantics of parsing these constructs. In particular, whenever a type parameter is introduced in the syntax of a description, a corresponding value parameter with printer function type is introduced in the generated printer code. We give the value parameter the name $print_\alpha$. Both type abstractions and recursive functions introduce such parameters. Notice that whereas the parsing semantics uses a fold to build a recursive data structure when interpreting a recursive type, the printing semantics uses an unfold to deconstruct a recursive data structure for printing.

5.4 Meta-theory

To validate our semantic definitions, we have proven two key metatheoretic results. First, we show that parsers and printers are *type-correct*, always returning representations and parse descriptors of the appropriate type. Second, we give a precise characterization of the results of parsers (and input requirements of printers) by defining the *canonical forms* of representation-parse descriptor pairs associated with a dependent DDC$^\alpha$ type.

Type Correctness. Demonstrating that generated parsers and printers are well formed and have the expected types is nontrivial primarily because the generated code expects parse descriptors to have a particular shape, and it is not completely obvious they do in the presence of polymorphism. Hence, to prove type correctness, we first need to characterize the shape of parse descriptors for arbitrary DDC$^\alpha$ types. Unfortunately, the most straightforward characterization is too weak to prove directly, and hence Definition 1 specifies a much stronger property as a logical relation. Lemma 2 establishes that the logical relation holds of all well-formed DDC$^\alpha$ types by induction on kinding derivations, and the desired characterization follows as a corollary.

Definition 1
- $\mathrm{H}(\tau : \mathsf{T})$ *iff* $\exists\,\sigma$ *s.t.* $[\![\tau]\!]_{PD} \equiv \mathtt{pd\_hdr} * \sigma$.
- $\mathrm{H}(\tau : \mathsf{T} \to \kappa)$ *iff* $\exists\,\sigma$ *s.t.* $[\![\tau]\!]_{PD} \equiv \sigma$ *and whenever* $\mathrm{H}(\tau' : \mathsf{T})$, *we have* $\mathrm{H}(\tau\,\tau' : \kappa)$.
- $\mathrm{H}(\tau : \sigma \to \kappa)$ *iff* $\exists\,\sigma'$ *s.t.* $[\![\tau]\!]_{PD} \equiv \sigma'$ *and* $\mathrm{H}(\tau\, e : \kappa)$ *for any expression* e.

Lemma 2
If $\Delta;\Gamma \vdash \tau : \kappa$ *then* $\mathrm{H}(\tau : \kappa)$.

Lemma 3
- *If* $\Delta;\Gamma \vdash \tau : \kappa$ *then* $\exists\sigma.[\![\tau]\!]_{PD} = \sigma$.
- *If* $\Delta;\Gamma \vdash \tau : \mathsf{T}$ *then* $\exists\sigma.[\![\tau]\!]_{PD} \equiv \mathtt{pd\_hdr} * \sigma$.

With this lemma, we can establish the type correctness of the generated parsers and printers. We prove the theorem using a more general induction hypothesis that applies to open types. This hypothesis must account for the fact that any free type variables in a DDC$^\alpha$ type τ will become free function variables in $[\![\tau]\!]_{\mathrm{P}}$. To that end, we define the functions $[\![\Delta]\!]_{\mathrm{PT}}$ and $[\![\Delta]\!]_{\mathrm{PPT}}$ which map type-variable contexts Δ in the DDC$^\alpha$ to value-variable contexts Γ in F_ω. In addition, the function $\|\Delta\|$ generates the appropriate F_ω type-variable context from the DDC$^\alpha$ context Δ.

$$
\begin{array}{ll}
\|\cdot\| = \cdot & \|\Delta,\alpha{:}\mathsf{T}\| = \|\Delta\|, \alpha_{\mathtt{rep}}{:}\mathsf{T}, \alpha_{\mathtt{PDb}}{:}\mathsf{T} \\
[\![\,\cdot\,]\!]_{\mathrm{PT}} = \cdot & [\![\Delta,\alpha{:}\mathsf{T}]\!]_{\mathrm{PT}} = [\![\Delta]\!]_{\mathrm{PT}}, \mathtt{parse}_\alpha{:}[\![\alpha{:}\mathsf{T}]\!]_{\mathrm{PT}} \\
[\![\,\cdot\,]\!]_{\mathrm{PPT}} = \cdot & [\![\Delta,\alpha{:}\mathsf{T}]\!]_{\mathrm{PPT}} = [\![\Delta]\!]_{\mathrm{PPT}}, \mathtt{print}_\alpha{:}[\![\alpha{:}\mathsf{T}]\!]_{\mathrm{PPT}}
\end{array}
$$

Lemma 4 (Type Correctness Lemma)
- *If* $\Delta;\Gamma \vdash \tau : \kappa$ *then* $\|\Delta\|, \Gamma, [\![\Delta]\!]_{PT} \vdash [\![\tau]\!]_{P} : [\![\tau{:}\kappa]\!]_{PT}$
- *If* $\Delta;\Gamma \vdash \tau : \kappa$ *then* $\|\Delta\|, \Gamma, [\![\Delta]\!]_{PPT} \vdash [\![\tau]\!]_{PP} : [\![\tau{:}\kappa]\!]_{PPT}$.

PROOF. By induction on the height of the kinding derivation. □

Theorem 5 (Type Correctness of Closed Types)
- *If* $\vdash \tau : \kappa$ *then* $\vdash [\![\tau]\!]_{P} : [\![\tau{:}\kappa]\!]_{PT}$.
- *If* $\vdash \tau : \kappa$ *then* $\vdash [\![\tau]\!]_{PP} : [\![\tau{:}\kappa]\!]_{PPT}$.

A practical implication of this theorem is that it is sufficient to check data descriptions (*i.e.* DDC$^\alpha$ types) for well-formedness to ensure that the generated types and functions are well formed. This property is sorely lacking in many common implementations of Lex and YACC, for which users must examine generated code to debug compile-time errors in specifications.

Canonical Forms for Parsed Data. DDC$^\alpha$ parsers generate pairs of representations and parse descriptors designed to satisfy a number of invariants. Of greatest importance is the fact that when the parse descriptor says there are no errors in a particular substructure, the programmer can count on the representation satisfying all of the syntactic and semantic constraints expressed by the DDC$^\alpha$ type description. When a parse descriptor and representation satisfy these invariants, we say the pair of data structures is in *canonical form*. While generated parsers produce canonical outputs, generated printers expect canonical inputs.

For each DDC$^\alpha$ type, its canonical forms are defined via two (mutually recursive) relations. The first relation, $\mathrm{Canon}_\nu(r,p)$, defines the canonical form of a representation r and a parse descriptor p at normal type ν. *Normal types* are those closed types with base kind T that are defined in Figure 17. Types with higher kind

$$\begin{array}{llll} \text{Normal Types} & \nu & ::= & C(e) \mid \lambda x.\tau \mid \Sigma\, x{:}\tau.\tau \mid \tau + \tau \\ & & \mid & \{x{:}\tau \mid e\} \mid \mu\alpha.\tau \mid \lambda\alpha.\tau \\ \text{Types} & \tau & ::= & \nu \mid \tau\, e \mid \tau\,\tau \mid \alpha \end{array}$$

Normalization:

$$\frac{\tau \to \tau'}{\tau\, e \to \tau'\, e} \quad \frac{e \to e'}{\nu\, e \to \nu\, e'} \quad \frac{}{(\lambda x.\tau)\, v \to \tau[v/x]}$$

$$\frac{\tau_1 \to \tau_1'}{\tau_1\, \tau_2 \to \tau_1'\, \tau_2} \quad \frac{\tau \to \tau'}{\nu\, \tau \to \nu\, \tau'} \quad \frac{}{(\lambda\alpha.\tau)\, \nu \to \tau[\nu/\alpha]}$$

Figure 17. DDC$^\alpha$ Normal Types, selected constructs

such as abstractions are not described by this relation as they cannot directly produce representations and PDs. The second definition, $\text{Canon}^*{}_\tau(r,p)$, normalizes τ, thereby eliminating outermost type and value applications. The result is a normal type ν and the requirements on ν are subsequently given by $\text{Canon}_\nu(r,p)$. For brevity in these definitions, we write $p.h.nerr$ as $p.nerr$ and use `pos` to denote the function that returns zero when passed zero and one when passed another natural number. The following definition gives the notion of F_ω expression equivalence we use.

Definition 6 (Canonical Forms (selected constructs))
(1) $\text{Canon}_\nu(r,p)$ *iff exactly one of the following is true:*

- $\nu = C(e)$ *and* $r = \texttt{inl}\ c$ *and* $p.nerr = 0$.
- $\nu = C(e)$ *and* $r = \texttt{inr none}$ *and* $p.nerr = 1$.
- $\nu = \Sigma\, x{:}\tau_1.\tau_2$ *and* $r = (r_1, r_2)$ *and* $p = (h, (p_1, p_2))$ *and* $h.nerr = \texttt{pos}(p_1.nerr) + \texttt{pos}(p_2.nerr)$, $\text{Canon}^*{}_{\tau_1}(r_1, p_1)$ *and* $\text{Canon}^*{}_{\tau_2[(r,p)/x]}(r_2, p_2)$.
- $\nu = \tau_1 + \tau_2$ *and* $r = \texttt{inl}\ r'$ *and* $p = (h, \texttt{inl}\ p')$ *and* $h.nerr = \texttt{pos}(p'.nerr)$ *and* $\text{Canon}^*{}_{\tau_1}(r', p')$.
- $\nu = \tau_1 + \tau_2$ *and* $r = \texttt{inr}\ r'$ *and* $p = (h, \texttt{inr}\ p')$ *and* $h.nerr = \texttt{pos}(p'.nerr)$ *and* $\text{Canon}^*{}_{\tau_2}(r', p')$.
- $\nu = \{x{:}\tau' \mid e\}$, $r = \texttt{inl}\ r'$ *and* $p = (h, p')$, *and* $h.nerr = \texttt{pos}(p'.nerr)$, $\text{Canon}^*{}_{\tau'}(r', p')$ *and* $e[(r',p')/x] \to^* \texttt{true}$.
- $\nu = \{x{:}\tau' \mid e\}$, $r = \texttt{inr}\ r'$ *and* $p = (h, p')$, *and* $h.nerr = 1 + \texttt{pos}(p'.nerr)$, $\text{Canon}^*{}_{\tau'}(r', p')$ *and* $e[(r',p')/x] \to^* \texttt{false}$.
- $\nu = \mu\alpha.\tau'$, $r = \texttt{fold}[[\![\mu\alpha.\tau']\!]_{rep}]\, r'$, $p = (h, \texttt{fold}[[\![\mu\alpha.\tau']\!]_{PD}]\, p')$, $p.nerr = p'.nerr$ *and* $\text{Canon}^*{}_{\tau'[\mu\alpha.\tau'/\alpha]}(r', p')$.

(2) $\text{Canon}^*{}_\tau(r,p)$ *iff* $\tau \to^* \nu$ *and* $\text{Canon}_\nu(r,p)$.

The first part of Lemma 7 states that parsers for well-formed types (of base kind) produce a canonical pair of representation and parse descriptor if they produce anything at all. Conversely, the second part states that, given a canonical representation and parse descriptor, the printer for well-formed types (of base kind) will not "go wrong" by calling the `badInput()` function.

Theorem 7 (Parsing to/Printing from Canonical Forms)
- *If* $\vdash \tau : \mathsf{T}$ *and* $[\![\tau]\!]_P\ (B, \omega) \to^* (\omega', r, p)$ *then* $\text{Canon}^*{}_\tau(r,p)$.
- *If* $\vdash \tau : \mathsf{T}$, $\text{Canon}^*{}_\tau(r,p)$ *and* $[\![\tau]\!]_{PP}\ (r,p) \to^* \texttt{e}$ *then* $\texttt{e} \neq \texttt{badInput}()$.

PROOF. Both items are proven by induction on the length of the respective F_ω evaluation relations. Within the induction they proceed by a case analysis on the structure of the type τ. □

6. Related Work

Many useful tools exist to help programmers generate parsers. Examples include compiler technology such as the many variants of LEX and YACC as well as interpreter technology such the parser combinator libraries found in functional programming languages (Haskell [18], for example). Likewise, there are tools to help programmers generate printers. Each of these technologies is very useful in its own domain, but PADS/ML is broader in its scope than each of them: a single PADS/ML description is sufficient to generate *both* a parser and a printer. And a statistical error analysis, a format debugger, an XML translator, and in the future, a query engine [10], a content-based search engine [21, 26], more statistical analyses, *etc.* Neither combinator libraries nor tools such as LEX and YACC are designed to generate such a range of artifacts from a single specification. Indeed, the proper way to think about combinator libraries in relation to PADS/ML is that they might serve as an alternative implementation strategy for some of the generated tools.

Generic programming [19, 16, 20] and design patterns such as the visitor pattern can facilitate the implementation of type-directed data structure traversals. Lammel and Peyton Jones' original "scrap your boilerplate" article [20] provides a detailed summary of the trade-offs between different techniques. We investigated using these techniques in our system; however, we found that most of them required language features such as type classes that are available only in Haskell. The generated PADS/ML traversal functors are less flexible than those possible in Haskell, but they suffice for many tools useful in practice.

The networking community has developed a number of domain-specific languages, including DataScript [2], PacketTypes [23], and Bro's packet processing language [27] for parsing and printing binary data. Like PADS/ML, these languages use a type-directed approach to describe ad hoc data and permit the user to define semantic constraints. In contrast to our work, these systems handle only binary data and assume the data is error-free. DFDL is a specification of a data format description language with an XML-based syntax and type structure [7, 3]. DFDL is still under development. It does not have a formal semantics, or a tool generation architecture. We believe that DFDL is similar in its expressiveness to PADS/C. However, because the specification is evolving, we cannot give a more detailed comparison.

There are a number of tools designed to convert ad hoc data formats into XML, including XSugar [5] and the Binary Format Description language (BFD) [24]. The scope of both of these projects is limited to conversion to-and-from XML; neither is of any use for analysts who do not wish to convert their data to XML (and there are compelling reasons why an analyst might not wish to make such a conversion). PADS/ML is thus broader in scope: it *can* convert data into XML, but it can do many other tasks as well.

Similarly, commercial database products provide some support for parsing data in external formats so the data can be imported into their systems, but they typically support only a limited number of formats and have fixed methods for coping with erroneous data. As with the XML systems, these tools are of no help in any task besides loading data into a database.

A complementary class of languages includes ASN.1 [9] and ASDL [1]. Both of these systems specify the *logical* in-memory representation of data and then automatically generate a *physical* on-disk representation. While very useful, this technology does not help with data that arrives in predetermined, ad hoc formats.

On the theoretical front, the scientific community's understanding of type-based languages for data description is much less mature. To the best of our knowledge, our previous work on the DDC [12] was the first to provide a formal interpretation of dependent types as parsers and to study the properties of these parsers including error correctness and type safety. The current paper extends and improves our earlier work by simplifying the basic theory in a number of subtle but important ways, by adding polymorphic types for the purpose of code reuse, and by specifying the semantics of printing.

Regular expressions and context-free grammars, the basis for LEX and YACC, have been well-studied, but they do not have dependency, a key feature necessary for expressing constraints and parsing ad hoc data. *Parsing Expression Grammars* (PEGs), studied in the early seventies [4], revitalized by Ford [14], and implemented using "packrat parsing" techniques [13, 15], are more closely related to PADS/ML's recursive descent parsers. However, the multiple interpretations of types in the DDC$^{\alpha}$ makes our theory substantially different from the theory of PEGs.

7. Conclusions

Vast quantities of important information exist only in ad hoc formats. Data analysts desperately need reliable, high-level tools to help them document, parse, analyze, transform, query, and visualize such data. PADS/ML is a high-level domain-specific language and system designed for this purpose. Inspired by the type structure of functional programming languages, PADS/ML uses dependent polymorphic 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 cosmology. 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. To give PADS/ML a precise semantics, we have simplified and extended the Data Description Calculus (DDC$^{\alpha}$) [12] to account for parametric polymorphism and to provide a semantics for printing.

Acknowledgments

We would like to thank Derek Dreyer for discussions and advice on advanced module systems.

References

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

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

[3] M. Beckerle and M. Westhead. GGF DFDL primer. `http://www.ggf.org/Meetings/GGF11/Documents/DFDL_Primer_v2.pdf`, May 2004. Global Grid Forum.

[4] A. Birman and J. D. Ullman. Parsing algorithms with backtrack. *Information and Control*, 23(1), Aug. 1973.

[5] C. Brabrand, A. Møller, and M. I. Schwartzbach. Dual syntax for XML languages. In *Tenth International Symposium on Database Programming Languages*, volume 3774 of *Lecture Notes in Computer Science*, pages 27–41. Springer-Verlag, August 2005.

[6] G. O. Consortium. Gene ontology project. http://www.geneontology.org.

[7] Data format description language (DFDL) a Proposal, Working Draft, Global Grid Forum. `https://forge.gridforum.org/projects/dfdl-wg/document/DFDL_Proposal/en/%2`, Aug 2005. Global Grid Forum.

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

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

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

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

[12] 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.

[13] B. Ford. Packrat parsing: Simple, powerful, lazy, linear time. In *ACM International Conference on Functional Programming*, pages 36–47. ACM Press, Oct. 2002.

[14] B. Ford. Parsing expression grammars: A recognition-based syntactic foundation. In *ACM Symposium on Principles of Programming Languages*, pages 111–122. ACM Press, Jan. 2004.

[15] R. Grimm. Practical packrat parsing. Technical Report TR2004-854, New York University, Mar. 2004.

[16] R. Hinze. A new approach to generic functional programming. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 119–132, Jan. 2000.

[17] R. Hinze and J. Jeuring. Generic Haskell: Practice and theory. Technical Report UU-CS-2003-015, Institute of Information and Computing Sciences, Utrecht University, 2003.

[18] G. Hutton and E. Meijer. Monadic parsing in Haskell. *Journal of Functional Programming*, 8(4):437–444, July 1998.

[19] J. Jeuring and P. Jansson. Polytypic programming. In *Second International School on Advanced Functional Programming*, volume 1129 of *Lecture Notes in Computer Science*, pages 68–114, Aug. 1996.

[20] R. Lämmel and S. Peyton Jones. Scrap your boilerplate: A practical design pattern for generic programming. *ACM SIGPLAN Notices*, 38(3):26–37, Mar. 2003. Proceedings of the ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI 2003).

[21] Q. Lv, W. Josephson, Z. Wang, M. Chrikar, and K. Li. Ferret: A toolkit for content-based similarity search of feature-rich data. In *EuroSys2006*, Apr. 2006.

[22] 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, Aug. 2005.

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

[24] J. Myers and A. Chappell. Binary format definition (BFD). `http://collaboratory.emsl.pnl.gov/sam/bfd/`, 2000.

[25] Tree formats. Workshop on molecular evolution. `http://workshop.molecularevolution.org/resources/fileformats/tree_forma%ts.php`.

[26] J. Oh. PADS and CASS utilization for beta coefficient estimation with the single-index model. Princeton University Undergraduate Senior Independent Work, May 2006.

[27] V. Paxson. A system for detecting network intruders in real-time. In *Computer Networks*, Dec. 1999.

[28] B. C. Pierce. *Types and Programming Languages*. The MIT Press, Feb. 2002.

A. Regulus Data Description in PADS/C

```
/* Pstring terminated by ';' or '|' */
Ptypedef Pstring_SE(:"/;|\\|/":) SVString;

Pstruct Nvp_string(:char * s:){
  s; "="; SVString val;
};

Pstruct Nvp_ip(:char * s:){
  s; "="; Pip val;
};

Pstruct Nvp_timestamp(:char * s:){
  s; "="; Ptstamp val;
};

Pstruct Nvp_Puint32(:char * s:){
  s; "="; Puint32 val;
};

Pstruct Nvp_a{
      Pstring(:'=':) name;
 '='; SVString       val;
};

Pstruct Details{
      Nvp_ip(:"src_addr":) source;
';';  Nvp_ip(:"dst_addr":) dest;
';';  Nvp_timestamp(:"start_time":) start_time;
';';  Nvp_timestamp(:"end_time":)   end_time;
';';  Nvp_Puint32(:"cycle_time":)   cycle_time;
};

Parray Nvp_seq{
  Nvp_a [] : Psep(';') && Pterm('|');
};

Punion Info(:int alarm_code:){
  Pswitch (alarm_code){
    Pcase 5074: Details   details;
    Pdefault:   Nvp_seq   generic;
  }
};

Penum Service {
   DOMESTIC,
   INTERNATIONAL,
   SPECIAL
};

Pstruct Raw_alarm {
       Puint32 alarm : alarm == 2 || alarm == 3;
 ':';  Popt Ptstamp start;
 '|';  Popt Ptstamp clear;
 '|';  Puint32      code;
 '|';  Nvp_string(:"dns1":) src_dns;
 ';';  Nvp_string(:"dns2":) dest_dns;
 '|';  Info(:code:) info;
 '|';  Service      service;
};

int chkCorr(Raw_alarm ra) { ...};

Precord Ptypedef Raw_alarm Alarm :
         Alarm a => {chkCorr(a)};

Psource Parray Source {
  Alarm[];
};
```

B. Complete Syntax and Semantics of DDC$^\alpha$

We first define the syntax of DDC$^\alpha$ terms:

$$
\begin{array}{llll}
\text{Kinds} & \kappa & ::= & \mathsf{T} \mid \sigma \to \kappa \mid \mathsf{T} \to \kappa \\
\text{Types} & \tau & ::= & \mathtt{unit} \mid \mathtt{bottom} \mid C(e) \mid \lambda x.\tau \mid \tau\, e \\
 & & \mid & \Sigma\, x{:}\tau.\tau \mid \tau + \tau \mid \tau \,\&\, \tau \mid \{x{:}\tau \mid e\} \mid \tau\,\mathtt{seq}(\tau, e, \tau) \\
 & & \mid & \alpha \mid \mu\alpha.\tau \mid \lambda\alpha.\tau \mid \tau\,\tau \\
 & & \mid & \mathtt{compute}(e{:}\sigma) \mid \mathtt{absorb}(\tau) \mid \mathtt{scan}(\tau)
\end{array}
$$

Figure 18 gives the complete kinding rules for the system.
The representation for each DDC$^\alpha$ type τ are defined as follows:

$$\boxed{[\![\tau]\!]_{\text{rep}} = \sigma}$$

$$
\begin{array}{lcl}
[\![\mathtt{unit}]\!]_{\text{rep}} & = & \mathtt{unit} \\
[\![\mathtt{bottom}]\!]_{\text{rep}} & = & \mathtt{none} \\
[\![C(e)]\!]_{\text{rep}} & = & \mathcal{B}_{\text{type}}(C) + \mathtt{none} \\
[\![\lambda x.\tau]\!]_{\text{rep}} & = & [\![\tau]\!]_{\text{rep}} \\
[\![\tau\, e]\!]_{\text{rep}} & = & [\![\tau]\!]_{\text{rep}} \\
[\![\Sigma\, x{:}\tau_1.\tau_2]\!]_{\text{rep}} & = & [\![\tau_1]\!]_{\text{rep}} * [\![\tau_2]\!]_{\text{rep}} \\
[\![\tau_1 + \tau_2]\!]_{\text{rep}} & = & [\![\tau_1]\!]_{\text{rep}} + [\![\tau_2]\!]_{\text{rep}} \\
[\![\tau_1 \,\&\, \tau_2]\!]_{\text{rep}} & = & [\![\tau_1]\!]_{\text{rep}} * [\![\tau_2]\!]_{\text{rep}} \\
[\![\{x{:}\tau \mid e\}]\!]_{\text{rep}} & = & [\![\tau]\!]_{\text{rep}} + [\![\tau]\!]_{\text{rep}} \\
[\![\tau\,\mathtt{seq}(\tau_{\text{sep}}, e, \tau_{\text{term}})]\!]_{\text{rep}} & = & \mathtt{int} * ([\![\tau]\!]_{\text{rep}}\,\mathtt{seq}) \\
[\![\alpha]\!]_{\text{rep}} & = & \alpha_{\mathtt{rep}} \\
[\![\mu\alpha.\tau]\!]_{\text{rep}} & = & \mu\alpha_{\mathtt{rep}}.[\![\tau]\!]_{\text{rep}} \\
[\![\lambda\alpha.\tau]\!]_{\text{rep}} & = & \lambda\alpha_{\mathtt{rep}}.[\![\tau]\!]_{\text{rep}} \\
[\![\tau_1\tau_2]\!]_{\text{rep}} & = & [\![\tau_1]\!]_{\text{rep}}[\![\tau_2]\!]_{\text{rep}} \\
[\![\mathtt{compute}(e{:}\sigma)]\!]_{\text{rep}} & = & \sigma \\
[\![\mathtt{absorb}(\tau)]\!]_{\text{rep}} & = & \mathtt{unit} + \mathtt{none} \\
[\![\mathtt{scan}(\tau)]\!]_{\text{rep}} & = & [\![\tau]\!]_{\text{rep}} + \mathtt{none}
\end{array}
$$

The parse descriptor for each DDC$^\alpha$ type τ are defined as follows:

$$\boxed{[\![\tau]\!]_{\text{PD}} = \sigma}$$

$$
\begin{array}{lcl}
[\![\mathtt{unit}]\!]_{\text{PD}} & = & \mathtt{pd\_hdr} * \mathtt{unit} \\
[\![\mathtt{bottom}]\!]_{\text{PD}} & = & \mathtt{pd\_hdr} * \mathtt{unit} \\
[\![C(e)]\!]_{\text{PD}} & = & \mathtt{pd\_hdr} * \mathtt{unit} \\
[\![\lambda x.\tau]\!]_{\text{PD}} & = & [\![\tau]\!]_{\text{PD}} \\
[\![\tau\, e]\!]_{\text{PD}} & = & [\![\tau]\!]_{\text{PD}} \\
[\![\Sigma\, x{:}\tau_1.\tau_2]\!]_{\text{PD}} & = & \mathtt{pd\_hdr} * [\![\tau_1]\!]_{\text{PD}} * [\![\tau_2]\!]_{\text{PD}} \\
[\![\tau_1 + \tau_2]\!]_{\text{PD}} & = & \mathtt{pd\_hdr} * ([\![\tau_1]\!]_{\text{PD}} + [\![\tau_2]\!]_{\text{PD}}) \\
[\![\tau_1 \,\&\, \tau_2]\!]_{\text{PD}} & = & \mathtt{pd\_hdr} * [\![\tau_1]\!]_{\text{PD}} * [\![\tau_2]\!]_{\text{PD}} \\
[\![\{x{:}\tau \mid e\}]\!]_{\text{PD}} & = & \mathtt{pd\_hdr} * [\![\tau]\!]_{\text{PD}} \\
[\![\tau\,\mathtt{seq}(\tau_{\text{sep}}, e, \tau_{\text{term}})]\!]_{\text{PD}} & = & \mathtt{pd\_hdr} * ([\![\tau]\!]_{\text{PD}}\,\mathtt{arr\_pd}) \\
[\![\alpha]\!]_{\text{PD}} & = & \mathtt{pd\_hdr} * \alpha_{\mathtt{PDb}} \\
[\![\mu\alpha.\tau]\!]_{\text{PD}} & = & \mathtt{pd\_hdr} * \mu\alpha_{\mathtt{PDb}}.[\![\tau]\!]_{\text{PD}} \\
[\![\lambda\alpha.\tau]\!]_{\text{PD}} & = & \lambda\alpha_{\mathtt{PDb}}.[\![\tau]\!]_{\text{PD}} \\
[\![\tau_1\tau_2]\!]_{\text{PD}} & = & [\![\tau_1]\!]_{\text{PD}}[\![\tau_2]\!]_{\text{PDb}} \\
[\![\mathtt{compute}(e{:}\sigma)]\!]_{\text{PD}} & = & \mathtt{pd\_hdr} * \mathtt{unit} \\
[\![\mathtt{absorb}(\tau)]\!]_{\text{PD}} & = & \mathtt{pd\_hdr} * \mathtt{unit} \\
[\![\mathtt{scan}(\tau)]\!]_{\text{PD}} & = & \mathtt{pd\_hdr} * ((\mathtt{int} * \mathtt{bits} * [\![\tau]\!]_{\text{PD}}) + \mathtt{unit})
\end{array}
$$

$$\boxed{[\![\tau]\!]_{\text{PDb}} = \sigma}$$

$$[\![\tau]\!]_{\text{PDb}} \quad = \quad \sigma \;\text{ where } [\![\tau]\!]_{\text{PD}} \equiv \mathtt{pd\_hdr} * \sigma$$

Figure 19 gives the parsing semantics for DDC$^\alpha$ type τ.

The type correctness theorem relies on base type parsers behaving properly. The following conditions make explicit the properties that base type semantic functions must satisfy.

Condition 8 (Conditions on Base-type Interfaces)
1. $\mathsf{dom}(\mathcal{B}_{\mathit{kind}}) = \mathsf{dom}(\mathcal{B}_{\mathit{imp}})$.

$\boxed{\|\Delta\| = \Delta}$

$$\|\cdot\| = \cdot \qquad\qquad \|\Delta, \alpha{:}\mathsf{T}\| = \|\Delta\|, \alpha_{\mathtt{rep}}{:}\mathsf{T}, \alpha_{\mathtt{PDb}}{:}\mathsf{T}$$

$\boxed{\Delta;\Gamma \vdash \tau : \kappa}$

$$\frac{\vdash \|\Delta\|, \Gamma\ \mathsf{ok}}{\Delta;\Gamma \vdash \mathtt{unit} : \mathsf{T}}\ \text{Unit} \qquad \frac{\vdash \|\Delta\|, \Gamma\ \mathsf{ok}}{\Delta;\Gamma \vdash \mathtt{bottom} : \mathsf{T}}\ \text{Bottom} \qquad \frac{\begin{array}{cc}\vdash \|\Delta\|, \Gamma\ \mathsf{ok} & \|\Delta\|, \Gamma \vdash e : \sigma \\ \mathcal{B}_{\text{kind}}(C) = \sigma \to \mathsf{T} & \end{array}}{\Delta;\Gamma \vdash C(e) : \mathsf{T}}\ \text{Const}$$

$$\frac{\Delta;\Gamma, x{:}\sigma \vdash \tau : \kappa}{\Delta;\Gamma \vdash \lambda x.\tau : \sigma \to \kappa}\ \text{Abs} \qquad \frac{\Delta;\Gamma \vdash \tau : \sigma \to \kappa \quad \|\Delta\|, \Gamma \vdash e : \sigma}{\Delta;\Gamma \vdash \tau\, e : \kappa}\ \text{App}$$

$$\frac{\Delta;\Gamma \vdash \tau : \mathsf{T} \quad \Delta;\Gamma, x{:}\llbracket\tau\rrbracket_{\text{rep}} * \llbracket\tau\rrbracket_{\text{PD}} \vdash \tau' : \mathsf{T}}{\Delta;\Gamma \vdash \Sigma\, x{:}\tau.\tau' : \mathsf{T}}\ \text{Prod}$$

$$\frac{\Delta;\Gamma \vdash \tau : \mathsf{T} \quad \Delta;\Gamma \vdash \tau' : \mathsf{T}}{\Delta;\Gamma \vdash \tau + \tau' : \mathsf{T}}\ \text{Sum} \qquad \frac{\Delta;\Gamma \vdash \tau : \mathsf{T} \quad \Delta;\Gamma \vdash \tau' : \mathsf{T}}{\Delta;\Gamma \vdash \tau \,\&\, \tau' : \mathsf{T}}\ \text{Intersection}$$

$$\frac{\Delta;\Gamma \vdash \tau : \mathsf{T} \quad \|\Delta\|, \Gamma, x{:}\llbracket\tau\rrbracket_{\text{rep}} * \llbracket\tau\rrbracket_{\text{PD}} \vdash e : \mathtt{bool}}{\Delta;\Gamma \vdash \{x{:}\tau \mid e\} : \mathsf{T}}\ \text{Con}$$

$$\frac{\begin{array}{c}\Delta;\Gamma \vdash \tau : \mathsf{T} \qquad \Delta;\Gamma \vdash \tau_s : \mathsf{T} \qquad \Delta;\Gamma \vdash \tau_t : \mathsf{T} \\ \|\Delta\|, \Gamma \vdash e : \llbracket\tau_m\rrbracket_{\text{rep}} * \llbracket\tau_m\rrbracket_{\text{PD}} \to \mathtt{bool} \quad (\tau_m = \tau\ \mathtt{seq}(\tau_s, e, \tau_t))\end{array}}{\Delta;\Gamma \vdash \tau\ \mathtt{seq}(\tau_s, e, \tau_t) : \mathsf{T}}\ \text{Seq}$$

$$\frac{\vdash \|\Delta\|, \Gamma\ \mathsf{ok} \quad \alpha{:}\mathsf{T} \in \Delta}{\Delta;\Gamma \vdash \alpha : \mathsf{T}}\ \text{TyVar} \quad \frac{\Delta, \alpha{:}\mathsf{T};\Gamma \vdash \tau : \mathsf{T}}{\Delta;\Gamma \vdash \mu\alpha.\tau : \mathsf{T}}\ \text{Rec} \quad \frac{\Delta, \alpha{:}\mathsf{T};\Gamma \vdash \tau : \kappa}{\Delta;\Gamma \vdash \lambda\alpha.\tau : \mathsf{T} \to \kappa}\ \text{TyAbs} \quad \frac{\Delta;\Gamma \vdash \tau_1 : \mathsf{T} \to \kappa \quad \Delta;\Gamma \vdash \tau_2 : \mathsf{T}}{\Delta;\Gamma \vdash \tau_1\, \tau_2 : \kappa}\ \text{TyApp}$$

$$\frac{\vdash \|\Delta\|, \Gamma\ \mathsf{ok} \quad \|\Delta\|, \Gamma \vdash e : \sigma \quad \llbracket\Delta\rrbracket_{\text{rep}} \vdash \sigma :: \mathsf{T}}{\Delta;\Gamma \vdash \mathtt{compute}(e{:}\sigma) : \mathsf{T}}\ \text{Compute} \qquad \frac{\Delta;\Gamma \vdash \tau : \mathsf{T}}{\Delta;\Gamma \vdash \mathtt{absorb}(\tau) : \mathsf{T}}\ \text{Absorb} \qquad \frac{\Delta;\Gamma \vdash \tau : \mathsf{T}}{\Delta;\Gamma \vdash \mathtt{scan}(\tau) : \mathsf{T}}\ \text{Scan}$$

Figure 18. DDC$^\alpha$ Kinding Rules

2. *If* $\mathcal{B}_{kind}(C) = \sigma \to \mathsf{T}$ *then* $\mathcal{B}_{opty}(C) = \sigma \rightharpoonup \llbracket C(e){:}\mathsf{T}\rrbracket_{PT}$ *(for any* e*).*
3. $\vdash \mathcal{B}_{type}(C) :: \mathsf{T}$.

C. Host Language

Bits	B	$::=$	$\cdot \mid 0\,B \mid 1\,B$
Constants	c	$::=$	$() \mid \mathtt{true} \mid \mathtt{false} \mid 0 \mid 1 \mid -1 \mid \ldots$
		$\mid$	$\mathtt{none} \mid B \mid \omega \mid \mathtt{ok} \mid \mathtt{err} \mid \mathtt{fail} \mid \ldots$
Values	v	$::=$	$c \mid \mathtt{fun}\ f\ x = e \mid (v, v)$
		$\mid$	$\mathtt{inl}\ v \mid \mathtt{inr}\ v \mid [\vec{v}]$
Operators	op	$::=$	$= \mid < \mid \mathtt{not} \mid \ldots$
Expressions	e	$::=$	$c \mid x \mid op(e) \mid \mathtt{fun}\ f\ x = e \mid e\ e$
		$\mid$	$\Lambda\alpha.e \mid e\ [\tau]$
		$\mid$	$\mathtt{let}\ x = e\ \mathtt{in}\ e \mid \mathtt{if}\ e\ \mathtt{then}\ e\ \mathtt{else}\ e$
		$\mid$	$(e, e) \mid \pi_i\, e \mid \mathtt{inl}\ e \mid \mathtt{inr}\ e$
		$\mid$	$\mathtt{case}\ e\ \mathtt{of}\ (\mathtt{inl}\ x \Rightarrow e \mid \mathtt{inr}\ x \Rightarrow e)$
		$\mid$	$[\vec{e}] \mid \mathtt{e} \mathbin{@} \mathtt{e} \mid e\,[e]$
		$\mid$	$\mathtt{fold}[\mu\alpha.\tau]\, e \mid \mathtt{unfold}[\mu\alpha.\tau]\, e$
Base Types	a	$::=$	$\mathtt{unit} \mid \mathtt{bool} \mid \mathtt{int} \mid \mathtt{none}$
		$\mid$	$\mathtt{bits} \mid \mathtt{offset} \mid \mathtt{errcode}$
Types	σ	$::=$	$a \mid \alpha \mid \sigma \to \sigma \mid \sigma * \sigma \mid \sigma + \sigma$
		$\mid$	$\sigma\,\mathtt{seq} \mid \forall\alpha.\sigma \mid \mu\alpha.\sigma \mid \lambda\alpha.\sigma \mid \sigma\ \sigma$
Kinds	κ	$::=$	$\mathsf{T} \mid \kappa \to \kappa$

D. Helper Functions

Generic Helpers:

```
Eof : bits * offset → bool
scanMax : int
fun max (m, n) = if m > n then m else n
fun pos n = if n = 0 then 0 else 1
fun isOk p = pos(p.h.nerr) = 0
fun isErr p = pos(p.h.nerr) = 1
fun max_ec (ec_1, ec_2) =
  if ec_1 = fail or ec_2 = fail then fail
  else if ec_1 = err or ec_2 = err then err
  else ok
```

Type-Specific Helpers:

```
fun R_unit () = ()
fun P_unit ω = ((0, ok, (ω, ω)), ())

fun R_bottom () = none
fun P_bottom ω = ((1, fail, (ω, ω)), ())

fun R_Σ (r_1, r_2) = (r_1, r_2)
fun H_Σ (h_1, h_2) =
  let nerr = pos(h_1.nerr) + pos(h_2.nerr) in
  let ec = if h_2.ec = fail then fail
     else max_ec h_1.ec h_2.ec in
  let sp = (h_1.sp.begin, h_2.sp.end) in
     (nerr, ec, sp)
fun P_Σ (p_1, p_2) = (H_Σ(p_1.h, p_2.h), (p_1, p_2))

fun R_+left r = inl r
fun R_+right r = inr r
fun H_+ h = (pos(h.nerr), h.ec, h.sp)
fun P_+left p = (H_+ p.h, inl p)
```

$\boxed{[\![\tau]\!]_{\mathrm{P}} = e}$

```
$[\![\mathtt{unit}]\!]_{\mathrm{P}} = \lambda(\mathtt{B},\omega).(\omega, \mathtt{R}_{\mathtt{unit}}(), \mathtt{P}_{\mathtt{unit}}(\omega))$
$[\![\mathtt{bottom}]\!]_{\mathrm{P}} = \lambda(\mathtt{B},\omega).(\omega, \mathtt{R}_{\mathtt{bottom}}(), \mathtt{P}_{\mathtt{bottom}}(\omega))$
$[\![C(e)]\!]_{\mathrm{P}} = \lambda(\mathtt{B},\omega).\mathcal{B}_{\mathrm{imp}}(C)\ (e)\ (\mathtt{B},\omega)$
$[\![\lambda x.\tau]\!]_{\mathrm{P}} = \lambda x.[\![\tau]\!]_{\mathrm{P}}$
$[\![\tau\, e]\!]_{\mathrm{P}} = [\![\tau]\!]_{\mathrm{P}}\ e$
$[\![\Sigma\, x{:}\tau.\tau']\!]_{\mathrm{P}} =$
 $\lambda(\mathtt{B},\omega)$.
   let $(\omega',\mathtt{r},\mathtt{p})$ = $[\![\tau]\!]_{\mathrm{P}}$ $(\mathtt{B},\omega)$ in
   let x = (r, p) in
   let $(\omega'',\mathtt{r}',\mathtt{p}')$ = $[\![\tau']\!]_{\mathrm{P}}$ $(\mathtt{B},\omega')$ in
   $(\omega'', \mathtt{R}_\Sigma(\mathtt{r},\mathtt{r}'), \mathtt{P}_\Sigma(\mathtt{p},\mathtt{p}'))$
$[\![\tau + \tau']\!]_{\mathrm{P}} =$
 $\lambda(\mathtt{B},\omega)$.
   let $(\omega',\mathtt{r},\mathtt{p})$ = $[\![\tau]\!]_{\mathrm{P}}$ $(\mathtt{B},\omega)$ in
   if isOk(p) then $(\omega', \mathtt{R}_{\mathtt{+left}}(\mathtt{r}), \mathtt{P}_{\mathtt{+left}}(\mathtt{p}))$
   else let $(\omega',\mathtt{r},\mathtt{p})$ = $[\![\tau']\!]_{\mathrm{P}}$ $(\mathtt{B},\omega)$ in
   $(\omega', \mathtt{R}_{\mathtt{+right}}(\mathtt{r}), \mathtt{P}_{\mathtt{+right}}(\mathtt{p}))$
$[\![\tau\,\&\,\tau']\!]_{\mathrm{P}} =$
 $\lambda(\mathtt{B},\omega)$.
   let $(\omega',\mathtt{r},\mathtt{p})$ = $[\![\tau]\!]_{\mathrm{P}}$ $(\mathtt{B},\omega)$ in
   let $(\omega'',\mathtt{r}',\mathtt{p}')$ = $[\![\tau']\!]_{\mathrm{P}}$ $(\mathtt{B},\omega)$ in
   $(\mathtt{max}(\omega',\omega''), \mathtt{R}_\&(\mathtt{r},\mathtt{r}'), \mathtt{P}_\&(\mathtt{p},\mathtt{p}'))$
$[\![\{x{:}\tau \mid e\}]\!]_{\mathrm{P}} =$
 $\lambda(\mathtt{B},\omega)$.
   let $(\omega',\mathtt{r},\mathtt{p})$ = $[\![\tau]\!]_{\mathrm{P}}$ $(\mathtt{B},\omega)$ in
   let x = (r, p) in
   let c = $e$ in
   $(\omega', \mathtt{R}_{\mathrm{con}}(\mathtt{c},\mathtt{r}), \mathtt{P}_{\mathrm{con}}(\mathtt{c},\mathtt{p}))$
```

```
$[\![\tau\ \mathtt{seq}(\tau_s, e, \tau_t)]\!]_{\mathrm{P}} =$
 $\lambda(\mathtt{B},\omega)$.
   letfun isDone $(\omega,\mathtt{r},\mathtt{p})$ =
     EoF$(\mathtt{B},\omega)$ or $e$ (r, p) or
     let $(\omega',\mathtt{r}',\mathtt{p}')$ = $[\![\tau_t]\!]_{\mathrm{P}}(\mathtt{B},\omega)$ in
     isOk(p')
   in
   letfun continue $(\omega,\omega',\mathtt{r},\mathtt{p})$ =
     if $\omega = \omega'$ or isDone $(\omega',\mathtt{r},\mathtt{p})$ then $(\omega',\mathtt{r},\mathtt{p})$
     else let $(\omega_{\mathrm{s}},\mathtt{r}_{\mathrm{s}},\mathtt{p}_{\mathrm{s}})$ = $[\![\tau_s]\!]_{\mathrm{P}}$ $(\mathtt{B},\omega')$ in
     let $(\omega_{\mathrm{e}},\mathtt{r}_{\mathrm{e}},\mathtt{p}_{\mathrm{e}})$ = $[\![\tau]\!]_{\mathrm{P}}$ $(\mathtt{B},\omega_{\mathrm{s}})$ in
     continue $(\omega,\omega_{\mathrm{e}},\mathtt{R}_{\mathrm{seq}}(\mathtt{r},\mathtt{r}_{\mathrm{e}}),\mathtt{P}_{\mathrm{seq}}(\mathtt{p},\mathtt{p}_{\mathrm{s}},\mathtt{p}_{\mathrm{e}}))$
   in
   let r = $\mathtt{R}_{\mathtt{seq\_init}}()$ in
   let p = $\mathtt{P}_{\mathtt{seq\_init}}(\omega)$ in
   if isDone $(\omega,\mathtt{r},\mathtt{p})$ then $(\omega,\mathtt{r},\mathtt{p})$
   else let $(\omega_{\mathrm{e}},\mathtt{r}_{\mathrm{e}},\mathtt{p}_{\mathrm{e}})$ = $[\![\tau]\!]_{\mathrm{P}}$ $(\mathtt{B},\omega)$ in
   continue $(\omega',\omega_{\mathrm{e}},\mathtt{R}_{\mathrm{seq}}(\mathtt{r},\mathtt{r}_{\mathrm{e}}),\mathtt{P}_{\mathrm{seq}}(\mathtt{p},\mathtt{P}_{\mathtt{unit}}(\omega),\mathtt{p}_{\mathrm{e}}))$
$[\![\alpha]\!]_{\mathrm{P}} = \mathtt{parse}_\alpha$
$[\![\mu\alpha.\tau]\!]_{\mathrm{P}} =$
 fun $\mathtt{parse}_\alpha$ (B:bits, $\omega$:offset) : offset $*$ $[\![\mu\alpha.\tau]\!]_{\mathrm{rep}}$ $*$ $[\![\mu\alpha.\tau]\!]_{\mathrm{PD}}$ =
   let $(\omega',\mathtt{r},\mathtt{p})$ = $[\![\tau]\!]_{\mathrm{P}}[[\![\mu\alpha.\tau]\!]_{\mathrm{rep}}/\alpha_{\mathrm{rep}}][[\![\mu\alpha.\tau]\!]_{\mathrm{PDb}}/\alpha_{\mathtt{PDb}}]$ $(\mathtt{B},\omega)$ in
     $(\omega', \mathtt{fold}[[\![\mu\alpha.\tau]\!]_{\mathrm{rep}}]\ \mathtt{r}, (\mathtt{p.h}, \mathtt{fold}[[\![\mu\alpha.\tau]\!]_{\mathrm{PDb}}]\ \mathtt{p}))$
$[\![\lambda\alpha.\tau]\!]_{\mathrm{P}} = \Lambda\alpha_{\mathtt{rep}}.\Lambda\alpha_{\mathtt{PDb}}.\lambda\mathtt{parse}_\alpha.[\![\tau]\!]_{\mathrm{P}}$
$[\![\tau_1\tau_2]\!]_{\mathrm{P}} = [\![\tau_1]\!]_{\mathrm{P}}\ [[\![\tau_2]\!]_{\mathrm{rep}}]\ [[\![\tau_2]\!]_{\mathrm{PDb}}]\ [\![\tau_2]\!]_{\mathrm{P}}$
$[\![\mathtt{compute}(e{:}\sigma)]\!]_{\mathrm{P}} =$
  $\lambda(\mathtt{B},\omega).(\omega, \mathtt{R}_{\mathrm{compute}}(e), \mathtt{P}_{\mathrm{compute}}(\omega))$
$[\![\mathtt{absorb}(\tau)]\!]_{\mathrm{P}} =$
 $\lambda(\mathtt{B},\omega)$.
   let $(\omega',\mathtt{r},\mathtt{p})$ = $[\![\tau]\!]_{\mathrm{P}}$ $(\mathtt{B},\omega)$ in
   $(\omega', \mathtt{R}_{\mathrm{absorb}}(\mathtt{p}), \mathtt{P}_{\mathrm{absorb}}(\mathtt{p}))$
$[\![\mathtt{scan}(\tau)]\!]_{\mathrm{P}} =$
 $\lambda(\mathtt{B},\omega)$.
   letfun try i =
     let $(\omega',\mathtt{r},\mathtt{p})$ = $[\![\tau]\!]_{\mathrm{P}}$ $(\mathtt{B},\omega+\mathtt{i})$ in
     if isOk(p) then
     $(\omega', \mathtt{R}_{\mathrm{scan}}(\mathtt{r}), \mathtt{P}_{\mathrm{scan}}(\mathtt{i}, \mathtt{sub}(\mathtt{B},\omega,\mathtt{i}+1), \mathtt{p}))$ else
     if EoF$(\mathtt{B},\omega+\mathtt{i})$ then
     $(\omega, \mathtt{R}_{\mathrm{scan\_err}}(), \mathtt{P}_{\mathrm{scan\_err}}(\omega))$ else
     try (i + 1)
   in try 0
```

Figure 19. DDC$^\alpha$ Parsing Semantics

```
fun $\mathtt{P}_{\mathtt{+right}}$ p = ($\mathtt{H}_+$ p.h, inr p)

fun $\mathtt{R}_\&$ (r, r') = (r, r')
fun $\mathtt{H}_\&$ ($\mathtt{h}_1$, $\mathtt{h}_2$) =
  let nerr = pos($\mathtt{h}_1$.nerr) + pos($\mathtt{h}_2$.nerr) in
  let ec = if $\mathtt{h}_1$.ec = fail and $\mathtt{h}_2$.ec = fail then fail
    else max_ec $\mathtt{h}_1$.ec $\mathtt{h}_2$.ec in
  let sp = ($\mathtt{h}_1$.sp.begin, max($\mathtt{h}_1$.sp.end, $\mathtt{h}_2$.sp.end)) in
    (nerr, ec, sp)
fun $\mathtt{P}_\&$ ($\mathtt{p}_1$, $\mathtt{p}_2$) = ($\mathtt{H}_\&$ ($\mathtt{p}_1$.h, $\mathtt{p}_2$.h), ($\mathtt{p}_1$, $\mathtt{p}_2$))

fun $\mathtt{R}_{\mathrm{con}}$ (c, r) = if c then inl r else inr r
fun $\mathtt{P}_{\mathrm{con}}$ (c, p) =
  if c then ((pos(p.h.nerr), p.h.ec, p.h.sp), p)
  else ((1 + pos(p.h.nerr), max_ec err p.h.ec, p.h.sp), p)


fun $\mathtt{R}_{\mathtt{seq\_init}}$ () = (0, [])
fun $\mathtt{P}_{\mathtt{seq\_init}}$ $\omega$ = ((0, ok, ($\omega$, $\omega$)), (0, 0, []))
fun $\mathtt{R}_{\mathrm{seq}}$ (r, $\mathtt{r}_{\mathrm{e}}$) = (r.len + 1, r.elts @ [$\mathtt{r}_{\mathrm{e}}$])
```

```
fun $\mathtt{H}_{\mathrm{seq}}$ (h, $\mathtt{h}_{\mathrm{s}}$, $\mathtt{h}_{\mathrm{e}}$) =
  let eerr = if h.neerr = 0 and $\mathtt{h}_{\mathrm{e}}$.nerr > 0
    then 1 else 0 in
  let nerr = h.nerr + pos($\mathtt{h}_{\mathrm{s}}$.nerr) + eerr in
  let ec = if $\mathtt{h}_{\mathrm{e}}$.ec = fail then fail
    else max_ec h.ec $\mathtt{h}_{\mathrm{e}}$.ec in
  let sp = (h.sp.begin, $\mathtt{h}_{\mathrm{e}}$.sp.end) in
    (nerr, ec, sp)
fun $\mathtt{P}_{\mathrm{seq}}$ (p, $\mathtt{p}_{\mathrm{s}}$, $\mathtt{p}_{\mathrm{e}}$) =
  ($\mathtt{H}_{\mathrm{seq}}$ (p.h, $\mathtt{p}_{\mathrm{s}}$.h, $\mathtt{p}_{\mathrm{e}}$.h),
  (p.neerr + pos($\mathtt{p}_{\mathrm{e}}$.h.nerr), p.len + 1, p.elts @ [$\mathtt{p}_{\mathrm{e}}$]))

fun $\mathtt{R}_{\mathrm{compute}}$ r = r
fun $\mathtt{P}_{\mathrm{compute}}$ $\omega$ = ((0, ok, ($\omega$, $\omega$)), ())

fun $\mathtt{R}_{\mathrm{absorb}}$ p = if isOk(p) then inl () else inr none
fun $\mathtt{P}_{\mathrm{absorb}}$ p = (p.h, ())

fun $\mathtt{R}_{\mathrm{scan}}$ r = inl r
fun $\mathtt{P_{scan}}$ (i, B, p) =
  let nerr = pos(i) + pos(p'.h.nerr) in
  let ec = if nerr = 0 then ok else err in
  let hdr = (nerr, ec, (p.sp.begin − i, p.sp.end)) in
     (hdr, inl (i, B, p))
fun $\mathtt{R_{scan\_err}}$ () = inr none
fun $\mathtt{P_{scan\_err}}$ $\omega$ = let hdr = (1, fail, ($\omega$, $\omega$)) in
  (hdr, inr ())
```

a

Definition 9 (Representation and PD Correlation Relation)
$\mathrm{Canon}_\nu(r, p)$ *iff exactly one of the following is true:*

- $\nu = \mathtt{unit}$ *and* $r = ()$ *and* $p.nerr = 0$.
- $\nu = \mathtt{bottom}$ *and* $r = \mathtt{none}$ *and* $p.nerr = 1$.
- $\nu = C(e)$ *and* $r = \mathtt{inl}\ c$ *and* $p.nerr = 0$.
- $\nu = C(e)$ *and* $r = \mathtt{inr\ none}$ *and* $p.nerr = 1$.
- $\nu = \Sigma\, x{:}\tau_1.\tau_2$ *and* $r = (r_1, r_2)$ *and* $p = (h, (p_1, p_2))$ *and* $h.nerr = \mathtt{pos}(p_1.nerr) + \mathtt{pos}(p_2.nerr)$, $\mathrm{Canon^*}_{\tau_1}(r_1, p_1)$ *and* $\mathrm{Canon^*}_{\tau_2[(r,p)/x]}(r_2, p_2)$.
- $\nu = \tau_1 + \tau_2$ *and* $r = \mathtt{inl}\ r'$ *and* $p = (h, \mathtt{inl}\ p')$ *and* $h.nerr = \mathtt{pos}(p'.nerr)$ *and* $\mathrm{Canon^*}_{\tau_1}(r', p')$.
- $\nu = \tau_1 + \tau_2$ *and* $r = \mathtt{inr}\ r'$ *and* $p = (h, \mathtt{inr}\ p')$ *and* $h.nerr = \mathtt{pos}(p'.nerr)$ *and* $\mathrm{Canon^*}_{\tau_2}(r', p')$.
- $\nu = \tau_1 \,\&\, \tau_2$, $r = (r_1, r_2)$ *and* $p = (h, (p_1, p_2))$, *and* $h.nerr = \mathtt{pos}(p_1.nerr) + \mathtt{pos}(p_2.nerr)$, $\mathrm{Canon^*}_{\tau_1}(r_1, p_1)$ *and* $\mathrm{Canon^*}_{\tau_2}(r_2, p_2)$.
- $\nu = \{x{:}\tau' \mid e\}$, $r = \mathtt{inl}\ r'$ *and* $p = (h, p')$, *and* $h.nerr = \mathtt{pos}(p'.nerr)$, $\mathrm{Canon^*}_{\tau'}(r', p')$ *and* $e[(r', p')/x] \rightarrow^* \mathtt{true}$.
- $\nu = \{x{:}\tau' \mid e\}$, $r = \mathtt{inr}\ r'$ *and* $p = (h, p')$, *and* $h.nerr = 1 + \mathtt{pos}(p'.nerr)$, $\mathrm{Canon^*}_{\tau'}(r', p')$ *and* $e[(r', p')/x] \rightarrow^* \mathtt{false}$.
- $\nu = \tau_e\ \mathtt{seq}(\tau_s, e, \tau_t,)$, $r = (len, [\vec{r_i}])$, $p = (h, (neerr, len', [\vec{p_i}]))$, $len = len'$, $neerr = \sum_{i=1}^{len} \mathtt{pos}(p_i.nerr)$, $\mathrm{Canon^*}_{\tau_e}(r_i, p_i)$, *(for* $i = 1 \ldots len$*), and* $h.nerr \geq \mathtt{pos}(neerr)$.
- $\nu = \mu\alpha.\tau'$, $p = (h, p')$, $p.nerr = p'.nerr$ *and* $\mathrm{Canon^*}_{\tau'[\mu\alpha.\tau'/\alpha]}(r, p')$.
- $\nu = \mathtt{compute}(e{:}\sigma)$ *and* $p.nerr = 0$.
- $\nu = \mathtt{absorb}(\tau')$, $r = \mathtt{inl}\ ()$, *and* $p.nerr = 0$.
- $\nu = \mathtt{absorb}(\tau')$, $r = \mathtt{inr\ none}$, *and* $p.nerr > 0$.
- $\nu = \mathtt{scan}(\tau')$, $r = \mathtt{inl}\ r'$, $p = (h, \mathtt{inl}\ (i, p'))$, $h.nerr = \mathtt{pos}(i) + \mathtt{pos}(p'.nerr)$, *and* $\mathrm{Canon^*}_{\tau'}(r', p')$.
- $\nu = \mathtt{scan}(\tau')$, $r = \mathtt{inr\ none}$, $p = (h, \mathtt{inr}\ ())$, *and* $h.nerr = 1$.

$\boxed{\llbracket\tau\rrbracket_{\mathrm{PP}} = e}$

```
$\llbracket\texttt{unit}\rrbracket_{\mathrm{PP}} = \lambda(\texttt{r},\texttt{pd}).\epsilon$
$\llbracket\texttt{bottom}\rrbracket_{\mathrm{PP}} = \lambda(\texttt{r},\texttt{pd}).\epsilon$
$\llbracket C(e)\rrbracket_{\mathrm{PP}} = \lambda(\texttt{r},\texttt{pd}).\mathcal{B}_{\mathrm{pp}}(C)\ (e)\ (\texttt{r},\texttt{pd})$
$\llbracket\lambda x.\tau\rrbracket_{\mathrm{PP}} = \lambda x.\llbracket\tau\rrbracket_{\mathrm{PP}}$
$\llbracket\tau\, e\rrbracket_{\mathrm{PP}} = \llbracket\tau\rrbracket_{\mathrm{PP}}\ e$
$\llbracket\Sigma\, x{:}\tau_1.\tau_2\rrbracket_{\mathrm{PP}} =$
  $\lambda(\texttt{r},\texttt{pd}).$
    let x = $(\texttt{r.1},\texttt{pd.2.1})$ in
    let $\texttt{bs}_1$ = $\llbracket\tau_1\rrbracket_{\mathrm{PP}}$ x in
    let $\texttt{bs}_2$ = $\llbracket\tau_2\rrbracket_{\mathrm{PP}}\ (\texttt{r.2},\texttt{pd.2.2})$ in
    $\texttt{bs}_1$ @ $\texttt{bs}_2$
$\llbracket\tau_1 + \tau_2\rrbracket_{\mathrm{PP}} =$
  $\lambda(\texttt{r},\texttt{pd}).$
    case $(\texttt{r},\texttt{pd.2})$ of
    $|\ (\texttt{inl r}_1, \texttt{inl p}_1) \Rightarrow \llbracket\tau_1\rrbracket_{\mathrm{PP}}\ (\texttt{r}_1,\texttt{p}_1)$
    $|\ (\texttt{inr r}_2, \texttt{inr p}_2) \Rightarrow \llbracket\tau_2\rrbracket_{\mathrm{PP}}\ (\texttt{r}_2,\texttt{p}_2)$
    $|\ \_ \Rightarrow \texttt{badInput}()$
$\llbracket\tau_1\,\&\,\tau_2\rrbracket_{\mathrm{PP}} =$
  $\lambda(\texttt{r},\texttt{pd}).$
    let $\texttt{p}_1$ = pd.2.1 in
    let $\texttt{p}_2$ = pd.2.2 in
    if $\texttt{p}_1\texttt{.h.sp.end} > \texttt{p}_2\texttt{.h.sp.end}$
      then $\llbracket\tau_1\rrbracket_{\mathrm{PP}}\ (\texttt{r.1},\texttt{p}_1)$
      else $\llbracket\tau_2\rrbracket_{\mathrm{PP}}\ (\texttt{r.2},\texttt{p}_2)$
$\llbracket\{x{:}\tau\ |\ e\}\rrbracket_{\mathrm{PP}} =$
  $\lambda(\texttt{r},\texttt{pd}).$
    case r of
    $|\ \texttt{inl r}_1 \Rightarrow \llbracket\tau\rrbracket_{\mathrm{PP}}\ (\texttt{r}_1,\texttt{pd.2})$
    $|\ \texttt{inr r}_2 \Rightarrow \llbracket\tau\rrbracket_{\mathrm{PP}}\ (\texttt{r}_2,\texttt{pd.2})$

$\llbracket\tau\ \texttt{seq}(\ell, e, \tau_t)\rrbracket_{\mathrm{PP}} =$
  $\lambda(\texttt{r},\texttt{pd}).$
    letfun print (rs, ps) =
      case (rs, ps) of
      $|\ ([],[]) \Rightarrow \epsilon$
      $|\ ([\texttt{r}],[\texttt{p}]) \Rightarrow \llbracket\tau\rrbracket_{\mathrm{PP}}\ (\texttt{r},\texttt{p})$
      $|\ (\texttt{r} :: \texttt{rs}, \texttt{p} :: \texttt{ps}) \Rightarrow$
        $\llbracket\tau\rrbracket_{\mathrm{PP}}\ (\texttt{r},\texttt{p})$ @
        printLit($\ell$) @
        print(rs, ps)
      $|\ \_ \Rightarrow \texttt{badInput}()$
    in
    print(r.elts, pd.elts)
$\llbracket\alpha\rrbracket_{\mathrm{PP}} = \texttt{print}_\alpha$
$\llbracket\mu\alpha.\tau\rrbracket_{\mathrm{PP}} =$
  fun $\texttt{print}_\alpha\ (\texttt{r} : \llbracket\mu\alpha.\tau\rrbracket_{\mathrm{rep}}, \texttt{pd} : \llbracket\mu\alpha.\tau\rrbracket_{\mathrm{PD}})$ : bits =
    $\llbracket\tau\rrbracket_{\mathrm{PP}}[\llbracket\mu\alpha.\tau\rrbracket_{\mathrm{rep}}/\alpha_{\texttt{rep}}][\llbracket\mu\alpha.\tau\rrbracket_{\mathrm{PDb}}/\alpha_{\texttt{PDb}}]$
      $(\texttt{unfold}[\llbracket\mu\alpha.\tau\rrbracket_{\mathrm{rep}}]\ \texttt{r}, \texttt{unfold}[\llbracket\mu\alpha.\tau\rrbracket_{\mathrm{PDb}}]\ \texttt{pd.2})$
$\llbracket\lambda\alpha.\tau\rrbracket_{\mathrm{PP}} = \Lambda\alpha_{\texttt{rep}}.\Lambda\alpha_{\texttt{PDb}}.\lambda\texttt{print}_\alpha.\llbracket\tau\rrbracket_{\mathrm{PP}}$
$\llbracket\tau_1\,\tau_2\rrbracket_{\mathrm{PP}} = \llbracket\tau_1\rrbracket_{\mathrm{PP}}\ [\llbracket\tau_2\rrbracket_{\mathrm{rep}}]\ [\llbracket\tau_2\rrbracket_{\mathrm{PDb}}]\ \llbracket\tau_2\rrbracket_{\mathrm{PP}}$
$\llbracket\texttt{compute}(e{:}\sigma)\rrbracket_{\mathrm{PP}} = \lambda(\texttt{r},\texttt{pd}).\epsilon$
$\llbracket\texttt{literal}(\ell)\rrbracket_{\mathrm{PP}} = \lambda((),\texttt{pd}).\texttt{printLit}(\ell)$
$\llbracket\texttt{scan}(\tau)\rrbracket_{\mathrm{PP}} =$
  $\lambda(\texttt{r},\texttt{pd}).$
    case $(\texttt{r},\texttt{pd.2})$ of
    $|\ (\texttt{inl r}_1, \texttt{inl p}_1) \Rightarrow \texttt{p}_1\texttt{.2}$ @ $\llbracket\tau\rrbracket_{\mathrm{PP}}\ (\texttt{r}_1,\texttt{p}_1\texttt{.3})$
    $|\ (\texttt{inr r}_2, \texttt{inr p}_2) \Rightarrow \epsilon$
    $|\ \_ \Rightarrow \texttt{badInput}()$
```

Figure 20. DDC$^\alpha$ Printing semantics