The Next 700 Data Description Languages

KATHLEEN FISHER
AT&T Labs Research
kfisher@research.att.com
and
YITZHAK MANDELBAUM
AT&T Labs Research
yitzhak@research.att.com
and
DAVID WALKER
Princeton University
dpw@CS.Princeton.EDU

1. THE CHALLENGE OF AD HOC DATA FORMATS

XML. HTML. CSV. JPEG. MPEG. These data formats represent vast quantities of industrial, governmental, scientific, and private data. Because they have been standardized and are widely used, many reliable, efficient, and convenient tools for processing data in these formats are readily available. For instance, your favorite programming language undoubtedly has libraries for parsing XML and HTML as well as reading and transforming images in JPEG or movies in MPEG. Query engines are available for querying XML documents. Widely-used applications like Microsoft Word and Excel automatically translate documents between HTML and other standard formats. In short, life is good when working with standard data formats. In an ideal world, all data would be in such formats. In reality, however, we are not nearly so fortunate.

An *ad hoc data format* is any non-standard data format. Typically, such formats do not have parsing, querying, analysis, or transformation tools readily available. Every day, network administrators, financial analysts, computer scientists, biologists, chemists, astronomers, and physicists deal with ad hoc data in a myriad of complex formats. Figure 1 gives a partial sense of the range and pervasiveness of such data. Since off-the-shelf tools for processing these ad hoc data formats do not exist or are not readily available, talented scientists, data analysts, and programmers must waste their time on low-level chores like parsing and format translation to extract the valuable information they need from their data.

This research was supported in part by National Science Foundation grants 0238328, 0612147 and 0615062. This work does not necessarily reflect the opinions or policy of the federal government or NSF and no official endorsement should be inferred.

Name & Use	Representation
Web server logs (CLF): Measure web workloads	Fixed-column ASCII records
AT&T provisioning data: Monitor service activation	Variable-width ASCII records
Call detail: Fraud detection	Fixed-width binary records
AT&T billing data: Monitor billing process	Various Cobol data formats
Netflow: Monitor network performance	Data-dependent number of fixed-width binary records
Newick: Immune system response simulation	Fixed-width ASCII records in tree-shaped hierarchy
Gene Ontology: Gene-gene correlations	Variable-width ASCII records in DAG-shaped hierarchy
CPT codes: Medical diagnoses	Floating point numbers
SnowMed: Medical clinic notes	keyword tags

Fig. 1. Selected ad hoc data sources.

Though the syntax of everyday programming languages might be considered "ad hoc," we explicitly exclude programming language syntax from our domain of interest.

In addition to the inconvenience of having to build custom processing tools from scratch, the nonstandard nature of ad hoc data frequently leads to other difficulties for its users. First, documentation for the format may not exist, or it may be out of date. For example, a common phenomenon is for a field in a data source to fall into disuse. After a while, a new piece of information becomes interesting, but compatibility issues prevent data suppliers from modifying the shape of their data, so instead they hijack the unused field, often failing to update the documentation in the process.

Second, such data frequently contain errors, for a variety of reasons: malfunctioning equipment, programming errors, non-standard values to indicate "no data available," human error in entering data, and unexpected data values caused by the lack of good documentation. Detecting errors is important, because otherwise they can corrupt "good" data. The appropriate response to such errors depends on the application. Some applications require the data to be error free: if an error is detected, processing needs to stop immediately and a human must be alerted. Other applications can repair the data, while still others can simply discard erroneous or unexpected values. For some applications, errors in the data can be the most interesting part because they can signal where two systems are failing to communicate.

Today, many programmers tackle the challenge of ad hoc data by writing scripts in a language like PERL. Unfortunately, this process is slow, tedious, and unreliable. Error checking and recovery in these scripts is often minimal or nonexistent because when present, such error code swamps the main-line computation. The program itself is often unreadable by anyone other than the original authors (and usually not even them in a month or two) and consequently cannot stand as documentation for the format. Processing code often ends up intertwined with parsing code, making it difficult to reuse the parsing code for different analyses. Hence, in general, software produced in this way is not the high-quality, reliable, efficient and maintainable code one should demand.

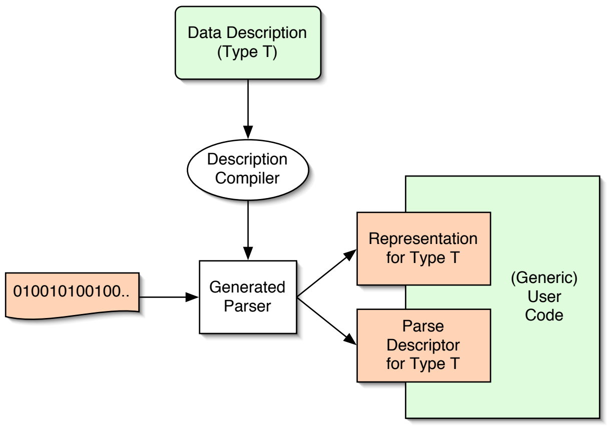

Fig. 2. Architecture of PADS system.

1.1 Promising Solutions

To address these challenges, researchers have begun to develop high-level languages for describing and processing ad hoc data. For instance, McCann and Chandra introduced PACKETTYPES [McCann and Chandra 2000], a specification language designed to help programmers process the binary data associated with networking protocols. Godmar Back developed DATASCRIPT [Back 2002], a scripting language with explicit support for specifying and parsing binary data formats. DATASCRIPT has been used to manipulate Java jar files and ELF object files. The developers of Erlang have also introduced language extensions that they refer to as *binaries* [Wikström and Rogvall 1999; Gustafsson and Sagonas 2004] to aid in packet processing and protocol programming. Finally, we are part of a group developing PADS, another system for managing ad hoc data. PADS focuses on robust error handling and tool generation. It is also unusual in that it supports a variety of data encodings: ASCII formats used by financial analysts, medical professionals and scientists, EBCDIC formats used in Cobol-based legacy business systems, binary data from network applications, and mixed encodings as well. PADS comes with not one but two specification languages: PADS/C [Fisher and Gruber 2005] generates libraries and tools for C programmers while PADS/ML [Mandelbaum et al. 2007] generates O'Caml code.

Although these languages differ in many details, they both derive their power from a remarkable insight: Types can describe data in both its external (on-disk) and internal (programmatic) forms. Figure 2 illustrates how systems such as PADS, DATASCRIPT, and PACKETTYPES exploit this dual interpretation of types. In the diagram, the data consumer constructs a type `T` to describe the syntax and semantic properties of the format in question. A compiler converts this description into parsing code, which maps raw data into a canonical in-memory *representation*. This canonical representation is guaranteed to be a data structure that itself has type `T`, or perhaps `T'`, the closest relative of `T` available in the host programming language being used. In the case of PADS, the parser also generates a *parse descriptor* (PD), which describes the errors detected in the data. A host language program can then analyze, transform or otherwise process the data representation and PD.

This architecture helps programmers take on the challenges of ad hoc data in multiple ways. First, format specifications in these languages serve as high-level documentation that is more easily read and maintained than the equivalent low-level PERL script or C parser. Importantly, DATASCRIPT, PACKETTYPES, and PADS all allow programmers to describe both the physical layout of data as well as its deeper semantic properties such as equality and range constraints on values, sortedness, and other forms of dependency. The intent is to allow analysts to capture all they know about a data source in a data description. If a data source is changed, as data sources frequently are, by the extension of a record with an additional field or new variant, one often only needs to make a single local change to the declarative description to keep it up to date.

Second, basing the description language on type theory is especially helpful as ordinary programmers have built up strong intuitions about types. The designers of data description languages have been able to exploit these intuitions to make the syntax and semantics of descriptions particularly easy to understand, even for beginners. For instance, an array type is used to describe sequences of data objects, while union types are used to describe alternatives.

Third, programmers can write generic, type-directed programs that produce tools for purposes other than just parsing. For instance, McCann and Chandra suggest using PACKETTYPES specifications to generate packet filters and network monitors automatically. Back used DATASCRIPT to generate infrastructure for visitor patterns over parsed data. PADS generates a statistical data analyzer, a pretty printer, an XML translator and an auxiliary library that enables XQueries using the Galax query engine[Fernández et al. 2003]. It is the declarative, domain-specific nature of these data description languages that makes it possible to generate all these value-added tools for programmers. The suite of tools, all of which can be generated from a single description, provides additional incentive for programmers to keep documentation up-to-date.

Fourth, these data description languages facilitate insertion of error handling code. The generated parsers check all possible error cases: system errors related to the input file, buffer, or socket; syntax errors related to deviations in the physical format; and semantic errors in which the data violates user constraints. Because these checks appear only in generated code, they do not clutter the high-level declarative description of the data source. Moreover, since tools are generated automatically by a compiler rather than written by hand, they are far more likely to be robust and far less likely to have dangerous vulnerabilities such as buffer overflows.

In summary, data description languages such as DATASCRIPT, PACKETTYPES, Erlang, and PADS meet the challenge of processing ad hoc data by providing a concise and precise form of "living" data documentation and producing reliable tools that handle errors robustly.

1.2 The Next 700 Data Description Languages

> The languages people use to communicate with computers differ in their intended aptitudes, towards either a particular application area, or a particular phase of computer use (high level programming, program assembly, job scheduling, etc). They also differ in physical appearance, and more important, in logical structure. The question arises, do the idiosyncrasies reflect basic logical properties of the situations that are being catered for? Or are they accidents of history and personal background that may be obscuring fruitful developments? This question is clearly important if we are trying to predict or influence language evolution.

To answer it we must think in terms, not of languages, but of families of languages. That is to say we must systematize their design so that a new language is a point chosen from a well-mapped space, rather than a laboriously devised construction.

— P. J. Landin, *The Next 700 Programming Languages*, 1966 [Landin 1966].

Landin asserts that principled programming language design involves thinking in terms of "families of languages" and choosing from a "well-mapped space." However, so far, when it comes to the domain of processing ad hoc data, there is no well-mapped space and no systematic understanding of the family of languages one might be dealing with.

The primary goal of this paper is to begin to understand the family of ad hoc data processing languages. We do so, as Landin did, by developing a semantic framework for defining, comparing, and contrasting languages in our domain. This semantic framework revolves around the definition of a data description calculus (DDC$^{\alpha}$). This calculus uses types from a dependent type theory to describe various forms of ad hoc data: base types to describe atomic pieces of data and type constructors to describe richer structures. We show how to give a denotational semantics to DDC$^{\alpha}$ by interpreting types as parsing functions that map external representations (bits) to data structures in a typed lambda calculus. More precisely, these parsers produce both internal representations of the external data and parse descriptors that pinpoint errors in the original source.

For many domains, researchers have a solid understanding of what makes a "reasonable" or "unreasonable" language. For instance, a reasonable typed language is one in which values of a given type have a well-defined canonical form and "programs don't go wrong." On the other hand, when we began this research, it was not at all clear how to decide whether our data description language and its interpretation were "reasonable" or "unreasonable." A conventional sort of canonical forms property, for instance, is not relevant as the input data source is not under system control, and, as mentioned above, is frequently buggy. Consequently, we have had to define and formalize a new correctness criterion for the language. In a nutshell, rather than requiring input data be error-free, we require that the internal data structures produced by parsing satisfy their specification whereever the parse descriptor says they will. Our invariant allows data consumers to rely on the integrity of the internal data structures marked as error-free.

To study and compare PADS/C, PADS/ML, PACKETTYPES, DATASCRIPT, and/or some other data description language, we advocate translating the language into DDC$^{\alpha}$. The translation decomposes the relatively complex, high-level descriptions of the language in question into a series of lower-level DDC$^{\alpha}$ descriptions, which have all been formally defined. We have done this decomposition for IPADS, an idealized version of the PADS/C language that captures the essence of the actual implementation. We have also analyzed many of the features of PADS/ML, PACKETTYPES and DATASCRIPT using our model. The process of giving semantics to these languages highlighted features that were ambiguous or ill-defined in the documentation that we had available to us.

To our delight, the process of giving PADS/C a semantics in this framework has had additional benefits. In particular, since we defined the semantics by reviewing the existing implementation, we found (and fixed!) a couple of subtle bugs. The semantics has also raised several design questions that we are continuing to study. It has also helped us explore important extensions. In particular, driven by examples found in biological data [Consortium ; Newick], we decided to add recursion to PADS/C. We used our semantic framework to study the ramifications of this addition.

Finally, DDC$^{\alpha}$ has been instrumental in the development of our latest data description language, PADS/ML. Unlike PADS/C, which was created prior to our semantic analysis, PADS/ML was defined with DDC$^{\alpha}$ already in hand. The semantics was a useful guide in all aspects of the PADS/ML implementation, but particularly so in the development of polymorphic descriptions, a new feature in PADS/ML. The compilation invariants required for correct code generation in the presence of polymorphism are quite subtle. However, using DDC$^{\alpha}$, we were able to workout the details in an clean, elegant setting and prove our implementation technique was correct.

In summary, this article makes the following theoretical and practical contributions:

—We define a semantic framework for understanding and comparing data description languages such as PADS/C, PADS/ML, PACKETTYPES, and DATASCRIPT. No one has previously given a formal semantics to any of these languages. In fact, as far as we are aware, no one has developed a general and complete "theory of front-ends" that encompasses both a semantics for recognition of concrete, external syntax and a semantics for internal representation of this data within a rich, strongly-typed programming language.

—At the center of the framework is DDC$^{\alpha}$, a calculus of data descriptions based on a polymorphic, dependent type theory. We give a denotational semantics to DDC$^{\alpha}$ by interpreting types both as parsers and, more conventionally, as classifiers for parsed data.

—We define an important correctness criterion for our language, stating that all errors in the parsed data are reported in the parse descriptor. We prove DDC$^{\alpha}$ parsers maintain this property.

—We define IPADS, an idealized version of the PADS/C data description language that captures its essential features, and show how to give it a semantics by translating it into DDC$^{\alpha}$. The process of defining the semantics led to the discovery of several bugs in the actual implemention.

—We have given semantics to features from several other data description languages including PACKETTYPES and DATASCRIPT. As Landin asserts, this process helps us understand the families of languages in this domain and the totality of their features, so that we may engage in principled language design as opposed to falling prey to "accidents of history and personal background."

—We use IPADS and DDC$^{\alpha}$ to experiment with a definition and implementation strategy for recursive data types. Recursive types are essential for representing tree-shaped hierarchical data [Consortium ; Newick]. We have integrated recursion into PADS/C, using our theory as a guide.

—We also used IPADS and DDC$^{\alpha}$ as a guide for the implementation of PADS/ML, a new data description language for O'Caml. The chief difficulty in the design involved understanding how to compile polymorphic descriptions into O'Caml. Polymorphism allows for effective "description reuse" and fits elegantly in the context of typed functional programming languages like O'Caml. DDC$^{\alpha}$ served as a simple formal framework in which we could work out and prove the correctness of our implementation strategy.

Most of the basic ideas mentioned above were presented at the ACM Symposium on Principles of Programming Languages in 2006, in a paper with the same title [Fisher et al.

2006]. However, there are two important differences. First, we have found several ways to improve the structure of the semantics of the DDC$^{\alpha}$ since we first introduced it in 2006. In particular, we were able to eliminate the "contractiveness" constraint, which allowed us to simplify our earlier kinding rules substantially. They now take on a much more standard format. Second, we have added polymorphism to the calculus so that we may use it to understand the semantics of PADS/ML's polymorphic, recursive and dependent data types. The addition of polymorphism led to a number of technical challenges in the proof of correctness of our system. Finally, this article differs from our previously published work as it explains the proof techniques and all intermediate lemmas needed to achieve our formal results. We have omitted the line-by-line details of the proofs, but key cases of the most challenging lemmas may be found in Mandelbaum's Ph.D. thesis [Mandelbaum 2006].

The rest of the paper describes our contributions in detail. Section 2 gives a gentle introduction to data description languages by introducing IPADS. Sections 3, 4 and 5 explain the syntax, semantics and metatheory of DDC$^{\alpha}$. Section 6 discusses encodings of IPADS, PADS/ML, PACKETTYPES and DATASCRIPT in DDC$^{\alpha}$ and Section 7 explains how we have already made use of our semantics in practice. Sections 8 and 9 discuss related work and conclude. We have explicitly excluded discussion of a variety of practical considerations concerning the engineering of either the PADS/C or PADS/ML systems from this article so we may focus specifically on the semantics of data description languages. We consider engineering concerns, system performance and the architecture of the PADS tool generation system beyond the scope of this article.

2. IPADS: AN IDEALIZED DDL

In this section, we define IPADS, an idealized data description language. IPADS captures the essence of PADS/C in a fashion similar to the way that MinML [Harper 2005] captures the essence of ML or Featherweight Java [Igarashi et al. 1999] captures the essence of Java. The main goal of this section is to introduce the reader to the form and function of IPADS by giving its syntax and walking through a couple of examples. Though the syntax differs, the structure of PADS/C's relatives PADS/ML, PACKETTYPES, and DATASCRIPT are similar. Later sections will show how to give a formal semantics to IPADS.

Preliminary Concepts. Like PADS/C, PADS/ML, PACKETTYPES, and DATASCRIPT, IPADS data descriptions are types. These types specify both the external data format (a sequence of bits or characters) and a mapping into a data structure in the host programming language. In PADS/C, the host language is C; in IPADS, the host language is an extension of the polymorphic lambda calculus. For the most part, however, the specifics of the host language are unimportant.

A complete IPADS description is a sequence of type definitions terminated by a single type. This terminal type describes the entirety of a data source, making use of the previous type definitions to do so. IPADS type definitions can have one of two forms. The form ($\alpha = t$) introduces the type identifier α and binds it to IPADS type t. The type identifier may be used in subsequent types. The second form (**`Prec`** $\alpha = t$) introduces a recursive type definition. In this case, α may appear in t.

Complex IPADS descriptions are built by using type constructors to glue together a collection of simpler types. In our examples, we assume IPADS contains a wide variety of base types including integers (**`Puint32`** is an ASCII representation of an unsigned integer that may be represented internally in 32 bits), characters (**Pchar**), strings (**Pstring**), dates (**Pdate**), IP addresses (**Pip**), and others. In general, these base types may be parameterized. For instance, we will assume **Pstring** is parameterized by an argument that signals termination of the string. For example, **Pstring**(" ") describes any sequence of characters terminated by a space. (Note that we do not consider the space to be part of the parsed string; it will be part of the next object.) Similarly, **Puint16_FW**(3) is an unsigned 16-bit integer described in exactly 3 characters in the data source. In general, we write $C(e)$ for a base type parameterized by a (host language) expression e.

When interpreted as a parser, each of these base types reads the external data source and generates a pair of data structures in the host language. The first data structure is the *internal representation* and the second is the *parse descriptor*, which contains metadata collected during parsing. For instance, **Puint32** reads a series of digits and generates an unsigned 32-bit integer as its internal representation. **Pstring** generates a host-language string. **Pdate** might read dates in a multitude of different formats, but always generates a tuple with time, day, month, and year fields as its internal representation. Whenever an IPADS parser encounters an unexpected character or bit-sequence, it sets the internal representation to `none` (*i.e.* null) and notes the error in the parse descriptor.

An IPADS *Example.* IPADS contains a rich collection of type constructors for creating sophisticated descriptions of ad hoc data. We present these constructors through a series of examples. The first example, shown in Figure 3, describes the Common Web Log Format [Krishnamurthy and Rexford 2001], which web servers use to log the requests they receive. Figure 4 shows two sample records. Briefly, each line in a log file represents one request; a complete log may contain any number of requests. A request begins with an IP address followed by two optional ids. In the example, the ids are missing and dashes stand in for them. Next is a date, surrounded by square brackets. A string in quotation marks follows, describing the request. Finally, a pair of integers denotes the response code and the number of bytes returned to the client.

The IPADS description of web logs is most easily read from bottom to top. The terminal type, which describes an entire web log, is an array type. Arrays in IPADS take three arguments: a description of the array elements (in this case, `entry_t`), a description of the separator that appears between elements (in this case, a newline marker **Pnl**), and a description of the terminator (in this case, the end-of-file marker). PADS/C itself provides a much wider selection of separators and termination conditions, but these additional variations are of little semantic interest so we omit them from IPADS. The host language representation for an array is a sequence of elements. We do not represent separators or terminators internally.

We use a **Pstruct** to describe the contents of each line in a web log. Like an array, a **Pstruct** describes a sequence of objects in a data source. We represent the result of parsing a **Pstruct** as a tuple in the host language. The elements of a **Pstruct** are either named fields (*e.g.* `client :` **Pip**) or anonymous fields (*e.g.* " ["). The **Pstruct** `entry_t` declares that the first thing on the line is an IP address (**Pip**) followed by a space character (" "). Next, the data should contain an `authid_t` followed by another space, *etc.*

The last field of `entry_t` is quite different from the others. It has a **Pcompute** type, meaning it does not match any characters in the data source, but it does form a part of the internal representation used by host programs. The argument of a **Pcompute** field is an

```
authid_t = Punion {
  unauthorized : "-";
  id           : Pstring (" ");
};

response_t =
  Pfun(x:int) =
   Puint16_FW(x) Pwhere y.100 <= y and y < 600;

entry_t = Pstruct {
  client   : Pip;              " ";
  remoteid : authid_t;         " ";
  localid  : authid_t;         " [";
  date     : Pdate("]");       "] \"";
  request  : Pstring("\"");    "\" ";
  response : response_t 3;     " ";
  length   : Puint32;
  academic : Pcompute (getdomain client) == "edu" : bool;
};

entry_t Parray(Pnl, Peof)
```

Fig. 3. IPADS Common Web Log Format Description

arbitrary host language expression (and its type) that determines the value of the associated field. In the example, the field `academic` computes a boolean that indicates whether the web request came from an academic site. Notice that the computation depends upon a host language value constructed earlier — the value stored in the `client` field. IPADS structs are a form of dependent record and, in general, later fields may refer to the values contained in earlier ones.

The `entry_t` description uses the type `authid_t` to describe the two fields `remoteid` and `localid`. The `authid_t` type is a **`Punion`** with two branches. Unions are represented internally as sum types. If the data source can be described by the first branch (a dash), then the internal representation is the first injection into the sum. If the data source cannot be described by the first branch, but can be described by the second branch then the internal representation is the second injection. Otherwise, there is an error.

Finally, the `response_t` type is a **`Pfun`**, a user-defined parameterized type. The parameter of `response_t` is a host language integer. The body of the **`Pfun`** expression is a **`Puint16_FW`** where `x`, the fixed width, is the argument of the function. In addition, the value of the fixed-width integer is constrained by the **`Pwhere`** clause. In this case, the **`Pwhere`** clause demands that the fixed-width integer `y` that is read from the source lie between 100 and 599. Any value outside this range will be considered a semantic error. In general, a **`Pwhere`** clause may be attached to any type specification. It closely resembles the semantic constraints found in practical parser generators such as ANTLR [Parr and Quong 1995].

A Recursive IPADS *Example.* Figure 5 presents a second IPADS example. In this example, IPADS describes the Newick Standard format, a flat representation of tree-structured data. The leaves of the trees are names that describe an "entity". In our variant of Newick

```
207.136.97.49 - - [15/Oct/1997:18:46:51 -0700]
"GET /tk/p.txt HTTP/1.0" 200 30
tj62.aol.com - - [16/Oct/1997:14:32:22 -0700]
"POST /scpt/confirm HTTP/1.0" 200 941
```

Fig. 4. Sample Common Web Log Data. Each record is broken with a newline to format it on this page.

```
node_t = Popt Pstruct {
                 name : Pstring(":"); ":";
                 dist : Puint32;
               };

Prec tree_t = Punion {
    internal : Pstruct {
        "(";  branches : tree_t Parray(",",")");
        "):"; dist : Puint32;
      };
    leaf : node_t;
  };

Pstruct { body : tree_t; ";"; }

(* Example: (B:3,(A:5,C:10,E:2):12,D:0):32; *)
```

Fig. 5. IPADS Newick Format Description

Standard, leaf names may be omitted. If the leaf name does appear, it is followed by a colon and a number. The number describes the "distance" from the parent node. Microbiologists often use distances to describe the number of genetic mutations that have to occur to move from the parent to the child. An internal tree node may have any number of (comma-separated) children within parentheses. Distances follow the closed-paren of the internal tree node.

The Newick Standard format and other formats that describe tree-shaped hierarchies [Consortium ; Newick ;] provide strong motivation for including recursion in IPADS. We have not been able to find any useable description of Newick data as simple sequences (structs and arrays) and alternatives (unions); some kind of recursive description appears essential. The definition of the type `tree_t` introduces recursion. It also uses the type **`Popt`** t, a trivial union that either parses t or nothing at all.

Formal Syntax. Figure 6 summarizes the formal syntax of IPADS. Expressions e and types σ are taken from the host language, described in Section 3.2. Notice, however, that we use x for host language expression variables and α for IPADS type variables. In the examples, we have abbreviated the syntax in places. For instance, we omit the operator "**`Plit`**" and formal label x when specifying constant types in **`Pstruct`**s, writing "c;" instead of "x : **`Plit`** c;". In addition, all base types C formally have a single parameter, but we have omitted parameters for base types such as **`Puint32`**. Finally, the type **`Palt`**, which did not appear in the examples, describes data that is described by all the branches simultaneously and produces a set of values - one from each type. Intuitively, **`Palt`** is a form of intersection type.

$$
\begin{array}{llll}
\text{Types} & t & ::= & C(e) \mid \mathbf{Plit}\ c \mid \mathbf{Pfun}(x:\sigma) = t \mid t\ e \\
& & \mid & \mathbf{Pstruct}\{\overrightarrow{x{:}t}\} \mid \mathbf{Punion}\{\overrightarrow{x{:}t}\} \mid \mathbf{Palt}\{\overrightarrow{x{:}t}\} \mid t\ \mathbf{Pwhere}\ x.e \\
& & \mid & \mathbf{Popt}\ t \mid t\ \mathbf{Parray}(t,t) \mid \mathbf{Pcompute}\ e{:}\sigma \mid \alpha \mid \mathbf{Prec}\ \alpha.t \\
\text{Programs} & p & ::= & t \mid \alpha = t;\ p \mid \mathbf{Prec}\ \alpha = t;\ p
\end{array}
$$

Fig. 6. IPADS Syntax

3. A DATA DESCRIPTION CALCULUS

At the heart of our work is a data description calculus (DDC$^{\alpha}$), containing simple, orthogonal type constructors designed to capture the core features of data description languages. Consequently, the syntax of DDC$^{\alpha}$ is at a significantly lower level of abstraction than that of PADS/C, PADS/ML or IPADS. Like any of these languages, however, the form and function of DDC$^{\alpha}$ features are directly inspired by type theory.

Informally, we may divide the features that make up DDC$^{\alpha}$ into types and type operators. Each DDC$^{\alpha}$ type describes the external representation of a piece of data and implicitly specifies how to transform that external representation into an internal one. The internal representation includes both the transformed value and a *parse descriptor* that characterizes the errors that occurred during parsing. Type operators provide for description reuse by abstracting over types.

Syntactically, the primitives of the calculus are similar to the types found in many dependent type systems, with a number of additions specific to the domain of data description. The types are *dependent* because data parsed earlier often guides parsing of later data (*i.e.*the form of the later data *depends* on the earlier data). In addition, parsing ad hoc formats correctly often involves checking constraints phrased as expressions in some conventional programming language. Data description languages tend to draw their expressions from their *host language* – the programming language in which their generated software artifacts are encoded. The host language of PADS/C, for example, is C and therefore the PADS/C constraint language is also C. We mimic this design in DDC$^{\alpha}$ and choose a single language – a variant of F_{ω}– for expressing both the expressions embedded in types and the interpretations of DDC$^{\alpha}$. This host language is discussed further in Section 3.2.

3.1 DDC$^{\alpha}$ Syntax

Figure 7 shows the syntax of DDC$^{\alpha}$. Expressions e and types σ belong to the host language. We use kinds κ to classify types so that we can ensure their well-formedness. Kind T classifies types that describe data. Kinds $\sigma \rightarrow \kappa$ and $\mathsf{T} \rightarrow \kappa$ describe functions from values to types and type to types, respectively.

The most basic types are `unit` and `bottom`. The former describes the empty string while the latter describes no string, failing on all input. The syntax $C(e)$ denotes a base type C parameterized by expression e.

We provide abstraction $\lambda x.\tau$ and application $\tau\ e$ so that we may parameterize types by expressions. Dependent sum types $\Sigma\, x{:}\tau_1.\tau_2$ describe a sequence of values in which the second type may refer to the value of the first. Sum types $\tau_1 + \tau_2$ express flexibility in the data format, as they describe data matching either τ_1 or τ_2. Unlike regular expressions or context-free grammars, which allow nondeterministic choice, sum-type parsers are deterministic, transforming the data according to τ_1 when possible and *only* attempting to

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

Fig. 7. DDC$^\alpha$ syntax

use τ_2 if there is an error in τ_1. Intersection types $\tau_1 \,\&\, \tau_2$ describe data that match both τ_1 and τ_2. They transform a single set of bits to produce a pair of values, one from each type. Constrained types $\{x{:}\tau \mid e\}$ transform data according to the underlying type τ and then check that the constraint e holds when x is bound to the parsed value.

The type $\tau\, \texttt{seq}(\tau_s, e, \tau_t)$ represents a sequence of values of type τ. The type τ_s specifies the type of the separator found between elements of the sequence. For sequences without separators, we use `unit` as the separator type. Expression e is a boolean-valued function that examines the parsed sequence after each element is read to determine if the sequence has completed. For example, a function that checks if the sequence has 100 elements would terminate a sequence when it reaches length 100. The type τ_t is used when data following the array will signal termination. Commonly, constrained types are used to specify that a particular value terminates the sequence. For example, the type $\{x{:}\mathbf{Pchar} \mid x =';'\}$ specifies that a semicolon terminates the array. However, if no particular value or set of values terminates the array, then a type that never succeeds (like `bottom`) could be used to ensure that the array is not terminated based on τ_t.

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. Note that we call functions from types to types *type abstractions* in contrast to *value abstractions*, which are functions from values to types.

DDC$^\alpha$ also has a number of "active" types. These types describe actions to be taken during parsing rather than strictly describing the data format. Type $\texttt{compute}(e{:}\sigma)$ allows us to include an element in the parsed output that does not appear in the data stream (although it is likely dependent on elements that do), based on the value of expression e. In contrast, type $\texttt{absorb}(\tau)$ parses data according to type τ but does not return its result. This behavior is useful for data that is important for parsing, but uninteresting to users of the parsed data, such as a separator. The last of the "active" types is $\texttt{scan}(\tau)$, which scans the input for data that can be successfully transformed according to τ. This type provides a form of error recovery as it allows us to discard unrecognized data until the "synchronization" type τ is found.

3.2 Host Language

In Figure 8, we present the host language of DDC$^\alpha$, a straightforward extension of F_ω with recursion[1] and a variety of useful constants and operators. We use this host language both to encode the parsing semantics of DDC$^\alpha$ and to write the expressions that can appear within DDC$^\alpha$ itself.

[1]The syntax for `fold` and `unfold`, particularly the choice of annotating `unfold` with a type, is based on the presentation of recursive types in Pierce [Pierce 2002]

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

Fig. 8. The syntax of the host language, an extension of F_ω with recursion and a variety of useful constants and operators.

As the calculus is largely standard, we highlight only its unusual features. 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-supplied expressions appearing in DDC$^\alpha$ types. Our expressions include arbitrary length sequences $[\vec{e}]$, sequence append $e \mathbin{@} e'$, and sequence indexing $e\,[e']$.

The type `none` is the singleton type of the constant `none`. Types `errcode` and `offset` classify error codes and bit string offsets, respectively. The remaining types have standard meanings: function types, product types, sum types, sequence types (τ `seq`), type variables (α), polymorphic types ($\forall\alpha.\sigma$), and recursive types ($\mu\alpha.\sigma$).

We extend the formal syntax with some syntactic sugar for use in the rest of the paper: anonymous functions $\lambda x.e$ for `fun` $f\ x = e$, with $f \notin \mathrm{FV}(e)$; function bindings `letfun` $f\ x = e$ `in` e' for `let` f = `fun` $f\ x = e$ `in` e'; `span` for `offset` $*$ `offset`. We often use pattern-matching syntax for pairs in place of explicit projections, as in $\lambda(B,\omega).e$ and `let` $(\omega, r, p) = e$ `in` e'. Although we have no formal records with named fields, we use a (named) dot notation for commonly occuring projections. For example, for a pair x of representation and parse descriptor, we use x.`rep` and x.`pd` for the left and right projections of x, respectively. Also, sums and products are right-associative. Hence, for example, $a * b * c$ is shorthand for $a * (b * c)$.

The static semantics ($\Gamma \vdash e : \sigma$), operational semantics ($e \rightarrow e'$), and type equivalence ($\sigma \equiv \sigma'$) are those of F_ω extended with recursive functions and iso-recursive types and are entirely standard. See, for example, Pierce [Pierce 2002].

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 functions, with the expectation the reader can construct them from context.

3.3 Example

As an example, we present an abbreviated description of the common log format as it might appear in DDC$^\alpha$. For brevity, this description does not fully capture the semantics of the IPADS description from Section 2. Additionally, we use the standard abbreviation $\tau * \tau'$ for products and introduce a number of type abbreviations in the form $\mathtt{name} = \tau$ before giving the type that describes the data source.

$$\mathtt{S} = \lambda\mathtt{str}.\{\mathtt{s}{:}\mathtt{Pstring\_FW}(1) \mid \mathtt{s} = \mathtt{str}\}$$

$$\mathtt{authid\_t} = \mathtt{S}(\text{“}-\text{”}) + \mathtt{Pstring}(\text{“ ”})$$

$$\mathtt{response\_t} = \lambda\mathtt{x}.\{\mathtt{y}{:}\mathtt{Puint16\_FW}(\mathtt{x}) \mid 100 \leq \mathtt{y}\ \mathtt{and}\ \mathtt{y} < 600\}$$

$$\begin{array}{l}\mathtt{entry\_t} = \\ \ \Sigma\,\mathtt{client}{:}\mathtt{Pip}. \qquad\qquad \mathtt{S}(\text{“ ”}) * \\ \ \Sigma\,\mathtt{remoteid}{:}\mathtt{authid\_t}. \quad \mathtt{S}(\text{“ ”}) * \\ \ \Sigma\,\mathtt{response}{:}\mathtt{response\_t}\,3. \\ \quad \mathtt{compute}(\mathtt{getdomain}\ \mathtt{client} = \text{“edu”}{:}\mathtt{bool})\end{array}$$

$$\mathtt{entry\_t}\ \mathtt{seq}(\mathtt{S}(\text{“\textbackslash n”}), \lambda\mathtt{x}.\mathtt{false}, \mathtt{bottom})$$

In the example, we define type constructor `S` to encode literals with a constrained type. We also use the following informal translations: **`Pwhere`** becomes a set-type, **`Pstruct`** a series of dependent sums, **`Punion`** a series of sums, and **`Parray`** a sequence. As the array terminates at the end of the file, we use λ`x.false` and `bottom` to indicate the absence of termination condition and terminator, respectively.

4. DDC$^\alpha$ SEMANTICS

At first glance, the primitives of DDC$^\alpha$ are deceptively simple. However, deeper thought reveals that their semantics is multifaceted. For example, each basic type simultaneously describes a collection of valid bit strings, two datatypes in the host language – one for the data representation itself and one for its parse descriptor – and a transformation from bit strings, including invalid ones, into data and corresponding metadata.

We give semantics to DDC$^\alpha$ types using three primary semantic functions, each of which precisely conveys a particular facet of a type's meaning. The functions $[\![\,\cdot\,]\!]_{\text{rep}}$ and $[\![\,\cdot\,]\!]_{\text{PD}}$ describe the *representation semantics* of DDC$^\alpha$, detailing the types of the data's in-memory representation and parse descriptor. The function $[\![\,\cdot\,]\!]_{\text{P}}$ describes the *parsing semantics* of DDC$^\alpha$, defining a host language function for each type that parses bit strings to produce a representation and parse descriptor. We define the set of valid bit strings for each type to be those strings for which the PD indicates no errors when parsed. In addition to these three semantic functions, we define a normalization relation, which facilitates reasoning about parameterized descriptions.

We begin the technical discussion by describing a kinding judgment that checks if a type is well formed — the other semantic functions should only be applied to well-formed DDC$^\alpha$ types. We then specify the normalization relation after which we formalize the three-fold semantics of DDC$^\alpha$ types. For reference, Table I lists all the functions and judgments defined in this section and a brief description of each. Additionally, Table II lists all of the F_ω judgments that we reference.

$\Delta;\Gamma \vdash \tau : \kappa$	*type kinding*
$\tau \rightarrow \tau'$	*type normalization*
$[\![\tau]\!]_{\text{rep}} = \sigma$	*representation-type interpretation of* DDC$^\alpha$
$[\![\tau]\!]_{\text{PD}} = \sigma$	*parse-descriptor type interpretation of* DDC$^\alpha$
$[\![\tau]\!]_{\text{PDb}} = \sigma$	*pd-body type interpretation of* DDC$^\alpha$
$[\![\tau]\!]_{\text{P}} = e$	*parsing semantics of* DDC$^\alpha$
$[\![\tau{:}\kappa]\!]_{\text{PT}} = \sigma$	F_ω *type of specified type's parsing function (parser-type)*
$[\![\Delta]\!]_{\text{PT}} = \Gamma$	*parser-type interpretation lifted to entire context*
$[\![\Delta]\!]_{F_\omega} = \Gamma$	F_ω *image of* DDC$^\alpha$ *type context*
$[\![\Delta]\!]_{\text{rep}} = \Gamma$	*representation-type variables in* $[\![\Delta]\!]_{F_\omega}$
$[\![\Delta]\!]_{\text{PD}} = \Gamma$	*parse-descriptor type variables in* $[\![\Delta]\!]_{F_\omega}$

Table I. DDC$^\alpha$ functions and judgments defined in this section.

$\vdash \Gamma$ ok	*well-formed contexts*
$\Gamma \vdash \sigma :: \kappa$	*well-formed types*
$\sigma \equiv \sigma'$	*type equivalence*
$\Gamma \vdash e : \sigma$	*expression typing*
$e \rightarrow e'$	*expression evaluation*

Table II. F_ω judgments referenced in this section.

4.1 DDC$^\alpha$ Kinding

The kinding judgment defined in Figure 9 determines well-formed DDC$^\alpha$ types. We use two contexts to express our kinding judgment:

$$\begin{array}{lcl}\Gamma & ::= & \cdot \mid \Gamma, x{:}\sigma \\ \Delta & ::= & \cdot \mid \Delta, \alpha{:}\mathsf{T}\end{array}$$

Context Γ is a finite partial map that binds expression variables to their types. When appearing in F_ω judgments, such contexts may also contain type-variable bindings of the form $\alpha{::}\kappa$. Context Δ is a finite partial map that binds type variables to their kinds. We provide the following mappings from DDC$^\alpha$ contexts Δ to F_ω contexts Γ.

$$\begin{array}{ll}[\![\,\cdot\,]\!]_{\text{rep}} = \cdot & [\![\,\cdot\,]\!]_{\text{PD}} = \cdot \\ [\![\Delta,\alpha{:}\mathsf{T}]\!]_{\text{rep}} = [\![\Delta]\!]_{\text{rep}}, \alpha_{\texttt{rep}}{::}\mathsf{T} & [\![\Delta,\alpha{:}\mathsf{T}]\!]_{\text{PD}} = [\![\Delta]\!]_{\text{PD}}, \alpha_{\texttt{PDb}}{::}\mathsf{T}\end{array}$$

Translation $[\![\Delta]\!]_{F_\omega}$ simply combines the two ($[\![\Delta]\!]_{F_\omega} = [\![\Delta]\!]_{\text{rep}}, [\![\Delta]\!]_{\text{PD}}$). These translations are used when checking the well-formedness of contexts Γ and types σ with open type variables.

As the rules are mostly straightforward, we highlight just a few of them. In rule BASE, we use the function $\mathcal{B}_{\text{kind}}$ to assign kinds to base types. Base types must be fully applied to arguments of the right type. Once fully applied, all base types have kind T. Rule DEPSUM, for dependent sums, shows that the name of the first component is bound to a pair of a representation and corresponding PD. The semantic functions defined in the next section determine the type of this pair. Type abstractions and recursive types (rules TYABS

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

$$\frac{\vdash [\![\Delta]\!]_{F_\omega}, \Gamma\ \mathsf{ok}}{\Delta;\Gamma \vdash \mathtt{unit} : \mathsf{T}}\ \textsc{Unit} \qquad \frac{\vdash [\![\Delta]\!]_{F_\omega}, \Gamma\ \mathsf{ok}}{\Delta;\Gamma \vdash \mathtt{bottom} : \mathsf{T}}\ \textsc{Bottom} \qquad \frac{\begin{array}{c}\vdash [\![\Delta]\!]_{F_\omega}, \Gamma\ \mathsf{ok} \qquad [\![\Delta]\!]_{F_\omega}, \Gamma \vdash e : \sigma \\ \mathcal{B}_{\text{kind}}(C) = \sigma \to \mathsf{T}\end{array}}{\Delta;\Gamma \vdash C(e) : \mathsf{T}}\ \textsc{Const}$$

$$\frac{\Delta;\Gamma, x{:}\sigma \vdash \tau : \kappa}{\Delta;\Gamma \vdash \lambda x.\tau : \sigma \to \kappa}\ \textsc{Abs} \qquad \frac{\Delta;\Gamma \vdash \tau : \sigma \to \kappa \quad [\![\Delta]\!]_{F_\omega}, \Gamma \vdash e : \sigma}{\Delta;\Gamma \vdash \tau\, e : \kappa}\ \textsc{App}$$

$$\frac{\Delta;\Gamma \vdash \tau : \mathsf{T} \quad \Delta;\Gamma, x{:}[\![\tau]\!]_{\text{rep}} * [\![\tau]\!]_{\text{PD}} \vdash \tau' : \mathsf{T}}{\Delta;\Gamma \vdash \Sigma\, x{:}\tau.\tau' : \mathsf{T}}\ \textsc{DepSum}$$

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

$$\frac{\Delta;\Gamma \vdash \tau : \mathsf{T} \quad [\![\Delta]\!]_{F_\omega}, \Gamma, x{:}[\![\tau]\!]_{\text{rep}} * [\![\tau]\!]_{\text{PD}} \vdash e : \mathtt{bool}}{\Delta;\Gamma \vdash \{x{:}\tau \mid e\} : \mathsf{T}}\ \textsc{Con}$$

$$\frac{\begin{array}{c}\Delta;\Gamma \vdash \tau : \mathsf{T} \qquad \Delta;\Gamma \vdash \tau_s : \mathsf{T} \qquad \Delta;\Gamma \vdash \tau_t : \mathsf{T} \\ [\![\Delta]\!]_{F_\omega}, \Gamma \vdash e : [\![\tau_m]\!]_{\text{rep}} * [\![\tau_m]\!]_{\text{PD}} \to \mathtt{bool} \quad (\tau_m = \tau\, \mathtt{seq}(\tau_s, e, \tau_t))\end{array}}{\Delta;\Gamma \vdash \tau\, \mathtt{seq}(\tau_s, e, \tau_t) : \mathsf{T}}\ \textsc{Seq}$$

$$\frac{\vdash [\![\Delta]\!]_{F_\omega}, \Gamma\ \mathsf{ok} \quad \alpha{:}\mathsf{T} \in \Delta}{\Delta;\Gamma \vdash \alpha : \mathsf{T}}\ \textsc{TyVar} \qquad \frac{\Delta, \alpha{:}\mathsf{T};\Gamma \vdash \tau : \mathsf{T}}{\Delta;\Gamma \vdash \mu\alpha.\tau : \mathsf{T}}\ \textsc{Rec} \qquad \frac{\Delta, \alpha{:}\mathsf{T};\Gamma \vdash \tau : \kappa}{\Delta;\Gamma \vdash \lambda\alpha.\tau : \mathsf{T} \to \kappa}\ \textsc{TyAbs}$$

$$\frac{\Delta;\Gamma \vdash \tau_1 : \mathsf{T} \to \kappa \quad \Delta;\Gamma \vdash \tau_2 : \mathsf{T}}{\Delta;\Gamma \vdash \tau_1\, \tau_2 : \kappa}\ \textsc{TyApp} \qquad \frac{\vdash [\![\Delta]\!]_{F_\omega}, \Gamma\ \mathsf{ok} \quad [\![\Delta]\!]_{F_\omega}, \Gamma \vdash e : \sigma \quad [\![\Delta]\!]_{\text{rep}} \vdash \sigma :: \mathsf{T}}{\Delta;\Gamma \vdash \mathtt{compute}(e{:}\sigma) : \mathsf{T}}\ \textsc{Compute}$$

$$\frac{\Delta;\Gamma \vdash \tau : \mathsf{T}}{\Delta;\Gamma \vdash \mathtt{absorb}(\tau) : \mathsf{T}}\ \textsc{Absorb} \qquad \frac{\Delta;\Gamma \vdash \tau : \mathsf{T}}{\Delta;\Gamma \vdash \mathtt{scan}(\tau) : \mathsf{T}}\ \textsc{Scan}$$

Fig. 9. DDC$^\alpha$ kinding rules

and REC) restrict their type variable to kind T. This restriction simplifies the metatheory of DDC$^\alpha$ with little practical impact. Finally, with the introduction of potentially open host types, we must now check in rule COMPUTE that the only (potentially) open type variables in σ are the representation-type variables bound (implicitly) in Δ.

At the beginning of this chapter, we mentioned that DDC$^\alpha$ is an extension and improvement of our prior work on DDC. The improvements relate to changes in the kinding rules. In particular, we have replaced the context M of DDC, which mapped recursive-type variables to their definitions, with a simpler context Δ which merely assigns a kind (always T) to open type variables. The type variables bound by recursive types are now treated as

$$
\begin{array}{lrcl}
\text{Normal} & \nu & ::= & \texttt{unit} \mid \texttt{bottom} \mid C(e) \mid \lambda x.\tau \mid \Sigma\, x{:}\tau.\tau \\
\text{Types} & & \mid & \tau + \tau \mid \tau \,\&\, \tau \mid \{x{:}\tau \mid e\} \mid \tau\, \texttt{seq}(\tau, e, \tau) \\
 & & \mid & \mu\alpha.\tau \mid \lambda\alpha.\tau \\
 & & \mid & \texttt{compute}(e{:}\sigma) \mid \texttt{absorb}(\tau) \mid \texttt{scan}(\tau) \\
\text{Types} & \tau & ::= & \nu \mid \tau\, e \mid \tau\, \tau \mid \alpha
\end{array}
$$

Fig. 10. Revised DDC$^\alpha$ Syntax

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

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

Fig. 11. DDC$^\alpha$ weak-head normalization

abstract, just like the type variables bound by type abstractions. Correspondingly, the rule for type variables (TYVAR) now has a standard form, and the premise of the rule for recursive types (REC) is now nearly identical to the premise of the rule for type abstractions (TYABS).

4.2 DDC$^\alpha$ Normalization

To specify the rules of normalization, we must first refactor the syntax of DDC$^\alpha$ by distinguishing the subset of weak-head normal types (ν) from all types τ, as shown in Figure 10. In addition, we must define type and value substitution for DDC$^\alpha$. The notation $\tau'[\tau/\alpha]$ denotes standard capture-avoiding substitution of types into types, except for constructs that contain an F_ω expression e or type σ. For those constructs, the alternative substitution $[[\![\tau]\!]_{\text{rep}}/\alpha_{\texttt{rep}}][[\![\tau]\!]_{\text{PDb}}/\alpha_{\texttt{PDb}}]$ is applied to the subcomponent expression or type. For example,

$$\texttt{compute}(e{:}\sigma)[\tau/\alpha] = \texttt{compute}(e[[\![\tau]\!]_{\text{rep}}/\alpha_{\texttt{rep}}][[\![\tau]\!]_{\text{PDb}}/\alpha_{\texttt{PDb}}] : \sigma[[\![\tau]\!]_{\text{rep}}/\alpha_{\texttt{rep}}][[\![\tau]\!]_{\text{PDb}}/\alpha_{\texttt{PDb}}]).$$

This definition of substitution derives from the kinding rules of DDC$^\alpha$. In a judgment $\Delta, \alpha{:}\mathsf{T}; \Gamma \vdash \tau : \kappa$, the DDC$^\alpha$ type variable α implicitly binds the F_ω type variables $\alpha_{\texttt{rep}}$ and $\alpha_{\texttt{PDb}}$ for any types in Γ. Therefore, when replacing α in a DDC$^\alpha$ type, we must also make sure to replace all type variables $\alpha_{\texttt{rep}}$ and $\alpha_{\texttt{PDb}}$ in constituent F_ω expressions and types in a consistent manner. We denote standard capture-avoiding substitution of terms in DDC$^\alpha$ types with $\tau[v/x]$. Similarly, $\kappa[\sigma/\alpha]$ denotes standard capture-avoiding substitution of F_ω types into DDC$^\alpha$ kinds.

Normalization of DDC$^\alpha$ is based on a standard call-by-value small-step semantics of the lambda calculus. We present the rules of the normalization judgment in Figure 11.

4.3 Representation Semantics

In Figure 12, we present the representation type of each DDC$^\alpha$ primitive. While the primitives are dependent types, the host does not have such types, so the translation erases all dependency. Removing expressions from the types renders variable binding and application useless, so we drop those forms as well. Consequently, we translate abstraction and application according to their underlying types.

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

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

Fig. 12. Representation-type interpretation function.

In more detail, the DDC$^\alpha$ type `unit` consumes no input and produces only the `unit` value. Correspondingly, `bottom` consumes no input, but uniformly fails, producing the value `none`. The function $\mathcal{B}_{\text{type}}$ maps each base type to a representation for successfully parsed data. Note that this representation does not depend on the argument expression. As base type parsers can fail, we sum this type with `none` to produce the actual representation type. Intersection types produce a pair of values, one for each sub-type, because the representations of the subtypes need not be identical nor even compatible. Constrained types produce sums, where a left branch indicates the data satisfies the constraint and the right indicates it does not. In the latter case, the parser returns the offending data rather than `none` because the error is semantic rather than syntactic. Sequences produce a host language sequence paired with its length.

A type variable α in DDC$^\alpha$ 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. The output of a `compute` is exactly the computed value, and therefore shares its type. The output of `absorb` is a sum indicating whether parsing the underlying type succeeded or failed. The type of `scan` is similar, but also returns an element of the underlying type in case of success.

In Figure 13, we give the parse descriptor type for each DDC$^\alpha$ type. Each PD type has a header and body. This common shape allows us to define functions that polymorphically process PDs based on their headers. Each header 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 described by the descriptor. Formally, the type of the header (`pd_hdr`) is $\texttt{int} * \texttt{errcode} * \texttt{span}$. Each body consists of subdescriptors corresponding to the subcomponents of the representation and any type-specific metadata. For types with neither subcomponents nor special metadata, we use `unit` as the body type.

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

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

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

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

Fig. 13. Parse-descriptor type interpretation function

We discuss a few of the more complicated parse descriptors in detail. The parse descriptor body for sequences contains the parse descriptors of its elements, the number of element errors, and the sequence length. Note that the number of element errors is distinct from the number of sequence errors, as sequences can have errors that are not related to their elements (such as errors reading separators). We introduce an abbreviation for array PD body types, $\texttt{arr\_pd}\ \sigma = \texttt{int} * \texttt{int} * (\sigma\ \texttt{seq})$. The `compute` parse descriptors have no subelements because the data they describe is not parsed from the data source. The `absorb` PD type is `unit` as with its representation. We assume that just as the user does not want the representation to be kept, so too the parse descriptor. The `scan` parse descriptor is either `unit`, in case no match was found, or records the number of bits skipped before the type was matched along with the type's corresponding parse descriptor.

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

It is important to note that the PD interpretation is not defined for all types. The problem lies with the interpretation of type application ($[\![\tau_1\ \tau_2]\!]_{\text{PD}} = [\![\tau_1]\!]_{\text{PD}}\ [\![\tau_2]\!]_{\text{PDb}}$). The interpretation requires that $[\![\tau_2]\!]_{\text{PDb}}$ be defined, which, in turn, requires that $[\![\tau_2]\!]_{\text{PD}} \equiv \texttt{pd\_hdr} * \sigma$, for some σ. Yet, this requirement is not met by all types; for example, $\lambda\alpha.\tau$.

$\boxed{[\![\tau{:}\kappa]\!]_{\mathrm{PT}} = \sigma}$

$$
\begin{array}{lcl}
[\![\tau{:}\mathsf{T}]\!]_{\mathrm{PT}} & = & \mathtt{bits} * \mathtt{offset} \rightarrow \mathtt{offset} * [\![\tau]\!]_{\mathrm{rep}} * [\![\tau]\!]_{\mathrm{PD}} \\
[\![\tau{:}\sigma \rightarrow \kappa]\!]_{\mathrm{PT}} & = & \sigma \rightarrow [\![\tau e{:}\kappa]\!]_{\mathrm{PT}},\ \text{for any e.} \\
[\![\tau{:}\mathsf{T} \rightarrow \kappa]\!]_{\mathrm{PT}} & = & \forall \alpha_{\mathtt{rep}}.\forall \alpha_{\mathtt{PDb}}.[\![\alpha{:}\mathsf{T}]\!]_{\mathrm{PT}} \rightarrow [\![\tau\alpha{:}\kappa]\!]_{\mathrm{PT}} \\
 & & (\alpha_{\mathtt{rep}}, \alpha_{\mathtt{PDb}} \notin \mathsf{FTV}(\kappa) \cup \mathsf{FTV}(\tau))
\end{array}
$$

Fig. 14. F_ω types for parsing functions.

4.4 Parsing Semantics of the DDC$^\alpha$

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, and so forth. Figure 14 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.

Figure 15 shows the parsing semantics function. For each type, the input to the corresponding parser is a bit string and an offset which indicates the point in the bit string at which parsing should commence. The output is a new offset, a representation of the parsed data, and a parse descriptor. As the bit string input is never modified, it is not returned as an output. In addition to specifying how to handle correct data, each function describes how to transform corrupted bit strings, marking detected errors in a parse descriptor. The semantics function is partial, applying only to well-formed DDC$^\alpha$ types.

For any type, there are three steps to parsing: parse the subcomponents of the type (if any), assemble the resultant representation, and tabulate metadata based on subcomponent metadata (if any). For the sake of clarity, we have factored the latter two steps into separate representation and PD constructor functions which we define for many of the types. For some types, we additionally factor the PD header construction into a separate function. For example, the representation and PD constructors for `unit` are $\mathtt{R}_{\mathtt{unit}}$ and $\mathtt{P}_{\mathtt{unit}}$, respectively, and the header constructor for dependent sums is $\mathtt{H}_\Sigma$. The constructor functions are shown in Figure 17 and Figure 18. We have also factored out some commonly occuring code into auxiliary functions, explained as needed and defined formally in Figure 16.

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. Note that the purpose of the `fail` code is to indicate to any higher level elements that some form of error recovery is required. Hence, the whole parse is marked as failed exactly when the parse ends in failure. The error count is determined by subcomponent error counts and any errors associated directly with the type itself. If a subcomponent has errors then the error count is increased by one; otherwise it is not increased at all. We use the function `pos`, which maps all positive numbers to 1 (leaving zero as is), to assist in calculating the contribution of subcomponents to the total error count. Errors at the level of the element itself - such as constraint violation

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

```
$[\![\mathtt{unit}]\!]_{\mathrm{P}} = \lambda(\mathtt{B},\omega).(\omega, \mathtt{R}_{\mathtt{unit}}(), \mathtt{P}_{\mathtt{unit}}(\omega))$
$[\![\mathtt{bottom}]\!]_{\mathrm{P}} = \lambda(\mathtt{B},\omega).(\omega, \mathtt{R}_{\mathtt{bot}}(), \mathtt{P}_{\mathtt{bot}}(\omega))$
$[\![C(e)]\!]_{\mathrm{P}} = \lambda(\mathtt{B},\omega).\mathcal{B}_{\mathrm{imp}}(C)\ (e)\ (\mathtt{B},\omega)$
$[\![\lambda x.\tau]\!]_{\mathrm{P}} = \lambda x.[\![\tau]\!]_{\mathrm{P}}$
$[\![\tau\ e]\!]_{\mathrm{P}} = [\![\tau]\!]_{\mathrm{P}}\ e$
$[\![\Sigma\, x{:}\tau.\tau']\!]_{\mathrm{P}} =$
$\lambda(\mathtt{B},\omega)$.
   let $(\omega', \mathtt{r}, \mathtt{p})$ = $[\![\tau]\!]_{\mathrm{P}}\ (\mathtt{B},\omega)$ in
   let x = $(\mathtt{r}, \mathtt{p})$ 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 = $(\mathtt{r}, \mathtt{p})$ in
   let c = $e$ in
   $(\omega', \mathtt{R}_{\mathtt{con}}(\mathtt{c},\mathtt{r}), \mathtt{P}_{\mathtt{con}}(\mathtt{c},\mathtt{p}))$
```

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

Fig. 15. DDC$^{\alpha}$ parsing semantics

```
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
```

Fig. 16. Auxiliary functions. The type of PD headers is `int * errcode * span`. We refer to the projections using dot notation as `nerr`, `ec` and `sp`, respectively. A span is a pair of offsets, referred to as `begin` and `end`, respectively.

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

fun R_bot () = none
fun P_bot 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)
fun P_+right p = (H_+ p.h, inr p)

fun R_& (r, r') = (r, r')
fun H_& (h_1, h_2) =
 let nerr = pos(h_1.nerr) + pos(h_2.nerr) in
 let ec = if h_1.ec = fail and h_2.ec = fail then fail
    else max_ec h_1.ec h_2.ec in
 let sp = (h_1.sp.begin, max(h_1.sp.end, h_2.sp.end)) in
    (nerr, ec, sp)
fun P_& (p_1, p_2) = (H_& (p_1.h, p_2.h), (p_1, p_2))
```

Fig. 17. Constructor functions, part 1. The type of parse descriptor headers is `int * errcode * span`. We refer to the projections using dot notation as `nerr`, `ec` and `sp`, respectively. A span is a pair of offsets, referred to as `begin` and `end`, respectively.

```
fun R_con (c, r) = if c then inl r else inr r
fun P_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 R_seq_init () = (0, [])
fun P_seq_init ω = ((0, ok, (ω, ω)), (0, 0, []))
fun R_seq (r, r_e) = (r.len + 1, r.elts @ [r_e])
fun H_seq (h, h_s, h_e) =
 let eerr = if h.neerr = 0 and h_e.nerr > 0
   then 1 else 0 in
 let nerr = h.nerr + pos(h_s.nerr) + eerr in
 let ec = if h_e.ec = fail then fail
   else max_ec h.ec h_e.ec in
 let sp = (h.sp.begin, h_e.sp.end) in
   (nerr, ec, sp)
fun P_seq (p, p_s, p_e) =
 (H_seq (p.h, p_s.h, p_e.h),
 (p.neerr + pos(p_e.h.nerr), p.len + 1, p.elts @ [p_e]))

fun R_compute r = r
fun P_compute ω = ((0, ok, (ω, ω)), ())

fun R_absorb p = if isOk(p) then inl () else inr none
fun P_absorb p = (p.h, ())

fun R_scan r = inl r
fun P_scan (i, 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, p))
fun R_scan_err () = inr none
fun P_scan_err ω = let hdr = (1, fail, (ω, ω)) in
  (hdr, inr ())
```

Fig. 18. Constructor functions, part 2.

in constrained types - are generally counted individually.

With this background, we can now discuss the semantics. The `unit` and `bottom` descriptions do not consume any input. Hence, the output offset is the same as the input offset in the parsers for these constructs. A look at their constructors shows that the parse descriptor for `unit` always indicates no errors and a corresponding `ok` code, while that of `bottom` always indicates failure with an error count of one and the `fail` error code. The semantics of base types applies the implementation of the base type's parser, provided by the function $\mathcal{B}_{\text{imp}}$, to the appropriate arguments. Abstraction and application are defined directly in terms of host language abstraction and application. Dependent sums read the first element at ω and then the second at ω', the offset returned from parsing the first element. Notice that we bind the pair of the returned representation and parse descriptor to the variable x before parsing the second element, implicitly mapping the DDC$^{\alpha}$ variable x to the host language variable x in the process. Finally, we combine the results using the constructor functions, returning ω'' as the final offset of the parse.

Sums first attempt to parse according to the left type, returning the resulting value if it parses without errors. Otherwise, they parse according to the right type. Intersections read both types starting at the same offset. They advance the stream to the maximum of the two offsets returned by the component parsers. The construction of the parse descriptor is similar to that of products. For constrained types, we call the parser for the underlying type τ, bind `x` to the resulting rep and PD, and check whether the constraint is satisfied. The result indicates whether the data has a semantic error and is used in constructing the representation and PD. For example, the PD constructor will add one to the error count if the constraint is not satisfied. Notice that we advance the stream independent of whether the constraint was satisfied.

Sequences have the most complicated semantics because the number of subcomponents depends upon a combination of the data, the termination predicate, and the terminator type. Consequently, the sequence parser uses the function `isDone` and the recursive function `continue` to implement this open-ended semantics. Function `isDone` determines if the parser should terminate by checking whether the end of the source has been reached, the termination condition e has been satisfied, or the terminator type can be read from the stream without errors at ω. Function `continue` takes four arguments: two offsets, a sequence representation, and a sequence PD. The two offsets are the starting and ending offset of the previous round of parsing. They are compared to determine whether the parser is progressing in the source, a check that is critical to ensuring that the parser terminates. Next, the parser checks whether the sequence is finished, and if so, terminates. Otherwise, it attempts to read a separator followed by an element and then continues parsing the sequence with a call to `continue`. Then, the body of the parser creates an initial sequence representation and parse descriptor and then checks whether the sequence described is empty. If not, it reads an element and creates a new rep and PD for the sequence. Note that it passes the PD for `unit` in place of a separator PD, as no separator is read before the first element. Finally, it continues reading the sequence with a call to `continue`.

Because of the iterative nature of sequence parsing, the representation and PD are constructed incrementally. The parser first creates an empty representation and PD and then adds elements to them with each call to `continue`. The error count for an array is the sum of the number of separators with errors plus one if there were any element errors. Therefore, in function $\mathtt{H}_{\mathtt{seq}}$ we first check if the element is the first with an error, setting `eerr` to one if so. Then, the new error count is a sum of the old, potentially one for a separator error, and `eerr`. In $\mathtt{P}_{\mathtt{seq}}$ we calculate the element error count by unconditionally adding one if the element had an error.

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 correspond to arbitrary DDC$^\alpha$ types (of kind T), and, therefore, can have different F_ω types, 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 $\tau_1\,\tau_2$ becomes the application of the interpretation of τ_1 to the representation-type, PD-body type, and parsing-function interpretations of τ_2.

The `scan` type attempts to parse the underlying type from the stream at an increasing offset i from the original offset ω, until success is achieved or the end of the file is reached. In the semantics we give here, offsets are incremented one bit at a time – a practical implementation would choose some larger increment (for example, 32 bits at a time). Note that, upon success, i is passed to the PD constructor function, which both records it in the PD and sets the error code based on it. It is considered a semantic error for the value to be found at a positive i, whereas it is a syntactic error for it not to be found at all.

Notice that the upper-bound on the running time of `scan` is at least linear in the size of the data, depending on the particular argument type. More precisely, if the running time of a type τ is $O(f(n))$, where n is the size of the data, then the running time of $\texttt{scan}(\tau)$ is $O(nf(n))$. While such a running time is potentially high, it is reasonable if it is only incurred for erroneous data, in which case the cost is not incurred on the "fast path" of processing good data; or, if $f(n)$ is 1 and `scan` consumes all of the scanned data, in which case the total running time of the parser is linear in the amount of data consumed, which is the best running time achievable without skipping data. However, we cannot guarantee that either of these conditions are met. The `scan` type can legally appear in branches of sums, in which case the cost could be incurred for valid data (that matches a different branch) without consuming any of the data scanned.

In PADS/C and PADS/ML, we control the potentially high cost of `scan` in two ways. First, we only scan for literals, thereby bounding the running time to linear in the size of the data source. Second, we set a data-source independent maximum on the number of bits scanned for any particular instance of `scan`, rather than potentially scanning until end of the data source. Together, these factors reduce the running time of scanning to $O(1)$. However, the second factor implies that PADS/C and PADS/ML, unlike DDC$^\alpha$, do not guarantee to find the targets of scans, even if they are present in the data source. This difference between DDC$^\alpha$ and the PADS languages could have a significant impact an any guarantees we might make about error recovery based on DDC$^\alpha$ alone. We leave for future work the development of a more sophisticated semantics for `scan` that accounts for the unreliable nature of scans in PADS/C and PADS/ML.

Returning to our discussion of the semantics of DDC$^\alpha$, we note that `compute` only calls the compute constructors without performing any parsing. The representation constructor returns the value computed by e, while the PD records no errors and reports a span of length 0, as no data is consumed by the computation. The `absorb` parser first parses the underlying type and then calls the absorb constructors, passing only the PD, which is needed by the rep constructor to determine whether an error occured while parsing the underlying type. If so, the value returned is a `none`. Otherwise, it is `unit`. The absorb parse descriptor duplicates the error information of its underlying type.

5. METATHEORY

One of the most difficult, and perhaps most interesting, challenges of our work on DDC$^\alpha$ was determining what properties we wanted to hold. What are the "correct" invariants of data description languages? While there are many well-known desirable invariants for programming languages, the metatheory of data description languages has been uncharted.

We present the following two properties as critical invariants of our theory. Just like

$$\frac{}{\tau \to_0 \tau} \qquad \frac{\tau \to \tau' \quad \tau' \to_k \tau''}{\tau \to_{k+1} \tau''}$$

$$\frac{}{e \to_0 e} \qquad \frac{e \to e' \quad e' \to_k e''}{e \to_{k+1} e''}$$

Fig. 19. K-steps normalization and evaluation judgments

the classic Progress and Preservation theorems should hold for any conventional typed programming language, we feel that the following properties should hold, in some form, for any data description language.

—**Parser Type Correctness**: For a DDC$^\alpha$ type τ, the representation and PD output by the parsing function of τ will have the types specified by $[\![\tau]\!]_{\text{rep}}$ and $[\![\tau]\!]_{\text{PD}}$, respectively.

—**Canonical Forms of Parsed Data**: We give a precise characterization of the results of parsers by defining the *canonical forms* of representation-parse descriptor pairs associated with a dependent DDC$^\alpha$ type. Of particular relevance to data description, we show that the errors reported in the parse descriptor will accurately reflect the errors present in the representation.

The aim of this section is to formally state and prove that these critical properties hold for our DDC$^\alpha$ theory. However, before we can do so, we must establish some basic properties of our semantics. We begin with a number of properties that we expect to hold for variable names. First, all variable names introduced by the parsing semantics function should be considered taken from a separate syntactic domain than variables that may appear in ordinary expressions. Therefore, they are by definition "fresh" with respect to any expressions that can be written by the user. Second, for those types with bound variables, the potential alpha-conversion when performing a substitution on the type exactly parallels any alpha-conversion of the same variable where it appears in the translation of the type. Last, all constructors, support functions and base-type parsers are closed with respect to user-defined variable names.

Next, we require that DDC$^\alpha$ base types satisfy the properties that we desire to hold of the rest of the calculus. Below is a formal statement of these requirements. Note that by condition 3, base type parsers must be closed.

Condition 1 (Conditions on Base-types)
(1) $\mathsf{dom}(\mathcal{B}_{kind}) = \mathsf{dom}(\mathcal{B}_{imp})$.
(2) *If* $\mathcal{B}_{kind}(C) = \sigma \to \mathsf{T}$ *then* $\mathcal{B}_{opty}(C) = \sigma \to [\![C(e){:}\mathsf{T}]\!]_{PT}$ *(for any* e *of type* σ*).*
(3) $\vdash \mathcal{B}_{imp}(C) : \mathcal{B}_{opty}(C)$.

The evaluation of F_ω terms and the normalization of DDC$^\alpha$ types are both defined with a small-step semantics. However, it is useful to be able to reason about terms and types that are related by arbitrarily many (k) steps in these semantics, rather than just one. To this end, in Figure 19, we define two judgments that respectively generalize evaluation and normalization to k steps. Next, we state some properties of these judgments.

Lemma 2 (Properties of K-step Evaluation)
(1) *If* $e_1 \to_k e_1'$ *then* $e_1\, e_2 \to_k e_1'\, e_2$.

(2) *If* $e_2 \rightarrow_k e_2'$ *then* $v\, e_2 \rightarrow_k v\, e_2'$.
(3) *If* $e \rightarrow_k e'$ *then* $e\,[\sigma] \rightarrow_k e'\,[\sigma]$.
(4) *If* $e_1 \rightarrow_i e_2$ *and* $e_2 \rightarrow_j e_3$ *then* $e_1 \rightarrow_{(i+j)} e_3$.

PROOF. By induction on the number of steps in evaluation relation. □

Lemma 3 (Properties of K-step Normalization)
(1) *If* $\tau_1 \rightarrow_k \tau_1'$ *then* $\tau_1\,\tau_2 \rightarrow_k \tau_1'\,\tau_2$.
(2) *If* $\tau_2 \rightarrow_k \tau_2'$ *then* $\nu\,\tau_2 \rightarrow_k \nu\,\tau_2'$.
(3) *If* $\tau_1 \rightarrow_k \tau_1'$ *then* $\tau_1\, e \rightarrow_k \tau_1'\, e$.
(4) *If* $e \rightarrow_k e'$ *then* $\nu\, e \rightarrow_k \nu\, e'$.
(5) *If* $\tau_1 \rightarrow_i \tau_2$ *and* $\tau_2 \rightarrow_j \tau_3$ *then* $\tau_1 \rightarrow_{(i+j)} \tau_3$.

PROOF. By induction on the number of steps in evaluation relation. □

Lemma 4 (K-step Evaluation Inversion)
(1) *If* $e_1\, e_2 \rightarrow_k v$ *then* $k > 0$ *and* $\exists\, i, j, v_1, v_2$ *s.t.* $e_1 \rightarrow_i v_1$ *and* $e_2 \rightarrow_j v_2$, *with* $i+j < k$.
(2) *If* $e\,[\sigma] \rightarrow_k v$ *then* $\exists\, i, v'$ *s.t.* $e \rightarrow_i v'$, *with* $i < k$.
(3) *If* $(\texttt{fun}\ f\ x = e)\ v \rightarrow_k v'$ *then* $e[(\texttt{fun}\ f\ x = e)/f][v/x] \rightarrow_{k-1} v'$.
(4) *If* $\texttt{let}\ x = e\ \texttt{in}\, e' \rightarrow_k v$ *then* $\exists\, i, v'$ *s.t.* $e \rightarrow_i v'$ *with* $i < k$.
(5) *If* $\texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \rightarrow_k v$ *and* $e \rightarrow^* \texttt{true}$ *then* $\exists\, i$ *s.t.* $e_1 \rightarrow_i v$ *with* $i < k$.
(6) *If* $\texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \rightarrow_k v$ *and* $e \rightarrow^* \texttt{false}$ *then* $\exists\, i$ *s.t.* $e_2 \rightarrow_i v$ *with* $i < k$.

PROOF. By induction on the number of steps in the evaluation relation. □

Lemma 5 (Confluence of Evaluation)
If $e \rightarrow_k v$ *and* $e \rightarrow_i e'$ *then* $e' \rightarrow_{k-i} v$.

PROOF. By induction on the height of the first derivation, using determinacy of single-step evaluation as needed. □

A number of DDC$^\alpha$ properties involve reasoning about terms that are equivalent up-to equivalent typing annotations. Therefore, we now define this equivalence and state some of its properties.

Definition 6 (Expression Equivalence)
$e \equiv e'$ *iff* e *is syntactically equal to* e' *modulo alpha-conversion of bound variables and equivalence of typing annotations.*

Lemma 7 (Properties of Expression Equivalence)
(1) *If* $e \equiv e'$ *and* $e \rightarrow_k e_1$ *then* $\exists\, e_1'$ *s.t.* $e' \rightarrow_k e_1'$ *and* $e_1 \equiv e_1'$.
(2) *If* $e \equiv e'$ *then* $e_1[e/x] \equiv e_1[e'/x]$.
(3) *If* $\sigma \equiv \sigma'$ *then* $e[\sigma/\alpha] \equiv e[\sigma'/\alpha]$.
(4) $e \equiv e$.
(5) *If* $e \equiv e'$ *then* $e' \equiv e$.
(6) *If* $e \equiv e'$ *and* $e' \equiv e''$ *then* $e \equiv e''$.

PROOF. Part 1. By induction on the number of steps in the evaluation relation. Note that evaluation in F_ω is not influenced by typing annotations. Part 2: By induction on size of e_1. Part 3: By induction on size of e and definition of expression equivalence. Parts 4, 5, 6: By reflexivity, symmetry and transitivity of expression equality and type equivalence. □

Next, we state two properties of F_ω type equivalence that are needed later.

Lemma 8 (Properties of F_ω Type Equivalence)
(1) *If* $\Gamma \vdash \sigma :: \kappa$ *and* $\sigma \equiv \sigma'$ *then* $\Gamma \vdash \sigma' :: \kappa$.
(2) *If* $\Gamma, x{:}\sigma, \Gamma' \vdash e : \sigma_1$ *and* $\sigma \equiv \sigma'$ *then* $\Gamma, x{:}\sigma', \Gamma' \vdash e : \sigma_1$.

Next, we show that substitution commutes with all of the semantic interpretations of DDC$^\alpha$. For clarity, we first introduce two substitution-related abbreviations:

$$\begin{aligned}\langle \tau/\alpha \rangle &= [[\![\tau]\!]_{\text{rep}}/\alpha_{\texttt{rep}}][[\![\tau]\!]_{\text{PDb}}/\alpha_{\texttt{PDb}}]\\ \{\tau/\alpha\} &= [[\![\tau]\!]_{\text{rep}}/\alpha_{\texttt{rep}}][[\![\tau]\!]_{\text{PDb}}/\alpha_{\texttt{PDb}}][[\![\tau]\!]_{\text{P}}/\texttt{parse}_\alpha]\end{aligned}$$

Lemma 9 (Commutativity of Substitution and Semantic Interpretation)
(1) $[\![\tau[\tau'/\alpha]]\!]_{rep} = [\![\tau]\!]_{rep}\langle \tau'/\alpha \rangle$.
(2) *If* $\Delta; \Gamma \vdash \tau : \kappa$ *then* $[\![\tau[\tau'/\alpha]]\!]_{rep} = [\![\tau]\!]_{rep}[[\![\mu\alpha.\tau]\!]_{rep}/\alpha_{\texttt{rep}}]$.
(3) *If* $\exists\, \sigma$ *s.t.* $[\![\tau]\!]_{PD} = \sigma$ *and* $\exists\, \sigma$ *s.t.* $[\![\tau']\!]_{PD} \equiv \texttt{pd\_hdr} * \sigma$ *then* $[\![\tau[\tau'/\alpha]]\!]_{PD} \equiv [\![\tau]\!]_{PD}\langle \tau'/\alpha \rangle = [\![\tau]\!]_{PD}[[\![\tau]\!]_{PDb}/\alpha_{\texttt{PDb}}]$.
(4) *If* $\exists\, \sigma$ *s.t.* $[\![\tau]\!]_{PD} = \sigma$ *and* $\exists\, \sigma$ *s.t.* $[\![\tau']\!]_{PD} \equiv \texttt{pd\_hdr} * \sigma$ *then* $[\![\tau[\tau'/\alpha]]\!]_{P} \equiv [\![\tau]\!]_{P}\{\tau'/\alpha\}$.
(5) $[\![\tau[v/x]]\!]_{rep} = [\![\tau]\!]_{rep}$.
(6) $[\![\tau[v/x]]\!]_{PD} = [\![\tau]\!]_{PD}$.
(7) $[\![\tau[v/x]]\!]_{P} = [\![\tau]\!]_{P}[v/x]$.

PROOF. Parts 1,3-7: By induction on structure of types. Part 2 is proven by induction on the height of the kinding derivation. The most interesting case is `compute`, as it is the only construct in which a variable of the form $\alpha_{\texttt{PDb}}$ might appear. However, as the type is well-formed, we know from the kinding rules that the only type variables allowed in σ are of the form $\alpha_{\texttt{rep}}$. For part 4, note that variables of the form $\texttt{parse}_{\texttt{rep}}$ cannot appear in any τ – they can only be introduced by the parsing semantics function. For part 6, note that the open variables in $[\![\tau]\!]_{\text{P}}$ are exactly those that are open in τ itself, as none are introduced in the translation. □

Next, we prove a similar commutativity result for the $[\![\cdot : \cdot]\!]_{\text{PT}}$ function.

Lemma 10
If $\exists\, \sigma$ *s.t.* $[\![\tau]\!]_{PD} = \sigma$ *and* $\exists\, \sigma$ *s.t.* $[\![\tau']\!]_{PD} \equiv \texttt{pd\_hdr} * \sigma$ *then* $[\![\tau[\tau'/\alpha]{:}\kappa\langle \tau'/\alpha \rangle]\!]_{PT} = [\![\tau{:}\kappa]\!]_{PT}\langle \tau'/\alpha \rangle$.

PROOF. By induction on the size of the kind, using Lemma 9 for T case. □

Lemma 11
The function $[\![\cdot]\!]_{rep}$ *is total.*

PROOF. By induction on the structure of types. □

Next we present some standard type-theoretic results for DDC$^\alpha$ kinding and normalization.

Lemma 12 (DDC$^\alpha$ Preservation)
If $\vdash \tau : \kappa$ *and* $\tau \rightarrow^* \nu$ *then* $\vdash \nu : \kappa$.

PROOF. By induction on the kinding derivation. □

Lemma 13 (DDC$^\alpha$ Inversion)
All kinding rules are invertable.

PROOF. By inspection of the kinding rules. □

Lemma 14 (DDC$^\alpha$ Canonical Forms)
If $\vdash \nu : \kappa$ *then either*

—$\kappa = \mathsf{T}$, *or*
—$\kappa = \sigma \rightarrow \kappa$ *and* $\tau = \lambda x.\tau'$, *or*
—$\kappa = \mathsf{T} \rightarrow \kappa$ *and* $\tau = \lambda \alpha.\tau'$.

PROOF. By kinding rules and grammar of normalized types ν. □

Finally, we state the substitution lemmas that we assume to hold of the various underlying F_ω judgments followed by a substitution lemma for DDC$^\alpha$.

Lemma 15 (F_ω Substitution)
(1) *If* $\vdash \Gamma, \alpha{::}\mathsf{T}, \Gamma'$ ok *and* $\Gamma \vdash \sigma :: \mathsf{T}$ *then* $\vdash \Gamma, \Gamma'[\sigma/\alpha]$ ok.
(2) *If* $\Gamma, \alpha{::}\mathsf{T} \vdash \sigma :: \kappa$ *and* $\Gamma \vdash \sigma_1 :: \mathsf{T}$ *then* $\Gamma \vdash \sigma[\sigma_1/\alpha] :: \mathsf{T}$.
(3) *If* $\Gamma, \alpha{::}\mathsf{T}, \Gamma' \vdash e : \sigma$ *and* $\Gamma \vdash \sigma_1 :: \mathsf{T}$ *then* $\Gamma, \Gamma'[\sigma_1/\alpha] \vdash e[\sigma_1/\alpha] : \sigma[\sigma_1/\alpha]$.
(4) *If* $\Gamma, x{:}\sigma' \vdash e : \sigma$ *and* $\Gamma \vdash v : \sigma'$ *then* $\Gamma \vdash e[v/x] : \sigma$

PROOF. These are standard properties of F_ω. They are all proven by induction on the height of the first derivation. □

Lemma 16 (DDC$^\alpha$ Substitution)
(1) *If* $\Delta; \Gamma, x{:}\sigma \vdash \tau : \kappa$ *and* $[\![\Delta]\!]_{F_\omega}; \Gamma \vdash v : \sigma$ *then* $\Delta; \Gamma \vdash \tau[v/x] : \kappa$.
(2) *If* $\Delta, \alpha{:}\mathsf{T}; \Gamma, \Gamma' \vdash \tau : \kappa$ *and* $\Delta; \Gamma \vdash \tau' : \mathsf{T}$ *then* $\Delta; \Gamma, \Gamma'[\tau'/\alpha] \vdash \tau[\tau'/\alpha] : \kappa[\tau'/\alpha]$.

PROOF. For both parts, by induction on the first derivation, using Lemma 15 as needed. □

Finally, we state another commutativity property for the semantic functions. In essence, it says that evaluation (aka. normalization, type equivalence) commutes with semantic interpretation. This result has inherent value for reasoning about DDC$^\alpha$, as it allows one to reason about the semantics of DDC$^\alpha$ functions directly in terms of the stated normalization rules, rather than indirectly through semantic interpretation and the evaluation/equivalence rules of the semantic domain. Note that the premise of the lemma involves parser evaluation because that is what is needed for later use. We posit without proof that this lemma would be equally true if the second premise were switched with the first conclusion.

Lemma 17 (Commutativity of Evaluation and Semantic Interpretation)
If $\vdash \tau : \kappa$ *and* $[\![\tau]\!]_P \rightarrow^* v$ *then* $\exists\, \nu$ *such that*

(1) $\tau \rightarrow^* \nu$,
(2) $v \equiv [\![\nu]\!]_P$,
(3) $[\![\tau]\!]_{rep} \equiv [\![\nu]\!]_{rep}$, *and*
(4) $[\![\tau]\!]_{PD} \equiv [\![\nu]\!]_{PD}$.

PROOF. By induction on the number of steps in the evaluation. Within the induction, we proceed using a case-by-case analysis of the possible structures of type τ. $\square$

5.1 Type Correctness

Our first key theoretical result is that the various semantic functions we have defined are coherent. In particular, we show that for any well-kinded DDC$^\alpha$ type τ, the corresponding parser is well typed, returning a pair of the corresponding representation and parse descriptor.

Demonstrating that generated parsers 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.

The particular shape required is that every parse descriptor be a pair of a header and an (arbitrary) body. The most straightforward characterization of this property is too weak to prove directly, so we instead characterize it as a logical relation in Definition 18. Lemma 22 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 18
—$\mathrm{H}(\tau : \mathsf{T})$ *iff* $\exists\, \sigma$ *s.t.* $[\![\tau]\!]_{PD} \equiv \texttt{pd\_hdr} * \sigma$.
—$\mathrm{H}(\tau : \mathsf{T} \rightarrow \kappa)$ *iff* $\exists\, \sigma$ *s.t.* $[\![\tau]\!]_{PD} \equiv \sigma$ *and whenever* $\mathrm{H}(\tau' : \mathsf{T})$, *we have* $\mathrm{H}(\tau\, \tau' : \kappa)$.
—$\mathrm{H}(\tau : \sigma \rightarrow \kappa)$ *iff* $\exists\, \sigma'$ *s.t.* $[\![\tau]\!]_{PD} \equiv \sigma'$ *and* $\mathrm{H}(\tau\, e : \kappa)$ *for any expression* e.

Lemma 19
If $\mathrm{H}(\tau : \mathsf{T})$ *then* $\exists\, \sigma\, s.t. [\![\tau]\!]_{PD} = \sigma$.

PROOF. Follows immediately from definition of $\mathrm{H}(\tau : \mathsf{T})$. $\square$

Note that we implicitly demand that $[\![\tau]\!]_{\mathrm{PD}}$ is well defined in the hypothesis of the lemma. We cannot assume that it is well-defined, even for well-formed τ, as that is part of what we are trying to prove.

Lemma 20
If $[\![\tau]\!]_{PD} \equiv [\![\tau']\!]_{PD}$ *then* $\mathrm{H}(\tau : \mathsf{T})$ *iff* $\mathrm{H}(\tau' : \mathsf{T})$.

PROOF. By induction on the structure of the kind. $\square$

Lemma 21
If $\mathrm{H}(\tau : \kappa)$ *and* $\mathrm{H}(\tau' : \mathsf{T})$ *then* $\mathrm{H}(\tau[\tau'/\alpha] : \kappa)$.

PROOF. By induction on the structure of the kind. □

Lemma 22
If $\Delta; \Gamma \vdash \tau : \kappa$ *then* $\mathrm{H}(\tau : \kappa)$.

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

Corollary 23
—*If* $\Delta; \Gamma \vdash \tau : \kappa$ *then* $\exists \sigma. [\![\tau]\!]_{PD} = \sigma$.
—*If* $\Delta; \Gamma \vdash \tau : \mathsf{T}$ *then* $\exists \sigma. [\![\tau]\!]_{PD} \equiv \texttt{pd\_hdr} * \sigma$.

PROOF. Immediate from definition of $\mathrm{H}(\tau : \kappa)$ and Lemma 22. □

We can now prove a general result stating that if a type is well formed, then its type interpretations will be well formed, and that the kind of the type will correspond to the kinds of its interpretations. We first state this correspondence formally and then state and prove the lemma.

Definition 24 (DDC$^\alpha$ Kind Interpretation in F_ω)
—$K(\mathsf{T}) = \mathsf{T}$
—$K(\sigma \to \kappa) = K(\kappa)$
—$K(\mathsf{T} \to \kappa) = \mathsf{T} \to K(\kappa)$

Lemma 25 (Representation-Type Well Formedness)
If $\Delta; \Gamma \vdash \tau : \kappa$ *then*

—$[\![\Delta]\!]_{rep} \vdash [\![\tau]\!]_{rep} :: K(\kappa)$
—$[\![\Delta]\!]_{PD} \vdash [\![\tau]\!]_{PD} :: K(\kappa)$
—*If* $\kappa = \mathsf{T}$ *then* $[\![\Delta]\!]_{PD} \vdash [\![\tau]\!]_{PDb} :: \mathsf{T}$.

PROOF. By induction using Lemma 22 and Lemma 8, part 1. □

We continue by stating and proving that parsers are type correct. However, to do so, we must first establish some typing properties of the representation and parse-descriptor constructors, as at least one of them appears in most parsing functions. In particular, we prove that each constructor produces a value whose type corresponds to its namesake DDC$^\alpha$ type. For clarity, we abbreviate $\texttt{pd\_hdr} * \sigma$ as $\sigma\,\texttt{pd}$.

Lemma 26 (Types of Constructors)
—$\mathrm{R}_{\texttt{unit}} : \texttt{unit} \to \texttt{unit}$
—$\mathrm{P}_{\texttt{unit}} : \texttt{offset} \to \texttt{pd\_hdr} * \texttt{unit}$
—$\mathrm{R}_{\texttt{bottom}} : \texttt{unit} \to \texttt{none}$
—$\mathrm{P}_{\texttt{bottom}} : \texttt{offset} \to \texttt{pd\_hdr} * \texttt{unit}$
—$\mathrm{R}_{\Sigma} : \forall \alpha, \beta. \alpha * \beta \to \alpha * \beta$
—$\mathrm{P}_{\Sigma} : \forall \alpha, \beta. \alpha\,\texttt{pd} * \beta\,\texttt{pd} \to (\alpha\,\texttt{pd} * \beta\,\texttt{pd})\,\texttt{pd}$
—$\mathrm{R}_{+\texttt{left}} : \forall \alpha, \beta. \alpha \to \alpha + \beta$
—$\mathrm{R}_{+\texttt{right}} : \forall \alpha, \beta. \beta \to \alpha + \beta$

—$\mathtt{P}_{\mathtt{+left}} : \forall\alpha, \beta.\alpha\,\mathtt{pd} \to \mathtt{pd\_hdr} * (\alpha\,\mathtt{pd} + \beta\,\mathtt{pd})$

—$\mathtt{P}_{\mathtt{+right}} : \forall\alpha, \beta.\beta\,\mathtt{pd} \to \mathtt{pd\_hdr} * (\alpha\,\mathtt{pd} + \beta\,\mathtt{pd})$

—$\mathsf{R}_{\&} : \forall\alpha, \beta.\alpha * \beta \to \alpha * \beta$

—$\mathsf{P}_{\&} : \forall\alpha, \beta.\alpha\,\mathtt{pd} * \beta\,\mathtt{pd} \to \mathtt{pd\_hdr} * (\alpha\,\mathtt{pd} * \beta\,\mathtt{pd})$.

—$\mathsf{R}_{\mathsf{con}} : \forall\alpha.\mathtt{bool} * \alpha \to \alpha + \alpha$

—$\mathsf{P}_{\mathsf{con}} : \forall\alpha.\mathtt{bool} * \alpha\,\mathtt{pd} \to \mathtt{pd\_hdr} * \alpha\,\mathtt{pd}$

—$\mathsf{R}_{\mathtt{seq\_init}} : \forall\alpha.\mathtt{unit} \to \mathtt{int} * \alpha\,\mathtt{seq}$

—$\mathsf{P}_{\mathtt{seq\_init}} : \forall\alpha.\mathtt{offset} \to \mathtt{pd\_hdr} * (\alpha\,\mathtt{pd}\ \mathtt{arr\_pd})$

—$\mathsf{R}_{\mathtt{seq}} : \forall\alpha.(\mathtt{int} * \alpha\,\mathtt{seq}) * \alpha \to \mathtt{int} * \alpha\,\mathtt{seq}$

—$\mathsf{P}_{\mathtt{seq}} : \forall\alpha_{elt}, \alpha_{sep}.(\mathtt{pd\_hdr} * (\alpha_{elt}\,\mathtt{pd}\ \mathtt{arr\_pd})) * \alpha_{sep}\,\mathtt{pd} * \alpha_{elt}\,\mathtt{pd} \to$
$\mathtt{pd\_hdr} * (\alpha_{elt}\,\mathtt{pd}\ \mathtt{arr\_pd})$

—$\mathsf{R}_{\mathtt{compute}} : \forall\alpha.\alpha \to \alpha$

—$\mathsf{P}_{\mathtt{compute}} : \mathtt{offset} \to \mathtt{pd\_hdr} * \mathtt{unit}$

—$\mathsf{R}_{\mathtt{absorb}} : \forall\alpha.\alpha\,\mathtt{pd} \to \mathtt{unit} + \mathtt{none}$

—$\mathsf{P}_{\mathtt{absorb}} : \forall\alpha.\alpha\,\mathtt{pd} \to \mathtt{pd\_hdr} * \mathtt{unit}$

—$\mathsf{R}_{\mathtt{scan}} : \forall\alpha.\alpha \to \alpha + \mathtt{none}$

—$\mathsf{P}_{\mathtt{scan}} : \forall\alpha.\mathtt{int} * \alpha\,\mathtt{pd} \to \mathtt{pd\_hdr} * ((\mathtt{int} * \alpha\,\mathtt{pd}) + \mathtt{unit})$

—$\mathsf{R}_{\mathtt{scan\_err}} : \forall\alpha.\mathtt{unit} \to \alpha + \mathtt{none}$

—$\mathsf{P}_{\mathtt{scan\_err}} : \forall\alpha.\mathtt{offset} \to \mathtt{pd\_hdr} * ((\mathtt{int} * \alpha) + \mathtt{unit})$

PROOF. By typing rules of F_ω. □

With our next lemma, we establish the type correctness of the generated parsers. We prove the lemma using a 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 function $[\![\Delta]\!]_{\mathrm{PT}}$ which maps the set of type-variable bindings in a DDC$^\alpha$ context Δ to a corresponding set of function-variable bindings in an F_ω context Γ.

$$[\![\,\cdot\,]\!]_{\mathrm{PT}} = \cdot \qquad [\![\Delta, \alpha{:}\mathsf{T}]\!]_{\mathrm{PT}} = [\![\Delta]\!]_{\mathrm{PT}}, \mathtt{parse}_\alpha{:}[\![\alpha{:}\mathsf{T}]\!]_{\mathrm{PT}}$$

Lemma 27 (Type Correctness Lemma)
If $\Delta; \Gamma \vdash \tau : \kappa$ *then* $[\![\Delta]\!]_{F_\omega}, \Gamma, [\![\Delta]\!]_{PT} \vdash [\![\tau]\!]_P : [\![\tau{:}\kappa]\!]_{PT}$

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

Theorem 28 (Type Correctness of Closed Types)
If $\vdash \tau : \kappa$ *then* $\vdash [\![\tau]\!]_P : [\![\tau{:}\kappa]\!]_{PT}$.

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.

5.2 Canonical Forms

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 reports that 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 dependent DDC$^\alpha$ type description. When a parse descriptor and representation satisfy these invariants and correspond properly, we say the pair of data structures is *canonical* or in *canonical form*.

For each DDC$^\alpha$ type, its canonical forms are defined via two (mutually recursive) relations. The first, $\mathrm{Canon}_\nu(r,p)$, defines the canonical form of a representation r and a parse descriptor p at normal type ν. It is defined for all closed normal types ν with base kind T. Types with higher kind such as abstractions are excluded from this definition as they cannot directly produce representations and PDs.

A second definition, $\mathrm{Canon}^*{}_\tau(r,p)$ normalizes τ to a ν, thereby eliminating outermost type and value applications. Then, the requirements on ν are given by $\mathrm{Canon}_\nu(r,p)$. For brevity, 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.

Definition 29 (Canonical Forms I)
$\mathrm{Canon}_\nu(r,p)$ *iff exactly one of the following is true:*

—$\nu = \texttt{unit}$ *and* $r = ()$ *and* $p.nerr = 0$.

—$\nu = \texttt{bottom}$ *and* $r = \texttt{none}$ *and* $p.nerr = 1$.

—$\nu = C(e)$ *and* $r = \texttt{inl}\ c$ *and* $p.nerr = 0$.

—$\nu = C(e)$ *and* $r = \texttt{inr}\ \texttt{none}$ *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* $h.nerr = \texttt{pos}(p_1.nerr) + \texttt{pos}(p_2.nerr)$, $\mathrm{Canon}^*{}_{\tau_1}(r_1, p_1)$ *and* $\mathrm{Canon}^*{}_{\tau_2[(r,p)/x]}(r_2, p_2)$.

—$\nu = \tau_1 + \tau_2$ *and* $r = \texttt{inl}\ r'$ *and* $p = (h, \texttt{inl}\ p')$ *and* $h.nerr = \texttt{pos}(p'.nerr)$ *and* $\mathrm{Canon}^*{}_{\tau_1}(r', p')$.

—$\nu = \tau_1 + \tau_2$ *and* $r = \texttt{inr}\ r'$ *and* $p = (h, \texttt{inr}\ p')$ *and* $h.nerr = \texttt{pos}(p'.nerr)$ *and* $\mathrm{Canon}^*{}_{\tau_2}(r', p')$.

—$\nu = \tau_1 \,\&\, \tau_2$, $r = (r_1, r_2)$ *and* $p = (h, (p_1, p_2))$, *and* $h.nerr = \texttt{pos}(p_1.nerr) + \texttt{pos}(p_2.nerr)$, $\mathrm{Canon}^*{}_{\tau_1}(r_1, p_1)$ *and* $\mathrm{Canon}^*{}_{\tau_2}(r_2, p_2)$.

—$\nu = \{x{:}\tau' \mid e\}$, $r = \texttt{inl}\ r'$ *and* $p = (h, p')$, *and* $h.nerr = \texttt{pos}(p'.nerr)$, $\mathrm{Canon}^*{}_{\tau'}(r', p')$ *and* $e[(r', p')/x] \rightarrow^* \texttt{true}$.

—$\nu = \{x{:}\tau' \mid e\}$, $r = \texttt{inr}\ r'$ *and* $p = (h, p')$, *and* $h.nerr = 1 + \texttt{pos}(p'.nerr)$, $\mathrm{Canon}^*{}_{\tau'}(r', p')$ *and* $e[(r', p')/x] \rightarrow^* \texttt{false}$.

—$\nu = \tau_e\ \texttt{seq}(\tau_s, e, \tau_t,)$, $r = (len, [\vec{r_i}])$, $p = (h, (neerr, len', [\vec{p_i}]))$, $neerr = \sum_{i=1}^{len} \texttt{pos}(p_i.nerr)$, $len = len'$, $\mathrm{Canon}^*{}_{\tau_e}(r_i, p_i)$ *(for* $i = 1 \ldots len$*), and* $h.nerr \geq \texttt{pos}(neerr)$.

—$\nu = \mu\alpha.\tau'$, $r = \texttt{fold}[[\![\mu\alpha.\tau']\!]_{rep}]\, r'$, $p = (h, \texttt{fold}[[\![\mu\alpha.\tau']\!]_{PD}]\, p')$, $p.nerr = p'.nerr$ *and*
$\mathrm{Canon}^*{}_{\tau'[\mu\alpha.\tau'/\alpha]}(r', p')$.

—$\nu = \texttt{compute}(e{:}\sigma)$ *and* $p.nerr = 0$.

—$\nu = \texttt{absorb}(\tau')$, $r = \texttt{inl}\ ()$, *and* $p.nerr = 0$.

—$\nu = \texttt{absorb}(\tau')$, $r = \texttt{inr}\ \texttt{none}$, *and* $p.nerr > 0$.

—$\nu = \texttt{scan}(\tau')$, $r = \texttt{inl}\ r'$, $p = (h, \texttt{inl}\ (i, p'))$, $h.nerr = \texttt{pos}(i) + \texttt{pos}(p'.nerr)$, *and* $\mathrm{Canon}^*{}_{\tau'}(r', p')$.

—$\nu = \texttt{scan}(\tau')$, $r = \texttt{inr}\ \texttt{none}$, $p = (h, \texttt{inr}\ ())$, *and* $h.nerr = 1$.

Definition 30 (Canonical Forms II)
$\mathrm{Canon}^*{}_{\tau}(r, p)$ *iff* $\tau \rightarrow^* \nu$ *and* $\mathrm{Canon}_{\nu}(r, p)$.

We first prove that the representation and parse-descriptor constructors, under the appropriate conditions, produce values in canonical form.

Lemma 31 (Constructors Produce Values in Canonical Form)

—$\mathrm{Canon}_{\texttt{unit}}(\mathtt{R}_{\texttt{true}}(), \mathtt{P}_{\texttt{true}}(\omega))$.

—$\mathrm{Canon}_{\texttt{bottom}}(\mathtt{R}_{\texttt{false}}(), \mathtt{P}_{\texttt{false}}(\omega))$.

—*If* $\mathrm{Canon}^*{}_{\tau_1}(r_1, p_1)$ *and* $\mathrm{Canon}^*{}_{\tau_2[(r,p)/x]}(r_2, p_2)$ *then* $\mathrm{Canon}_{\Sigma\, x:\tau_1.\tau_2}(\mathtt{R}_{\Sigma}(\mathtt{r}_1, \mathtt{r}_2), \mathtt{P}_{\Sigma}(\mathtt{p}_1, \mathtt{p}_2))$.

—*If* $\mathrm{Canon}^*{}_{\tau}(r, p)$ *then* $\mathrm{Canon}_{\tau+\tau'}(\mathtt{R}_{+\texttt{left}}(\mathtt{r}), \mathtt{P}_{+\texttt{left}}(\mathtt{p}))$.

—*If* $\mathrm{Canon}^*{}_{\tau}(r, p)$ *then* $\mathrm{Canon}_{\tau'+\tau}(\mathtt{R}_{+\texttt{right}}(\mathtt{r}), \mathtt{P}_{+\texttt{right}}(\mathtt{p}))$.

—*If* $\mathrm{Canon}^*{}_{\tau_1}(r_1, p_1)$ *and* $\mathrm{Canon}^*{}_{\tau_2}(r_2, p_2)$ *then* $\mathrm{Canon}_{\tau_1\,\&\,\tau_2}(\mathtt{R}_{\&}(\mathtt{r}_1, \mathtt{r}_2), \mathtt{P}_{\&}(\mathtt{p}_1, \mathtt{p}_2))$.

—*If* $\mathrm{Canon}^*{}_{\tau}(r, p)$ *and* $e[(r, p)/x] \rightarrow^* c$ *then* $\mathrm{Canon}_{\{x:\tau\,|\,e\}}(\mathtt{R}_{\texttt{set}}(\mathtt{c}, \mathtt{r}), \mathtt{P}_{\texttt{set}}(\mathtt{c}, \mathtt{p}))$

—$\mathrm{Canon}_{\tau\ \texttt{seq}(\tau_s, e, \tau_t)}(\mathtt{R}_{\texttt{seq\_init}}(), \mathtt{P}_{\texttt{seq\_init}}(\omega))$.

—*If* $\mathrm{Canon}_{\tau\ \texttt{seq}(\tau_s, e, \tau_t)}(r, p)$ *and* $\mathrm{Canon}^*{}_{\tau}(r', p')$ *then, for any* p'', $\mathrm{Canon}_{\tau\ \texttt{seq}(\tau_s, e, \tau_t)}(\mathtt{R}_{\texttt{seq}}(\mathtt{r}, \mathtt{r}'), \mathtt{P}_{\texttt{seq}}(\mathtt{p}, \mathtt{p}'', \mathtt{p}'))$.

—$\mathrm{Canon}_{\texttt{compute}(e:\sigma)}(\mathtt{R}_{\texttt{compute}}(\mathtt{e}), \mathtt{P}_{\texttt{compute}}(\omega))$.

—$\mathrm{Canon}_{\texttt{absorb}(\tau)}(\mathtt{R}_{\texttt{absorb}}(\mathtt{p}), \mathtt{P}_{\texttt{absorb}}(\mathtt{p}))$.

—*If* $\mathrm{Canon}^*{}_{\tau}(r, p)$ *then* $\mathrm{Canon}_{\texttt{scan}(\tau)}(\mathtt{R}_{\texttt{scan}}(\mathtt{r}), \mathtt{P}_{\texttt{scan}}(\mathtt{i}, \mathtt{p}))$.

—$\mathrm{Canon}_{\texttt{scan}(\tau)}(\mathtt{R}_{\texttt{scan\_err}}(), \mathtt{P}_{\texttt{scan\_err}}(\omega))$.

PROOF. By inspection of the constructor functions. □

In addition, we require that base-type parsers produce values in canonical form:

Condition 32 (Base Type Parsers Produce Values in Canonical Form)
If $\vdash v : \sigma$, $\mathcal{B}_{\mathit{kind}}(C) = \sigma \rightarrow \mathsf{T}$ *and* $\mathcal{B}_{\mathit{imp}}(C)\ v\ (B, \omega) \rightarrow^* (\omega', r, p)$ *then* $\mathrm{Canon}_{C(v)}(r, p)$.

Lemma 33 states that the parsers for well-formed types (of base kind) will produce a canonical pair of representation and parse descriptor, if they produce anything at all.

Lemma 33 (Parsing to Canonical Forms)
If $\vdash \tau : \mathsf{T}$ *and* $[\![\tau]\!]_P\ (B, \omega) \rightarrow^* (\omega', r, p)$ *then* $\mathrm{Canon}^*{}_{\tau}(r, p)$.

PROOF. By induction on the height of the second derivation – that is, the number of steps taken to evaluate. Within the induction, we proceed using a case-by-case analysis of the possible structures of type τ. □

$$\boxed{prog \Downarrow \tau \text{ prog}}$$

$$\frac{t \Downarrow \tau}{t \Downarrow \tau \text{ prog}}\ \textsc{Prog-One} \qquad \frac{p[t/\alpha] \Downarrow \tau \text{ prog}}{\alpha = t;\ p \Downarrow \tau \text{ prog}}\ \textsc{Prog-Def} \qquad \frac{p[\mathbf{\texttt{Prec}}\ \alpha.t/\alpha] \Downarrow \tau \text{ prog}}{\mathbf{\texttt{Prec}}\ \alpha = t;\ p \Downarrow \tau \text{ prog}}\ \textsc{Prog-RecDef}$$

$$\boxed{t \Downarrow \tau}$$

$$\frac{}{C(e) \Downarrow C(e)}\ \textsc{Base} \qquad \frac{t \Downarrow \tau}{\mathbf{\texttt{Pfun}}(x : \sigma) = t \Downarrow \lambda x.\tau}\ \textsc{Pfun} \qquad \frac{t \Downarrow \tau}{t\ e \Downarrow \tau\ e}\ \textsc{App}$$

$$\frac{t_i \Downarrow \tau_i}{\begin{array}{c}\mathbf{\texttt{Pstruct}}\{x_1{:}t_1 \ldots x_n{:}t_n\} \Downarrow \\ \Sigma\, x_1{:}\tau_1.\cdots\Sigma\, x_{n-1}{:}\tau_{n-1}.\tau_n\end{array}}\ \textsc{Pstruct} \qquad \frac{t_i \Downarrow \tau_i}{\begin{array}{c}\mathbf{\texttt{Punion}}\{x_1{:}t_1 \ldots x_n{:}t_n\} \Downarrow \\ \tau_1 + \cdots + \tau_n + \texttt{bottom}\end{array}}\ \textsc{Punion}$$

$$\frac{t_i \Downarrow \tau_i}{\mathbf{\texttt{Palt}}\{x_1{:}t_1 \ldots x_n{:}t_n\} \Downarrow \tau_1 \& \ldots \& \tau_n}\ \textsc{Palt} \qquad \frac{t \Downarrow \tau}{\mathbf{\texttt{Popt}}\ t \Downarrow \tau + \texttt{unit}}\ \textsc{Popt}$$

$$\frac{t \Downarrow \tau}{t\ \mathbf{\texttt{Pwhere}}\, x.e \Downarrow \{x{:}\tau \mid \texttt{if isOk}(x.\texttt{pd})\ \texttt{then}\ e\ \texttt{else true}\}}\ \textsc{Pwhere}$$

$$\frac{t \Downarrow \tau \quad t_{sep} \Downarrow \tau_s \quad t_{term} \Downarrow \tau_t \quad (f = \lambda \texttt{x.false})}{t\ \mathbf{\texttt{Parray}}(t_{sep}, t_{term}) \Downarrow \tau\ \texttt{seq}(\texttt{scan}(\tau_s), f, \tau_t)}\ \textsc{Parray} \qquad \frac{}{\mathbf{\texttt{Pcompute}}\ e{:}\sigma \Downarrow \texttt{compute}(e{:}\sigma)}\ \textsc{Pcompute}$$

$$\frac{\text{Ty}(c) = \tau}{\mathbf{\texttt{Plit}}\ c \Downarrow \texttt{scan}(\texttt{absorb}(\{x{:}\tau \mid x = c\}))}\ \textsc{Plit} \qquad \frac{}{\alpha \Downarrow \alpha}\ \textsc{Var} \qquad \frac{t \Downarrow \tau}{\mathbf{\texttt{Prec}}\ \alpha.t \Downarrow \mu\alpha.\tau}\ \textsc{Prec}$$

Fig. 20. Encoding IPADS in DDC$^\alpha$

Corollary 34
If $\text{Canon}^*{}_\tau(r, p)$ *and* $p.h.nerr = 0$ *then there are no syntactic or semantic errors in the representation data structure* r.

This corollary is important as it ensures that a single check is sufficient to verify the validity of a data structure. Only if the data structure is not valid will further checking of substructures be required.

6. ENCODING DDLS IN DDC$^\alpha$

We can better understand data description languages by elaborating their constructs into the types of DDC$^\alpha$. We start by specifying the complete elaboration of IPADS into DDC$^\alpha$. We then discuss other features of PADS/C, PADS/ML, DATASCRIPT, and PACKETTYPES that are not found in IPADS. Finally, we briefly discuss some limitations of DDC$^\alpha$.

6.1 IPADS Elaboration

We specify the elaboration from IPADS to DDC$^\alpha$ with two judgments: $p \Downarrow \tau$ prog indicates that the IPADS program p is encoded as DDC$^\alpha$ type τ, while $t \Downarrow \tau$ does the same for IPADS types t. These judgments are defined in Figure 20.

As much of the elaboration is straightforward, we mention only a few important points. Notice we add `bottom` as the last branch of the DDC$^\alpha$ sum when elaborating **`Punion`** so that the parse will fail if none of the branches match rather than returning the result of the last branch. We base this behavior directly on the actual PADS/C language. In the elaboration of **`Pwhere`**, we only check the constraint if the underlying value parses with no errors. For **`Parray`**s, we add simple error recovery by scanning for the separator type. This behavior allows us to easily skip erroneous elements. We use the `scan` type in the same way for **`Plit`**, as literals often appear as field separators in **`Pstruct`**s. We also absorb the literal, as its value is known statically. We use the function $\mathrm{Ty}(c)$ to determine the correct type for the particular literal. For example, a string literal would require a **`Pstring`** type.

6.2 Beyond IPADS

We now give semantics to four features not found in IPADS: PADS/C switched unions, PADS/ML polymorphic, recursive datatypes, DATASCRIPT arrays, and PACKETTYPES overlays.

PADS/C *switched unions.* A switched union, like a **`Punion`**, indicates variability in the data format with a set of alternative formats (branches). However, instead of trying each branch in turn, the switched union takes an expression that determines which branch to use. Typically, this expression depends upon data read earlier in the parse. Each branch is preceded by a tag, and the first branch whose tag matches the expression is selected. If none match then the default branch t_{def} is chosen. The syntax of a switched union is **`Pswitch`** $e\ \{\overrightarrow{e \Rightarrow x{:}t}\ t_{\mathrm{def}}\}$.

To aid in our elaboration of **`Pswitch`**, we define a type `if` e `then` t_1 `else` t_2 that allows us to choose between two types conditionally:

$$\frac{t_1 \Downarrow \tau_1 \quad t_2 \Downarrow \tau_2 \quad (c = \texttt{compute}(\texttt{if}\ e\ \texttt{then 1 else 2 :Pint}))}{\texttt{if}\ e\ \texttt{then}\ t_1\ \texttt{else}\ t_2 \Downarrow c * (\{x{:}\texttt{unit} \mid \texttt{not}\ e\} + \tau_1)\ \&\ (\{x{:}\texttt{unit} \mid e\} + \tau_2)}$$

The computed value c records which branch of the conditional is selected. If the condition e is true, c will be 1, the left-hand side of the intesection will parse τ_1 and the right will parse nothing. Otherwise, c will be 2, the left-hand side will parse nothing and the right τ_2.

Now, we can encode **`Pswitch`** as syntactic sugar for a series of cascading conditional types.

$$\begin{array}{l} \textbf{Pswitch}\ e\ \{ \\ \quad e_1 \Rightarrow x_1{:}t_1 \\ \quad \ldots \\ \quad e_n \Rightarrow x_n{:}t_n \\ \quad t_{\mathrm{def}}\} \end{array} = \begin{array}{l} \texttt{if}\ e = e_1\ \texttt{then}\ t_1\ \texttt{else} \\ \ldots \\ \texttt{if}\ e = e_n\ \texttt{then}\ t_1\ \texttt{else} \\ t_{\mathrm{def}} \end{array}$$

Note that we can safely replicate e as the host language is pure.

PADS/ML *polymorphic, recursive datatypes.* We have also developed an encoding of PADS/ML's polymorphic, recursive datatypes. We present this encoding in two steps. First, we extend IPADS with type abstraction and application, and specify their elobaration into DDC$^{\alpha}$. Notice that IPADS type abstractions can have multiple parameters.

$$\text{Types}\ \ t\ ::=\ \ldots \mid \textbf{PFun}\ (\overrightarrow{\alpha}) = t \mid t\ (\overrightarrow{t})$$

$$\frac{t \Downarrow \tau}{\texttt{PFun}(\overrightarrow{\alpha}) = t \Downarrow \overrightarrow{\lambda\, \alpha}.\tau} \qquad \frac{t \Downarrow \tau \qquad \overrightarrow{t \Downarrow \tau}}{t\ (\overrightarrow{t}\ ) \ \Downarrow \tau\ \overrightarrow{\tau}}$$

Next, we extend IPADS programs to include datatype bindings. Datatype bindings include the name of the type, α, a list of type parameters ($\overrightarrow{\alpha}$), a single value parameter x, and a body that consists of a list of named variants. As with **Prec** bindings, we do not specify the meaning of datatype bindings in DDC$^\alpha$ directly. Rather, we decompose a given datatype into a compound IPADS type, which is then substituted into the remainder of the program.

$$\text{Programs}\ \ p\ ::=\ ...\ |\ \texttt{Pdatatype}\ \alpha\ \ (\overrightarrow{\alpha})(x:\sigma) = \{\overrightarrow{x{:}t}\};\ p$$

$$\frac{p[t'/\alpha] \Downarrow \tau \text{ prog} \qquad (t' = \texttt{PFun}\ (\overrightarrow{\alpha}) =\ \texttt{Pfun}(x:\sigma) = \texttt{Prec}\ \alpha.\,\texttt{Punion}\{\overrightarrow{x{:}t}\})}{\texttt{Pdatatype}\ \alpha\ (\overrightarrow{\alpha})\ (x:\sigma) = \{\overrightarrow{x{:}t}\};\ p \Downarrow \tau \text{ prog}}$$

There are two important points to notice about the decomposition. First, a datatype is decomposed into no less than four IPADS (and, by extension, DDC$^\alpha$) types. Second, and more subtly, the recursive type is nested inside of the abstractions, thereby preventing the definition of nonuniform datatypes. Indeed, the name of the bound datatype, α, plays two distinct roles – within the recursive type, it is a monomorphic type referring only to the recursive type itself, while within the rest of the program it is a polymorphic type referring to the entire type abstraction.

DATASCRIPT *arrays.* Next, we introduce DATASCRIPT-style arrays $t\ [\mathit{length}]$, used to describe binary data. They are parameterized by an optional length field, instead of a separator and terminator. If the user supplies the length of the sequence, the array parser reads exactly that number of elements. Arrays with the length field specfied can be encoded in a straightforward manner with DDC$^\alpha$ sequences:

$$\frac{t \Downarrow \tau \quad (f = \lambda((\texttt{len}, \texttt{elts}), \texttt{p}).\texttt{len} = \mathit{length})}{t\ [\mathit{length}] \Downarrow \tau\ \texttt{seq}(\texttt{unit}, f, \texttt{bottom})}$$

As these arrays have neither separators nor terminators, we use `unit` (always succeeds, parsing nothing) and `bottom` (always fails, parsing nothing), respectively, for separator and terminator. The function f takes a pair of array representation and PD and compares the sequence length recorded in the representation to length.

Arrays of unspecified length are more difficult to encode as they must check the next element for parse errors without consuming it from the data stream. A termination predicate cannot encode this check as they cannot perform lookahead. Therefore, we must use the terminator type to look ahead for an element parse error. For this purpose, we construct a type which succeeds where τ fails and fails where τ succeeds:

$$\{x{:}\tau + \texttt{unit}\ |\ \texttt{case}\ x.\mathit{rep}\ \texttt{of}\ (\texttt{inl}\ \_ \Rightarrow \texttt{false}\ \mid\ \texttt{inr}\ \_ \Rightarrow \texttt{true})\}$$

Abbreviated $\texttt{not}(\tau)$, this type attempts to parse a τ. On success, the representation will be a left injection. The constraint in the constrained type will therefore fail. If a τ cannot be parsed, the sum will default to `unit`, the rep will be a right injection, and the constraint will succeed. The use of the sum in the underlying type is critical as it allows the constrained type to be error free even when parsing τ fails.

With not, we can encode the unbounded DATASCRIPT array as follows:

$$\frac{t \Downarrow \tau}{t\ [] \Downarrow \tau\ \texttt{seq}(\texttt{unit}, \lambda \texttt{x.false}, \texttt{not}(\tau))}$$

Note that the termination predicate is trivially false, as we use the lookahead-terminator exclusively to terminate the array.

PACKETTYPES overlays. Finally, we consider the *overlay* construct found in PACKETTYPES. An overlay allows description authors "to merge two type specifications by embedding one within the other, as is done when one protocol is *encapsulated* within another. Overlay[s] introduce additional substructure to an already existing field." [McCann and Chandra 2000]. For example, consider a network packet from a fictional protocol FP, where the packet body is represented as a simple byte-array.

```
FPPacket = Pstruct {
  header : FPHeader;
  body   : Pbyte Parray(Pnosep,Peof);
}
IPinFP = Poverlay FPPacket.body with IPPacket
```

Type **Pnosep** indicates that there are no separators between elements of the byte array and type **Peof** indicates that the array is terminated by the end-of-file. They can be encoded in DDC$^\alpha$ using unit and bottom, respectively. The overlay creates a new type IPinFP where the body field is an IPPacket rather than a simple byte array.

We have developed an elaboration of the overlay syntax into DDC$^\alpha$. In essence, overlays are syntactic sugar: overlaying a subfield of a given type replaces the type of that subfield with a new type. However, despite the essentially syntactic nature of overlays, we discovered a critical subtlety of semantic significance, not mentioned by the PACKETTYPES authors. Any expressions in the original type that refer to the overlayed field may no longer be well typed after applying the overlay. For example, consider extending FPPacket with a field that is constrained to be equal to the checksum of the body:

```
FPPacket = Pstruct {
  header   : FPHeader;
  body     : Pbyte Parray(Pnosep,Peof);
  checksum : Pint Pwhere cs.cs = checksum(body);
}
```

The checksum function requires that body be a byte array. Therefore, if we overlay body with a structured type like IPPacket, then body will no longer be a byte array and, so, the application of checksum to body will be ill-formed. We thought to disallow such expressions in the overlayed type. However, we found this to be a difficult, if not impossible task. More importantly, such a restriction is unnecessary. Instead, the new type can be checked for well formedness after the overlay process, an easy task in the DDC$^\alpha$ framework.

At this point, we have described the elaborations of some of the more interesting features of the languages that we have studied. However, to give a fuller sense of what is possible, we briefly list additional features of DATASCRIPT and PACKETTYPES for which we have found encodings in DDC$^\alpha$:

—PACKETTYPES: arrays, where clauses, structures, overlays, and alternation.

—DATASCRIPT: constrained types (enumerations and bitmask sets), value-parameterized types (which they refer to as "type parameters"), arrays, constraints, and (monotonically increasing) labels. These labels allow users to specify the location of a data element within the data source. They can be used, for example, to describe a data source that begins with a header specifying the location of the remaining data elements in the data source.

We know of a couple of features from data description languages that we cannot implement in DDC$^\alpha$ as it stands. First, we cannot support a label construct that permits the user to rewind the input. Second, DATASCRIPT allows the element type of an array to reference the representation of the array itself [Back 2002] (see, in particular, the example in Figure 5). This feature can be useful, for example, if the element type needs the index of the array element that is currently being processed. DDC$^\alpha$ does not support this kind of element-type parameterization. However, we do not view such limitations as particularly troublesome. Like the λ-calculus or π-calculus, DDC$^\alpha$ is intended to capture many common language features, while providing a convenient and effective basis for extension with new features.

7. APPLICATIONS OF THE SEMANTICS

The development of DDC$^\alpha$ and definition of a semantics for IPADS has had a substantial impact on the PADS/C and PADS/ML implementations. It has helped improve the implementations in a number of distinct ways, which we now discuss.

7.1 Bug Hunting

The DDC$^\alpha$ was developed, in part, through a line-by-line analysis of key portions of the PADS/C implementation, to uncover implicit invariants in the code. In the process of trying to understand and formalize these invariants we realized that our error accounting methodology was inconsistent, particularly in the case of arrays. When we realized the problem, we were able to formulate a clear rule to apply universally: each subcomponent adds 1 to the error count of its parent if and only if it has errors. If we had not tried to formalize our semantics, it is unlikely we would have made the error accounting rule precise, leaving our implementation buggy and inconsistent.

The semantics also helped us avoid potential nontermination of array parsers. In the original implementation of PADS/C arrays, it was possible to write nonterminating arrays, a bug that was only uncovered when it hung a real program. We have fixed the bug and used the semantics to verify our fix. [2]

7.2 Principled Language Implementation

Unlike the rest of PADS/C, the semantics of recursive types preceded the implementation. We used the semantics to guide our design decisions in the implementation. Perhaps more significantly, the semantics was used in its entirety to guide the implementation of PADS/ML. The semantics of type abstractions were particularly helpful, as they are a new feature not found in PADS/C. Before working through the formal semantics, we struggled

[2]The type `nothing array(nothing,eof)` where type `nothing` consumes no input, would not terminate in the orignal system. A careful read of the DDC$^\alpha$ semantics of arrays, which has now been implemented in PADS/C, shows that array parsing terminates after an iteration in which the array parser reads nothing.

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. We hope the calculus will serve as a guide for implementations of PADS in other host languages.

7.3 Distinguishing the Essential from the Accidental

In his 1965 paper, P.J. Landin asks "Do the idiosyncracies [of a language] reflect basic logical properties of the situations that are being catered for? Or are they accidents of history and personal background that may be obscuring fruitful developments?"

The semantics helped us answer this question with regard to the **`Pomit`** and **`Pcompute`** qualifiers of PADS/C. Originally, these qualifiers were only intended to be used on fields within **`Pstruct`**s. By an accident of the implementation, they appeared in **`Punion`**s as well, but spread no further. However, when designing DDC$^\alpha$, we followed the *principle of orthogonality*, which suggests that every linguistic concept be defined independently of every other. In particular, we observed that "omitting" data from, or including ("computing") data in, the internal representation is not dependent upon the idea of structures or unions. Furthermore, we found that developing these concepts as first-class constructors `absorb` and `compute` in DDC$^\alpha$ allowed us to encode the semantics of other PADS/C features elegantly (literals, for example). In this case, then, the DDC$^\alpha$ highlighed that the restriction of **`Pomit`** and **`Pcompute`** to mere type qualifers for **`Punion`** and **`Pstruct`** fields was an "accident of history," rather than a "basic logical property" of data description.

We conclude with an example of another feature to which Landin's question applies, but for which we do not yet know the answer. The **`Punion`** construct chooses between branches by searching for the first one without errors. However, this semantics ignores situations in which the correct branch in fact has errors. Often, this behavior will lead to parsing nothing and flagging a failure, rather than parsing the correct branch to the best of its ability. The process of developing a semantics brought this fact to our attention and it now seems clear we would like a more robust **`Punion`**, but we are not currently sure how to design one.

8. RELATED WORK

The primary purpose of this article is to describe the semantic theory of type-based data description languages. However, in the following paragraphs, we give an overview both of research in related theoretical topics and in implementation of practical technologies for managing ad hoc data.

Ad Hoc Data Description Languages. Clearly, the most closely related language designs are PADS/C [Fisher and Gruber 2005], which has data descriptions based on the type structure and syntax of the C programming language, and PADS/ML [Mandelbaum et al. 2007], which has data descriptions based on the type structure and syntax of O'Caml. As discussed in previous sections, PADS/C was first developed prior to the theory described in this paper, but then vetted and improved using the theory as a guide. On the other hand, PADS/ML was developed later, and the implementation built relatively directly by transcription from the formal inference rules. Both languages are capable of generating parser and printer libraries as well as a number of useful stand-alone tools for query support, format translation, and analysis of statistical properties.

The networking community has developed a number of domain-specific languages that use a type-based model for describing data much like PADS/C, PADS/ML, and DDC$^\alpha$. These include PACKETTYPES [McCann and Chandra 2000], DATASCRIPT [Back 2002] and Bro's [Paxson 1999]. These languages only handle binary data as they are primarily aimed at packet processing applications. As we suggested earlier in this article, DDC$^\alpha$ will serve as a useful platform for studying many of the features of these languages.

A somewhat different class of languages includes ASDL [ASDL] and ASN.1 [Dubuisson 2001] . Both of these systems specify the *logical* in-memory representation of data and then automatically generate a *physical* on-disk representation. Although useful for many purposes, this technology does not help process data that arrives in predetermined, ad hoc formats. Another language in this category is the Hierarchical Data Format 5 (HDF5) [Hierarchical Data Format 5]. This file format allows users to store scientific data, but it does not help users deal with legacy ad hoc formats.

At the other end of the spectrum, some of the oldest tools for describing data formats are parser generators for compiler construction such as LEX and YACC. While excellent for parsing programming languages, LEX and YACC are too heavyweight for parsing many of the simpler ad hoc data formats that arise in areas like networking, the computational sciences and finance. The user must learn both the lexer generator and the parser generator, and then specify the lexer and the parser separately, in addition to the glue code to use them together. In addition, LEX and YACC do not support data-dependent parsing, do not generate internal representations automatically, and do not supply a collection of value-added tools.

Grammar-based Parser Generators. More modern parser generators alleviate several of the problems of LEX and YACC by providing more built-in programming support. For instance, the ANTLR parser generator [Parr and Quong 1995] allows the user to add annotations to a grammar to direct construction of a parse tree. However, all nodes in the abstract syntax tree have a single type, hence the guidance is rather crude when compared with the richly-typed structures that can be constructed using typed languages such as PADS/C, PADS/ML, DATASCRIPT or DDC$^\alpha$. The SABLE/CC compiler construction tool [Agnon 1998] goes beyond ANTLR by producing LALR(1) parsers along with richly-typed ASTs quite similar to those of PADS/C. Also like PADS/C or PADS/ML, descriptions do not contain actions. Instead, actions are only performed on the generated ASTs.

DEMETER [Lieberherr 1988] is another parser generator in the same general tradition as Lex, Yacc, ANTLR and SABLE/CC in that it is based on context-free grammars. However, DEMETER's class dictionaries are even more powerful than previous systems as they automatically generate "visitor" functions that traverse the internal representation of parsed data.

Despite their many benefits, all of the context-free grammar-based tools — LEX, YACC, ANTLR, SABLE/CC, and DEMETER — have some deficiencies when compared with tools built on the type theory described by DDC$^\alpha$. In particular, none of them include dependent or polymorphic data descriptions directly in their specification language (though some forms of dependency can likely be "hacked," at least in LEX and YACC, by programming arbitrary host language code in the semantic actions). Moreover, while the semantics of context-free grammars are obviously well understood, the semantics of the tools themselves, including the semantic actions that generate internal data structures, have not been as thoroughly studied. For instance, we know of no proof that ANTLR or SABLE/CC generated parsers are type safe. Finally, the error handling strategies for conventional parser generators are different than those of PADS/ML. They do not provide the programmer with programmatic access to errors, as PADS/ML does with parse descriptors. That said, such a laundry list of differences risks obscuring the more essential difference – that these tools are targeted at a different domain. The type-based tools such as PADS, DATASCRIPT, and PACKETTYPES generate tools specificly suited to processing ad hoc data (both binary and ASCII) whereas the others generate tools suited to the processing and analysis of programs.

Modern Programming Technologies. There are many parallels between DDC$^\alpha$ and *parser combinators* [Burge 1975; Hutton and Meijer 1998]. In particular, DDC$^\alpha$'s dependent sum construct is reminiscent of the bind operator in the monadic formulation of parser combinators. Indeed, we can model DDC$^\alpha$'s dependent sums in Haskell as follows.

```
sigma :: P s -> (s->P t) -> P (s,t)
sigma m q = do {x <- m; y <- q x; return (x,y)}
```

However, there are a number of deeper differences between parser combinators and DDC$^\alpha$ descriptions:

—As a language of types, DDC$^\alpha$, and related languages such as PADS/C and PADS/ML, exploit programmer intuitions concerning the meaning of types directly, and has a completely different "look and feel" from Haskell combinator libraries.

—DDC$^\alpha$, with its parse descriptors, has quite a different error reporting mechanism from parser combinator libraries.

—The multi-faceted, nonstandard semantics of dependent DDC$^\alpha$ types is structured entirely differently from the semantics of parser generators given in the literature.

—A parser combinator library is specifying a *parser*, while a term in DDC$^\alpha$ is *describing a data format*, which means that the DDC$^\alpha$ term can be used to generate a printer and other analysis tools in addition to a parser.

Another related technology is *type-directed* or *generic* programming [Jeuring and Jansson 1996; Hinze 2000; Lämmel and Peyton Jones 2003]. Type-directed programming techniques allow users to define algorithms based on induction over the structure of a type rather than induction (or recursion) over the structure of a value. Clearly, the parsers defined by DDC$^\alpha$ are defined by induction over the structure of types and hence may be thought of as type-directed programs. However, most of the general-purpose research on type-directed programming gives little or no insight into the specific problem of how one defines parsers from a language of dependent types. Likewise, the semantics of generic programming languages clearly does not directly serve as a semantics for PADS/C or PADS/ML.

The closest connection between DDC$^\alpha$ and research in type-directed programming can likely be found in the work of van Weelden *et al* [van Weelden et al. 2005]. These authors investigated the use of polytypic programming to produce a parser for a language based only on the specification of its AST type(s). In this way, the AST types themselves serve as the grammar for the language. They also investigate applying this approach to other compiler-related analyses, like scope checking and type inference. However, while their "types-as-grammar" approach is clearly related to PADS/ML, they use standard (non-dependent) types as parser specifications, and they study parsing techniques for programming languages, not ad hoc data. Dependent types are very important in the domain of ad hoc data, where it is very common for a tag early in data to determine later parsing behavior or an integer to determine the length of some future array.

XML-based tools. Rather than programming directly with data in its ad hoc format, it may be useful to first convert it to XML. Once in XML, any one of hundreds of XML-based tools may be used to manipulate the data. XSugar [Brabrand et al. 2005] is one tool that allows users to specify an alternative non-XML syntax for XML languages using a context-free grammar. This tool automatically generates conversion tools between XML and non-XML syntax. Another such tool is the Binary Format Description language (BFD) [Myers and Chappell 2000]. BFD is able to convert the raw binary or ASCII data into XML-tagged data where it can then be processed using XML-processing tools. While both these tools are useful for many tasks, conversion to XML is not always the answer. Such conversion often results in an 8-10 times blowup in data size over the native form. Moreover, when the programmer is not familiar with XML, there is a high barrier to entry — not only does the programmer have to learn the ad hoc format, but they must also learn XML and the XML conversion tool. Altogether, this is too heavyweight for many simple data processing tasks.

DFDL is a data format specification language with an XML-based syntax and type structure [DFDL 2005; Beckerle and Westhead 2004]. DFDL is a language *specification*, not an entire system or an implementation – PADS/ML could, perhaps, serve as the basis of a robust DFDL implementation. Like the PADS/ML language, DFDL has a rich collection of base types and supports a variety of ambient codings. In terms of expressiveness, we believe the DFDL consortium has added dependency and semantic constraints to match the expressiveness of PADS/C. However, because the specification is still under development, we cannot give a more detailed comparison at this point.

XDTM [Moreau et al. 2005; Zhao et al. 2005] uses XML Schema to describe the locations of a collection of sources spread across a local file system or distributed across a network of computers. However, XDTM has no means of specifying the contents of files, so XDTM and PADS/ML solve complementary problems. The METS schema [METS 2003] is similar to XDTM as it describes metadata for objects in a digital library, including a hierarchy such objects.

Databases. Commercial database products provide support for parsing data in external formats so the data can be imported into their database systems, but they typically support a limited number of formats. Also, no declarative description of the original format is exposed to the user for their own use, and they have fixed methods for coping with erroneous data. For these reasons, PADS/ML is complementary to database systems. We strongly believe that in the future, commercial database systems could and should support a PADS-like description language that allows users to import information from almost any format.

Parsing Theory. To the best of our knowledge, our work on DDC$^{\alpha}$ is 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. Of course, there are other formalisms for defining parsers, most famously, regular expressions and contex-free grammars. In terms of recognition power, these formalisms differ from our type theory in that they have nondeterministic choice, but do not have dependency or constraints. We have found that dependency and constraints are absolutely essential for describing most of the ad hoc data sources we have studied. Perhaps more importantly though, unlike standard theories of context-free grammars, we do not treat our type theory merely as a recognizer for a collection of strings. Our type-based descriptions define *both* external data formats *and* rich invariants on the internal parsed data structures. This dual interpretation of types lies at the heart of tools such as PADS, DATASCRIPT and PACKETTYPES.

Parsing Expression Grammars (PEGs), studied in the early 70s [Birman and Ullman 1973] and revitalized more recently by Ford [Ford 2004], evolved from context-free grammars but have deterministic, prioritized choice like DDC$^\alpha$ as opposed to nondeterministic choice. Though PEGs have syntactic lookahead operators, they may be parsed in linear time through the use of "packrat parsing" techniques [Ford 2002; Grimm 2004]. Once again, our multiple interpretations of types in DDC$^\alpha$ makes our theory substantially different from the theory of PEGs.

9. CONCLUSION

Ad hoc data is pervasive and valuable: in industry, in medicine, and in scientific research. Such data tends to have poor documentation, to contain various kinds of errors, and to be voluminous. Unlike well-behaved data in standardized relational or XML formats, such data has little or no tool support, forcing data analysts and scientists to waste valuable time writing brittle custom code, even if all they want to do is convert their data into a well-behaved format. To improve the situation, various researchers have developed data description languages such as PADS, DATASCRIPT, and PACKETTYPES. Such languages allow analysts to write terse, declarative descriptions of ad hoc data. A compiler then generates a parser and customized tools. Because these languages are tailored to their domain, they can provide useful services automatically while a more general purpose tool, such as LEX/YACC or PERL, cannot.

In the spirit of Landin, we have taken the first steps toward specifying a semantics for this class of languages by defining the data description calculus DDC$^\alpha$. This calculus, which is a dependent type theory with a simple set of orthogonal primitives, is expressive enough to describe the features of PADS, DATASCRIPT, and PACKETTYPES. In keeping with the spirit of the data description languages, our semantics is transformational: instead of simply recognizing a collection of input strings, we specify how to transform those strings into canonical in-memory representations annotated with error information. Furthermore, we prove that the error information is meaningful, allowing analysts to rely on the error summaries rather than having to re-vet the data by-hand.

We have already used the semantics to identify bugs in the implementation of PADS/C and to highlight areas where PADS/C sacrifices safety for speed. We have also used the semantics as a guide for the design of a whole new language, PADS/ML, designed specifically for functional programmers. In the future, we hope DDC$^\alpha$ will serve as a solid foundation for the next 700 data description languages.

Acknowledgments

Thanks to Andrew Appel for suggesting we refer to Landin's seminal paper on the next 700 programming languages. Discussions with John Launchbury concerning parser combinators were also most helpful. We appreciate the insights and thoughtful reviews of the program committee for the 33rd ACM Symposium on Principles of Programming Languages, who commented on an earlier version of this work.

REFERENCES

AGNON, E. 1998. SableCC: An object oriented compiler framework. M.S. thesis, School of Computer Science, McGill University, Montreal.

ASDL. Abstract syntax description language. `http://sourceforge.net/projects/asdl`.

BACK, G. 2002. DataScript - A specification and scripting language for binary data. In *Generative Programming and Component Engineering*. Vol. 2487. Lecture Notes in Computer Science, 66–77.

BECKERLE, M. AND WESTHEAD, M. 2004. GGF DFDL primer. `http://www.ggf.org/Meetings/GGF11/Documents/DFDL_Primer_v2.pdf`. Global Grid Forum.

BIRMAN, A. AND ULLMAN, J. D. 1973. Parsing algorithms with backtrack. *Information and Control 23,* 1 (Aug.).

BRABRAND, C., MØLLER, A., AND SCHWARTZBACH, M. I. 2005. Dual syntax for XML languages. In *Tenth International Symposium on Database Programming Languages*. Lecture Notes in Computer Science, vol. 3774. Springer-Verlag, 27–41.

BURGE, W. 1975. *Recursive Programming Techniques*. Addison Wesley.

CONSORTIUM, G. O. Gene ontology project. http://www.geneontology.org/.

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

DUBUISSON, O. 2001. *ASN.1: Communication between heterogeneous systems*. Morgan Kaufmann.

FERNÁNDEZ, M. F., SIMÉON, J., CHOI, B., MARIAN, A., AND SUR, G. 2003. Implementing XQuery 1.0: The Galax experience. In *VLDB*. ACM Press, 1077–1080.

FISHER, K. AND GRUBER, R. 2005. Pads: A domain specific language for processing ad hoc data. In *ACM Conference on Programming Language Design and Implementation*. ACM Press, 295–304.

FISHER, K., MANDELBAUM, Y., AND WALKER, D. 2006. The next 700 data description languages. In *ACM Symposium on Principles of Programming Languages*. ACM Press, 2–15.

FORD, B. 2002. Packrat parsing:: simple, powerful, lazy, linear time. In *ACM International Conference on Functional Programming*. ACM Press, 36–47.

FORD, B. 2004. Parsing expression grammars: a recognition-based syntactic foundation. In *ACM Symposium on Principles of Programming Languages*. ACM Press, 111–122.

GRIMM, R. 2004. Practical packrat parsing. Tech. Rep. TR2004-854, New York University. Mar.

GUSTAFSSON, P. AND SAGONAS, K. 2004. Adaptive pattern matching on binary data. In *European Symposium on Programming*. Springer, 124–139.

HARPER, R. 2005. *Programming Languages: Theory and Practice.* Unpublished. Available at `http://www-2.cs.cmu.edu/~rwh/`.

Hierarchical Data Format 5. Hierarchical data format 5. National Center for Supercomputing Applications (NCSA) at the University of Illinois at Urbana-Champaign (UIUC). `http://hdf.ncsa.uiuc.edu/HDF5/`.

HINZE, R. 2000. A new approach to generic functional programming. In *ACM Symposium on Principles of Programming Languages*. 119–132.

HUTTON, G. AND MEIJER, E. 1998. Monadic Parsing in Haskell. *Journal of Functional Programming 8,* 4 (July), 437–444.

IGARASHI, A., PIERCE, B., AND WADLER, P. 1999. Featherwieght Java: a minimal core calculus for Java and GJ. In *ACM Conference on Object-oriented Programming, Systems, Languages, and Applications*. ACM Press, 132–146.

JEURING, J. AND JANSSON, P. 1996. Polytypic programming. In *Second International School on Advanced Functional Programming*. Lecture Notes in Computer Science, vol. 1129. 68–114.

KRISHNAMURTHY, B. AND REXFORD, J. 2001. *Web Protocols and Practice*. Addison Wesley.

LÄMMEL, R. AND PEYTON JONES, S. 2003. Scrap your boilerplate: a practical design pattern for generic programming. ACM Press, 26–37.

LANDIN, P. J. 1966. The next 700 programming languages. *Communications of the ACM 9,* 3 (Mar.), 157 – 166.

LIEBERHERR, K. 1988. Object-oriented programming with class dictionaries. *Lisp and Symbolic Computation 1*, 185–212.

MANDELBAUM, Y. 2006. The theory and practice of data description. Ph.D. thesis, Princeton University.

MANDELBAUM, Y., FISHER, K., WALKER, D., FERNANDEZ, M., AND GLEYZER, A. 2007. PADS/ML: A functional data description language. In *ACM Symposium on Principles of Programming Languages*. ACM Press, ?–? To appear.

MCCANN, P. AND CHANDRA, S. 2000. PacketTypes: Abstract specificationa of network protocol messages. In *ACM Conference of Special Interest Group on Data Communications*. ACM Press, 321–333.

METS 2003. METS: An overview and tutorial. `http://www.loc.gov/standards/mets/METSOverview.v2.html`.

MOREAU, L., ZHAO, Y., FOSTER, I., VOECKLER, J., AND WILDE, M. 2005. XDTM: The XML data type and mapping for specifying datasets. In *European Grid Conference*.

MYERS, J. AND CHAPPELL, A. 2000. Binary format definition (BFD). `http://collaboratory.emsl.pnl.gov/sam/bfd/`.

Newick. The Newick tree format. PHYLIP (the PHYLogeny Inference Package) web site. `http://evolution.genetics.washington.edu/phylip/newicktree.html`.

Newick data. Tree formats. Workshop on Molecular Evolution web site. `http://workshop.molecularevolution.org/resources/fileformats/tree_formats.php`.

PARR, T. J. AND QUONG, R. W. 1995. ANTLR: A predicated- *ll(k)* parser generator. *Software – practice and experience 25,* 7 (July), 789–810.

PAXSON, V. 1999. A system for detecting network intruders in real-time. In *Computer Networks*.

PIERCE, B. C. 2002. *Types and Programming Languages*. The MIT Press.

VAN WEELDEN, A., SMETSERS, S., AND PLASMEIJER, R. 2005. Polytypic syntax tree operations. In *Implementation and Application of Functional Languages, 17th International Workshop, IFL 2005, Revised Selected Papers*. Lecture Notes in Computer Science, vol. 4015. Springer, Dublin, Ireland.

WIKSTRÖM, C. AND ROGVALL, T. 1999. Protocol programming in Erlang using binaries. In *Fifth International Erlang/OTP User Conference*.

ZHAO, Y., DOBSON, J., FOSTER, I., MOREAU, L., AND WILDE, M. 2005. A notation and system for expressing and executing cleanly typed workflows on messy scientific data. *ACM SIGMOD Record 34,* 3, 37–43.