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. Though the syntax of everyday programming languages might be considered "ad hoc," we

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.

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.

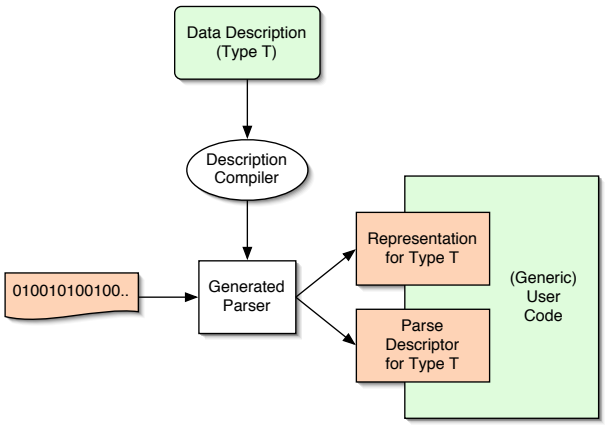

Fig. 2. Architecture of PADS system.

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. James and Malpani developed the *Data Definition Language* (DDL) [James and Malpani 2003], a scripting language similar to DATASCRIPT, but targeted at the .NET runtime. 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 all 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 type-based ad hoc data processing languages, 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 type-based 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.

DDC$^\alpha$ has a denotational semantics that interprets its data descriptions in two different ways. The first dimension of the semantics describes the types of the data structures that result from using a particular data description as a parser. Client programmers use this semantics to write safe, well-typed application programs against the libraries generated by DDC$^\alpha$ descriptions. The second dimension of the semantics explains how data descriptions are interpreted as parsing functions that map external representations (bits) into data structures in a typed lambda calculus. These parsers produce both 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 wherever 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 and helped us explore important extensions. In particular, driven by examples found in biological data [Consortium ; Newick data 2003], we decided to add recursion to PADS/C.

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 work out the details in a clean, elegant setting and prove our implementation technique 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, this is the first 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 data 2003]. 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 under-

standing 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.

One explicit non-contribution of this work is the development of new techniques for recognizing any particularly interesting class of grammars. The parsers generated by DDC$^\alpha$ or PADS are completely ordinary recursive-descent parsers. The novelty of our contribution comes entirely from establishing the connection between these traditional parsers and modern type theory.

Many 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, this article makes two important additional contributions. First, we have improved the structure of the semantics of the DDC$^\alpha$ in several ways. In particular, we eliminated the "contractiveness" constraint, which allowed us to substantially simplify and standardize the earlier kinding rules. Second, we added polymorphism to the calculus to elucidate 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. Note that we call the new version of our calculus DDC$^\alpha$, to distinguish it from the previous version, DDC. 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 and related data description languages such as PADS/ML, PACKETTYPES, and DATASCRIPT 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 form and function of IPADS by giving its syntax and explaining several examples. Later sections show how to give a formal semantics to IPADS.

Preliminary Concepts. As in PADS/C, PADS/ML, PACKETTYPES, and DATASCRIPT, the data descriptions in IPADS 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 C 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 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 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 closing parenthesis of the internal tree node.

The Newick Standard format and other formats that describe tree-shaped hierarchies [Consortium ; Newick data 2003] 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

```
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

```
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.

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.

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

```
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

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

Fig. 6. IPADS Syntax

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 the polymorphic lambda calculus F_ω[Girard 1972; Reynolds 1974] – for expressing both the expressions embedded in types and the interpretations of DDC$^\alpha$. We discuss this host language 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, which we define in Section 3.2. We use kinds κ to classify types. In particular, kind T classifies types that directly describe data. Types with this kind include integers, pairs of IP addresses and strings of length ten, among others. Kind $\sigma \rightarrow \kappa$ describes functions from values with type σ to types with arbitrary kind κ. An example of a type with such a kind is base type constructor `Pstring_FW` that takes as an argument an integer expression e and returns a type for strings with length e. Finally, kind $\mathsf{T} \rightarrow \kappa$ classifies functions from first-order types (those having kind T) to arbitrary types (those having kind κ). A useful type with such a kind is a function that takes any first-order type τ as an argument and returns the type that describes data corresponding to either τ or the character $'-'$, indicating a missing value. This kinding system disallows types that take type functions as arguments. Such types add complexity to the system, and we have not encountered a compelling need for them in practice.

The most basic types of kind T are `unit` and `bottom`. Type `unit` describes the empty string; it succeeds on all input. While vacuous by itself, the type `unit` is useful when combined with other type constructors. For example, a type that unions an integer type with `unit` corresponds to an optional integer. In contrast, the type `bottom` describes no strings; it fails on all input. When used within a compound type containing choices, `bottom` indicates that the choice leading to `bottom` fails and an alternative branch must be chosen. We will see another use of `bottom` when we discuss array types.

The syntax $C(e)$ denotes a base type C parameterized by expression e. Such a base type recognizes and transforms atomic values within the data source; typical examples include strings, various kinds of integers, dates, times, *etc.* The parameter expression plays a type-dependent role, specifying, for example, digit lengths for integers or terminating conditions for strings. If the parameter is not needed for a particular base type, we often omit it from the syntax for clarity. Concretely, we use the base type `Pstring`(`s`) to denote strings terminated by the string `s` and `Puint` to denote sequences of digits of arbitrary length. We adopt the convention that base types start with a capital `P`.

We provide abstraction $\lambda x.\tau$ and application $\tau\ e$ so that we may parameterize types by expressions. For example, if we had a fragment of a data format that was terminated in some circumstances by a vertical bar and in others by a semi-colon, we can use abstraction to parameterize the description of the format by the terminating character, yielding a description of the form $\lambda c.\tau_d$, where τ_d is a description of the fragment in terms of terminating character c. We can then apply the function to either a vertical bar or a semi-colon as circumstances require; *i.e.*, the application $(\lambda c.\tau_d)\ '|'$ specializes the description to the vertical bar case.

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. A common idiom for which such sums are useful is an integer field followed by a string of the corresponding length, for example: $\Sigma\,$`x`:`Pint`.`Pstring_FW`(`x`).

Sum types $\tau_1 + \tau_2$ express flexibility in the data format, as they describe data matching either τ_1 or τ_2. For example, the type `Puint32` + `Pstring_FW`(`3`) describes a format that is either an unsigned, 32-bit integer or a string of length three. 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 use

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

Fig. 7. DDC$^{\alpha}$ syntax

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. Examples that use this construct arise in various Cobol data formats and in processing internet packets, which can be viewed at multiple levels of abstraction, *e.g.*, as a header and an uninterpreted collection of bytes or as a header followed by a packet in the format of the next level of the communication stack.

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. For example, the type $\{\texttt{x:Pint} \mid \texttt{x} > 100\}$ checks that the integer `x` is greater than 100. As another example, the constrained type $\{\texttt{x:Pstring\_FW(1)} \mid \texttt{x} = '{,}'\}$ describes exactly the comma character. Because such "singleton types" arise frequently, we introduce a short-hand notation for them. In particular, we use $\texttt{S}(\text{“,”})$ as an abbreviation for the type above. Similarly, $\texttt{S}(\text{“;”})$ abbreviates the corresponding description of precisely the semi-colon character.

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 $\texttt{len}_{10}$ defined to be $\lambda\texttt{s.len s} = 10$ that checks if the sequence `s` has 10 elements would terminate a sequence when it reaches length 10. The type τ_t is used when data following the array will indicate the array's completion. Commonly, constrained singleton types are used to specify that a particular value terminates the sequence. For example, when used as a terminator, the type $\texttt{S}(\text{“;”})$ specifies that a semicolon ends the array. However, if no particular value or set of values terminates the array, then we can use `bottom` to ensure that the array is not terminated based on the terminating type τ_t. As an example, the type $\texttt{Pint32 seq}(\texttt{S}(\text{“,”}), \texttt{len}_{10}, \texttt{S}(\text{“;”}))$ describes a sequence of ten 32-bit integers separated by commas and terminated by a semi-colon.

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. As an example, the type abstraction $\lambda\alpha.\texttt{S}(\text{“}-\text{”}) + \alpha$ takes any (first-order) type as an argument and constructs a type that describes either the singleton string "$-$", denoting a missing value, or the argument type.

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

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

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

it is likely to depend 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.

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 pa-

[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]

per: anonymous functions $\lambda x.e$ for $\texttt{fun}\ f\ x = e$, with $f \notin \mathrm{FV}(e)$; function bindings $\texttt{letfun}\ f\ x = e\ \texttt{in}\ e'$ for $\texttt{let}\ f = \texttt{fun}\ f\ x = e\ \texttt{in}\ e'$; span for $\texttt{offset} * \texttt{offset}$. We often use pattern-matching syntax for pairs in place of explicit projections, as in $\lambda(B, \omega).e$ and $\texttt{let}\ (\omega, r, p) = e\ \texttt{in}\ e'$. 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 $x.\texttt{rep}$ and $x.\texttt{pd}$ 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 $\texttt{name} = \tau$ before giving the type that describes the data source.

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

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

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

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

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

In the example, we 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 $\lambda\texttt{x.false}$ 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.

$\Delta; \Gamma \vdash \tau : \kappa$	*type kinding*
$\tau \longrightarrow \tau'$	*type normalization*
$[\![\tau]\!]_{\text{rep}} = \sigma$	*representation-type interpretation of* DDC^α
$[\![\tau]\!]_{\text{PD}} = \sigma$	*parse-descriptor type interpretation of* DDC^α
$[\![\tau]\!]_{\text{PDb}} = \sigma$	*pd-body type interpretation of* DDC^α
$[\![\tau]\!]_{\text{P}} = e$	*parsing semantics of* DDC^α
$[\![\tau{:}\kappa]\!]_{\text{PT}} = \sigma$	*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^α *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^α functions and judgments defined in this section.

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

Table II. F_ω judgments referenced in this section.

We give semantics to DDC^α 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^α, detailing the types of the data's in-memory representation and parse descriptor. The function $[\![\cdot]\!]_{\text{P}}$ describes the *parsing semantics* of DDC^α, 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^α types. We then specify the normalization relation after which we formalize the three-fold semantics of DDC^α 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.

4.1 DDC^α Kinding

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

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

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

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

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

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

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

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

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

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

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

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

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

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.

$$[\![\,\cdot\,]\!]_{\text{rep}} = \cdot \qquad [\![\,\cdot\,]\!]_{\text{PD}} = \cdot$$
$$[\![\Delta, \alpha{:}\mathsf{T}]\!]_{\text{rep}} = [\![\Delta]\!]_{\text{rep}}, \alpha_{\mathtt{rep}}{::}\mathsf{T} \qquad [\![\Delta, \alpha{:}\mathsf{T}]\!]_{\text{PD}} = [\![\Delta]\!]_{\text{PD}}, \alpha_{\mathtt{PDb}}{::}\mathsf{T}$$

Translation $[\![\Delta]\!]_{F_\omega}$ simply combines the two ($[\![\Delta]\!]_{F_\omega} = [\![\Delta]\!]_{\text{rep}}, [\![\Delta]\!]_{\text{PD}}$). These translations

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

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

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

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

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

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 CONST, 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 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 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 $[[\![\tau]\!]_{\texttt{rep}}/\alpha_{\texttt{rep}}][[\![\tau]\!]_{\texttt{PDb}}/\alpha_{\texttt{PDb}}]$ is applied to the subcomponent expression or type. For

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

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

Fig. 12. Representation-type interpretation function.

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.

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

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

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

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

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

Fig. 13. Parse-descriptor type interpretation function

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.

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

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

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

Fig. 14. F_ω types for parsing functions.

subelements because the data they describe is not parsed from the data source. The `absorb` PD type is `unit` just like its representation. We assume that the user wants the parser to discard the parse descriptor just as it discards the representation. 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, $\lambda\alpha.\tau$.

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

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

$[\![\mathtt{unit}]\!]_{\mathrm{P}} = \lambda(\mathtt{B},\omega).(\omega, \mathtt{R}_{\mathtt{unit}}(), \mathtt{P}_{\mathtt{unit}}(\omega))$

$[\![\mathtt{bottom}]\!]_{\mathrm{P}} = \lambda(\mathtt{B},\omega).(\omega, \mathtt{R}_{\mathtt{bot}}(), \mathtt{P}_{\mathtt{bot}}(\omega))$

$[\![C(e)]\!]_{\mathrm{P}} = \lambda(\mathtt{B},\omega).\mathcal{B}_{\mathrm{imp}}(C)\ (e)\ (\mathtt{B},\omega)$

$[\![\lambda x.\tau]\!]_{\mathrm{P}} = \lambda x.[\![\tau]\!]_{\mathrm{P}}$

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

$[\![\Sigma\, x{:}\tau.\tau']\!]_{\mathrm{P}} =$
$\lambda(\mathtt{B},\omega).$
  $\mathtt{let}\ (\omega', \mathtt{r}, \mathtt{p}) = [\![\tau]\!]_{\mathrm{P}}\ (\mathtt{B},\omega)\ \mathtt{in}$
  $\mathtt{let}\ \mathtt{x} = (\mathtt{r},\mathtt{p})\ \mathtt{in}$
  $\mathtt{let}\ (\omega'', \mathtt{r}', \mathtt{p}') = [\![\tau']\!]_{\mathrm{P}}\ (\mathtt{B},\omega')\ \mathtt{in}$
  $(\omega'', \mathtt{R}_{\Sigma}(\mathtt{r},\mathtt{r}'), \mathtt{P}_{\Sigma}(\mathtt{p},\mathtt{p}'))$

$[\![\tau + \tau']\!]_{\mathrm{P}} =$
$\lambda(\mathtt{B},\omega).$
  $\mathtt{let}\ (\omega', \mathtt{r}, \mathtt{p}) = [\![\tau]\!]_{\mathrm{P}}\ (\mathtt{B},\omega)\ \mathtt{in}$
  $\mathtt{if}\ \mathtt{isOk}(\mathtt{p})\ \mathtt{then}$
    $(\omega', \mathtt{R}_{+\mathtt{left}}(\mathtt{r}), \mathtt{P}_{+\mathtt{left}}(\mathtt{p}))$
  $\mathtt{else\ let}\ (\omega', \mathtt{r}, \mathtt{p}) = [\![\tau']\!]_{\mathrm{P}}\ (\mathtt{B},\omega)\ \mathtt{in}$
  $(\omega', \mathtt{R}_{+\mathtt{right}}(\mathtt{r}), \mathtt{P}_{+\mathtt{right}}(\mathtt{p}))$

$[\![\tau\ \&\ \tau']\!]_{\mathrm{P}} =$
$\lambda(\mathtt{B},\omega).$
  $\mathtt{let}\ (\omega', \mathtt{r}, \mathtt{p}) = [\![\tau]\!]_{\mathrm{P}}\ (\mathtt{B},\omega)\ \mathtt{in}$
  $\mathtt{let}\ (\omega'', \mathtt{r}', \mathtt{p}') = [\![\tau']\!]_{\mathrm{P}}\ (\mathtt{B},\omega)\ \mathtt{in}$
  $(\mathtt{max}(\omega', \omega''), \mathtt{R}_{\&}(\mathtt{r},\mathtt{r}'), \mathtt{P}_{\&}(\mathtt{p},\mathtt{p}'))$

$[\![\{x{:}\tau \mid e\}]\!]_{\mathrm{P}} =$
$\lambda(\mathtt{B},\omega).$
  $\mathtt{let}\ (\omega', \mathtt{r}, \mathtt{p}) = [\![\tau]\!]_{\mathrm{P}}\ (\mathtt{B},\omega)\ \mathtt{in}$
  $\mathtt{let}\ \mathtt{x} = (\mathtt{r},\mathtt{p})\ \mathtt{in}$
  $\mathtt{let}\ \mathtt{c} = e\ \mathtt{in}$
  $(\omega', \mathtt{R}_{\mathtt{con}}(\mathtt{c},\mathtt{r}), \mathtt{P}_{\mathtt{con}}(\mathtt{c},\mathtt{p}))$

$[\![\tau\ \mathtt{seq}(\tau_s, e, \tau_t)]\!]_{\mathrm{P}} =$
$\lambda(\mathtt{B},\omega).$
  $\mathtt{letfun\ isDone}\ (\omega,\mathtt{r},\mathtt{p}) =$
    $\mathtt{EoF}(\mathtt{B},\omega)\ \mathtt{or}\ e\ (\mathtt{r},\mathtt{p})\ \mathtt{or}$
    $\mathtt{let}\ (\omega',\mathtt{r}',\mathtt{p}') = [\![\tau_t]\!]_{\mathrm{P}}(\mathtt{B},\omega)\ \mathtt{in}$
    $\mathtt{isOk}(\mathtt{p}')$
  $\mathtt{in}$
  $\mathtt{letfun\ continue}\ (\omega,\omega',\mathtt{r},\mathtt{p}) =$
    $\mathtt{if}\ \omega = \omega'\ \mathtt{or}\ \mathtt{isDone}\ (\omega',\mathtt{r},\mathtt{p})\ \mathtt{then}\ (\omega',\mathtt{r},\mathtt{p})$
    $\mathtt{else\ let}\ (\omega_{\mathtt{s}},\mathtt{r}_{\mathtt{s}},\mathtt{p}_{\mathtt{s}}) = [\![\tau_s]\!]_{\mathrm{P}}\ (\mathtt{B},\omega')\ \mathtt{in}$
    $\mathtt{let}\ (\omega_{\mathtt{e}},\mathtt{r}_{\mathtt{e}},\mathtt{p}_{\mathtt{e}}) = [\![\tau]\!]_{\mathrm{P}}\ (\mathtt{B},\omega_{\mathtt{s}})\ \mathtt{in}$
    $\mathtt{continue}\ (\omega',\omega_{\mathtt{e}},\mathtt{R}_{\mathtt{seq}}(\mathtt{r},\mathtt{r}_{\mathtt{e}}),\mathtt{P}_{\mathtt{seq}}(\mathtt{p},\mathtt{p}_{\mathtt{s}},\mathtt{p}_{\mathtt{e}}))$
  $\mathtt{in}$
  $\mathtt{let}\ \mathtt{r} = \mathtt{R}_{\mathtt{seq\_init}}()\ \mathtt{in}$
  $\mathtt{let}\ \mathtt{p} = \mathtt{P}_{\mathtt{seq\_init}}(\omega)\ \mathtt{in}$
  $\mathtt{if\ isDone}\ (\omega,\mathtt{r},\mathtt{p})\ \mathtt{then}\ (\omega,\mathtt{r},\mathtt{p})$
  $\mathtt{else\ let}\ (\omega_{\mathtt{e}},\mathtt{r}_{\mathtt{e}},\mathtt{p}_{\mathtt{e}}) = [\![\tau]\!]_{\mathrm{P}}\ (\mathtt{B},\omega)\ \mathtt{in}$
  $\mathtt{continue}\ (\omega,\omega_{\mathtt{e}},\mathtt{R}_{\mathtt{seq}}(\mathtt{r},\mathtt{r}_{\mathtt{e}}),\mathtt{P}_{\mathtt{seq}}(\mathtt{p},\mathtt{P}_{\mathtt{unit}}(\omega),\mathtt{p}_{\mathtt{e}}))$

$[\![\alpha]\!]_{\mathrm{P}} = \mathtt{parse}_{\alpha}$

$[\![\mu\alpha.\tau]\!]_{\mathrm{P}} =$
$\mathtt{fun}\ \mathtt{parse}_{\alpha}\ (\mathtt{B}{:}\mathtt{bits}, \omega{:}\mathtt{offset}) :$
    $\mathtt{offset} * [\![\mu\alpha.\tau]\!]_{\mathrm{rep}} * [\![\mu\alpha.\tau]\!]_{\mathrm{PD}} =$
  $\mathtt{let}\ (\omega', \mathtt{r}, \mathtt{p}) =$
    $[\![\tau]\!]_{\mathrm{P}}[[\![\mu\alpha.\tau]\!]_{\mathrm{rep}}/\alpha_{\mathtt{rep}}][[\![\mu\alpha.\tau]\!]_{\mathrm{PDb}}/\alpha_{\mathtt{PDb}}]\ (\mathtt{B},\omega)$
  $\mathtt{in}$
    $(\omega', \mathtt{fold}[[\![\mu\alpha.\tau]\!]_{\mathrm{rep}}]\,\mathtt{r}, (\mathtt{p.h}, \mathtt{fold}[[\![\mu\alpha.\tau]\!]_{\mathrm{PDb}}]\,\mathtt{p}))$

$[\![\lambda\alpha.\tau]\!]_{\mathrm{P}} = \Lambda\alpha_{\mathtt{rep}}.\Lambda\alpha_{\mathtt{PDb}}.\lambda\mathtt{parse}_{\alpha}.[\![\tau]\!]_{\mathrm{P}}$

$[\![\tau_1\tau_2]\!]_{\mathrm{P}} = [\![\tau_1]\!]_{\mathrm{P}}\ [[\![\tau_2]\!]_{\mathrm{rep}}]\ [[\![\tau_2]\!]_{\mathrm{PDb}}]\ [\![\tau_2]\!]_{\mathrm{P}}$

$[\![\mathtt{compute}(e{:}\sigma)]\!]_{\mathrm{P}} =$
  $\lambda(\mathtt{B},\omega).(\omega, \mathtt{R}_{\mathtt{compute}}(e), \mathtt{P}_{\mathtt{compute}}(\omega))$

$[\![\mathtt{absorb}(\tau)]\!]_{\mathrm{P}} =$
$\lambda(\mathtt{B},\omega).$
  $\mathtt{let}\ (\omega', \mathtt{r}, \mathtt{p}) = [\![\tau]\!]_{\mathrm{P}}\ (\mathtt{B},\omega)\ \mathtt{in}$
  $(\omega', \mathtt{R}_{\mathtt{absorb}}(\mathtt{p}), \mathtt{P}_{\mathtt{absorb}}(\mathtt{p}))$

$[\![\mathtt{scan}(\tau)]\!]_{\mathrm{P}} =$
$\lambda(\mathtt{B},\omega).$
  $\mathtt{letfun\ try\ i} =$
    $\mathtt{let}\ (\omega', \mathtt{r}, \mathtt{p}) = [\![\tau]\!]_{\mathrm{P}}\ (\mathtt{B}, \omega + \mathtt{i})\ \mathtt{in}$
    $\mathtt{if\ isOk}(\mathtt{p})\ \mathtt{then}$
      $(\omega', \mathtt{R}_{\mathtt{scan}}(\mathtt{r}), \mathtt{P}_{\mathtt{scan}}(\mathtt{i}, \mathtt{sub}(\mathtt{B},\omega,\mathtt{i}+1), \mathtt{p}))$
    $\mathtt{else\ if\ EoF}(\mathtt{B}, \omega + \mathtt{i})\ \mathtt{then}$
      $(\omega, \mathtt{R}_{\mathtt{scan\_err}}(), \mathtt{P}_{\mathtt{scan\_err}}(\omega))$
    $\mathtt{else\ try}\ (\mathtt{i}+1)$
  $\mathtt{in\ try\ 0}$

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 (ec1, ec2) =
 if ec1 = fail or ec2 = fail then fail
 else if ec1 = err or ec2 = 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 (r1, r2) = (r1, r2)
fun H_Σ (h1, h2) =
 let nerr = pos(h1.nerr) + pos(h2.nerr) in
 let ec = if h2.ec = fail then fail
    else max_ec h1.ec h2.ec in
 let sp = (h1.sp.begin, h2.sp.end) in
    (nerr, ec, sp)
fun P_Σ (p1, p2) = (H_Σ(p1.h, p2.h), (p1, p2))

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_& (h1, h2) =
 let nerr = pos(h1.nerr) + pos(h2.nerr) in
 let ec = if h1.ec = fail and h2.ec = fail then fail
    else max_ec h1.ec h2.ec in
 let sp = (h1.sp.begin, max(h1.sp.end, h2.sp.end)) in
    (nerr, ec, sp)
fun P_& (p1, p2) = (H_& (p1.h, p2.h), (p1, p2))
```

Fig. 17. Constructor functions, part 1. Parse descriptor headers are sometimes referenced using dot notation as `h`. Their type 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.

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 occurring 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 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^{α} 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 $\mathtt{scan}(\tau)$ 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 on 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 occurred 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 general meta-theoretic properties should hold of the language. What are the "correct" invariants of data description languages? While the languages community has a good understanding of the desirable invariants for conventional programming languages, the corresponding properties of data description languages have not been studied.

We present the following two properties as critical invariants of our theory. Just like 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 these critical properties and sketch the proof that they hold for our DDC$^\alpha$ theory. A full proof can be found in Appendix A.

Before proceeding to the main elements of our meta-theory, we state a few simple requirements of DDC$^\alpha$ base types. Note that that the interface $\mathcal{B}_{\text{opty}}$ specifies the types of base-type parsers.

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) $\vdash \mathcal{B}_{imp}(C) : \mathcal{B}_{opty}(C)$.

Note that by condition 3, base type parsers must be closed.

The first interesting lemma we prove about the DDC$^\alpha$ is that evaluation commutes with semantic interpretation. This property allows reasoning 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. 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 also hold if the second premise were switched with the first conclusion.

Lemma 2 (Commutativity of Evaluation and Semantic Interpretation)
If $\vdash \tau : \kappa$ *and* $[\![\tau]\!]_P \rightarrow^* v$ *then there exists normal type* ν *such that*

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

5.1 Type Correctness

Our first key theorem 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.

Theorem 3 (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 parser generators, 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 that satisfy a number of invariants. Most importantly, when the parse descriptor reports that there are no errors in a particular substructure, the programmer is guaranteed that the corresponding representation satisfies all of the syntactic and semantic constraints expressed by the dependent DDC$^\alpha$ type description. When the pair of a parse descriptor and a representation satisfy these invariants, we say the pair is *canonical* or in *canonical form*.

The canonical form for each DDC$^\alpha$ type is 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. This relation is defined for all closed normal types ν with base kind T. The definition excludes types with higher kind, such as abstractions, because such types cannot directly produce representations and PDs. The second definition, $\mathrm{Canon}^*{}_\tau(r, p)$, defines the canonical form at an arbitrary type τ by first normalizing τ to eliminate the outermost type and value applications and then applying the relation $\mathrm{Canon}_\nu(r, p)$ at the resulting normal type ν.

For brevity in the definitions, we write $p.h.nerr$ as $p.nerr$ and use `pos` to denote the function that returns zero when passed zero and one when passed another natural number.

Definition 4 (Canonical Forms I)
$\mathrm{Canon}_\nu(r, p)$ *holds if and only if exactly one of the following is true:*

—$\nu =$ `unit` *and* $r = ()$ *and* $p.nerr = 0$.
—$\nu =$ `bottom` *and* $r =$ `none` *and* $p.nerr = 1$.
—$\nu = C(e)$ *and* $r =$ `inl` c *and* $p.nerr = 0$.

—$\nu = C(e)$ *and* $r = \texttt{inr none}$ *and* $p.nerr = 1$.

—$\nu = \Sigma\, x{:}\tau_1.\tau_2$ *and* $r = (r_1, r_2)$ *and* $p = (h, (p_1, p_2))$ *and* $h.nerr = \texttt{pos}(p_1.nerr) + \texttt{pos}(p_2.nerr)$, $\text{Canon}^*{}_{\tau_1}(r_1, p_1)$ *and* $\text{Canon}^*{}_{\tau_2[(r,p)/x]}(r_2, p_2)$.

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

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

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

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

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

—$\nu = \tau_e\,\texttt{seq}(\tau_s, e, \tau_t,)$, $r = (len, [\vec{r_i}])$, $p = (h, (neerr, len', [\vec{p_i}]))$, $neerr = \sum_{i=1}^{len} \texttt{pos}(p_i.nerr)$, $len = len'$, $\text{Canon}^*{}_{\tau_e}(r_i, p_i)$ *(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* $\text{Canon}^*{}_{\tau'[\mu\alpha.\tau'/\alpha]}(r', p')$.

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

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

—$\nu = \texttt{absorb}(\tau')$, $r = \texttt{inr none}$, *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* $\text{Canon}^*{}_{\tau'}(r', p')$.

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

Definition 5 (Canonical Forms II)
$\text{Canon}^*{}_{\tau}(r, p)$ *holds if and only if* $\tau \rightarrow^* \nu$ *and* $\text{Canon}_{\nu}(r, p)$.

Theorem 7 establishes that our generated parsers yield Canonical Forms under the assumption that all base type parsers produce values in canonical form, a condition stated formally in Condition 6.

Condition 6 (Base Type Parsers Produce Values in Canonical Form)
If $\vdash v : \sigma$, $\mathcal{B}_{kind}(C) = \sigma \rightarrow \mathsf{T}$ *and* $\mathcal{B}_{imp}(C)\ v\ (B, \omega) \rightarrow^* (\omega', r, p)$ *then* $\text{Canon}_{C(v)}(r, p)$.

Theorem 7 (Parsing to Canonical Forms)
If $\vdash \tau : \mathsf{T}$ *and* $[\![\tau]\!]_P\ (B, \omega) \rightarrow^* (\omega', r, p)$ *then* $\text{Canon}^*{}_{\tau}(r, p)$.

Theorem 7 has the following useful corollary, which ensures that a single check of the top-level parse descriptor is sufficient to verify the validity of an entire data representation in canonical form.

Corollary 8
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.

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

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

$\boxed{t \Downarrow \tau}$

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

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

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

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

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

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

Fig. 19. Encoding IPADS in DDC$^\alpha$

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 19.

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

This section defines 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 $\{\overrightarrow{e \Rightarrow x{:}t}\ t_{\text{def}}\}$.

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:

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

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 intersection 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.

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

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 elaboration into DDC$^{\alpha}$. Notice that IPADS type abstractions can have multiple parameters.

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

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

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 ($x_{v1}, x_{v2}, \ldots$). 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.

$$\text{Programs } p ::= \ldots \mid \texttt{Pdatatype}\ \alpha\ (\overrightarrow{\alpha})(x:\sigma) = \{\overrightarrow{x_v{:}t}\};\ p$$

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

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, or *nested*, datatypes [Bird and Meertens 1998]. 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 specified can be encoded in a straightforward manner with DDC$^\alpha$ sequences:

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

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 (`len`) 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:

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

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 the DDC$^\alpha$ does not support in a straightforward manner. One such feature is a label construct that permits the user to specify the form of data at computed offsets. A second such feature is a **forall**

construct that allows users to express constraints between different elements of an array. While the DDC$^\alpha$ does not currently support these important features directly, we believe it provides a solid semantic framework in which such variations might be analyzed, explored and modelled in the future. For instance, one could investigate adding the **forall** constraint found in both DATASCRIPT and PADS/C to the DDC$^\alpha$ host language (or perhaps coding forall directly as a fold) and using it in conjunction with DDC$^\alpha$ set types to try to express the kind of array constraints Back shows are useful in a number of binary formats [Back 2002]. However, we leave such investigations to future work: Like the basic λ-calculus or π-calculus, DDC$^\alpha$ is intended to capture the most common language features, while remaining simple enough that it can be extended with new features relatively easily.

7. APPLICATIONS OF THE SEMANTICS

The development of the DDC$^\alpha$ and the 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 ways, which we now discuss.

7.1 Bug Hunting

We developed the DDC$^\alpha$, in part, through a line-by-line analysis of key portions of the PADS/C implementation. In the process of trying to understand and formalize the implicit invariants in this code, 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.

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. In particular, given the type `nothing` that consumes no input, the type `nothing array(nothing,eof)` 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. We have since fixed the bug and verified the revised implementation using the semantics.

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, particularly as related to the structure of parse descriptors for recursive types. When we started, it was not obvious whether recursive-type parse descriptors should have their own headers, or whether they could use the header available through a single unfolding of the type. Ultimately, we chose the latter, but this required that we carefully design our system so as to ensure that said header would be available.

In our first version of DDC$^\alpha$, DDC, we included a so-called *contractiveness* condition to ensure the desired parse descriptor structure [Fisher et al. 2006]. However, when we added polymorphic types to DDC, we found that the contractiveness condition was unsuited to polymorphism, specialized, as it was, to recursive types[2]. We therefore revis-

[2]The condition was overly restrictive; extended naively to type functions it would have disallowed many useful

ited the treatment of type variables and devised a uniform method of ensuring appropriate parse-description structure, that was appropriate for both recursive and polymorphic types. Specifically, we limited type abstraction for parse descriptors to abstraction over the *body* of the descriptors, and included the header explicitly in the PD-tranlsation of type variables. This subtle interaction between type abstraction and parse descriptor structure would have been very difficult both to notice and to reason in about in the context of a full implementation. The abstraction provided by the semantics was critical in enabling us to effectively redesign this element of the system and to subsequently be confident in its correctness.

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 to disentangle the invariants related to polymorphism, as discussed. After we had defined the calculus, we were able to implement type abstractions as OCAML functors in approximately a week. Additionally, the implementation of PADS/ML's `plist` type is an almost literal translation of the semantics of `pseq` into OCAML. 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 qualifiers 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 develop a semantic theory for type-based data description languages. To the best of our knowledge, there is no other comparable semantic

functions, including, for example, the identity function.

theory for this family of languages. Existing theories of regular expressions, context-free grammars, parsing expression grammars [Birman and Ullman 1973; Ford 2004; 2002; Grimm 2004] or even context-sensitive grammars specify what strings can be recognized by a grammar, but such a specification only captures half of the semantics of languages like PADS or PACKETTYPES. In contrast, there exist formalisms for specifying programming languages as algebras, in which a single language specification captures both the concrete and abstract syntax of the language being specified. However, these systems target programming languages (and the like), not data formats. Our new theory gives a complete explanation of data description languages both in terms of the strings that are recognized and the properties of internal data structures that are generated, and in a manner appropriate to data formats.

In the following paragraphs, we compare and contrast our semantics and the design of data description languages like PADS to more traditional grammar-based parser generators, algebraic specification formalisms, and other related technologies such as parser combinator libraries, type-directed programming techniques, and XML-based tools.

Grammar-based Parser Generators. 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. Moreover, 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. Consequently, in our experience, programmers simply do not use tools such as LEX and YACC for managing ad hoc data.

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 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 the PADS languages. Traditional parsers do not provide the programmer with programmatic access to errors, as PADS/ML or PADS/C do through the use of their parse descriptors. That said, such a laundry list of technical differences risks obscuring the essential points – that these tools are based on a completely different semantic foundation and have a far different overall "look and feel."

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 many of the ad hoc data sources we have studied, particularly binary formats in which length fields are used pervasively. 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) form the basis for yet another class of parsers. This formalism was studied in the early 70s [Birman and Ullman 1973] and was revitalized more recently by Ford [Ford 2004]. Like the DDC$^\alpha$, PEGs are notable for having greedy, prioritized choice as opposed to the nondeterministic choice found in regular expressions or context-free grammars. Greedy, prioritized choice resolves ambiguities that would otherwise arise essentially by defining them away. PEGs also have syntactic lookahead operators and may be parsed in linear time through the use of "packrat parsing" techniques [Ford 2002; Grimm 2004]. Once again, however, the multiple interpretations of types in DDC$^\alpha$ makes our theory substantially different from the theory of PEGs.

Algebraic Specification Formalisms. Early results demonstrating a correspondence between algebras and languages (for example, Rus [Rus 1972]) led to the development of a number of systems for specifying languages based on algebraic principles. From a semantic perspective, these *algebraic specification formalisms* are closer to DDC$^\alpha$ than parser generator languages. We briefly discuss one such system: the Syntax Defintion Formalism SDF2 [Visser 1997] and its companion system ASF, the Algebraic Specification Formalism [Bergstra et al. 1989]. For a more detailed discussion of earlier systems, we refer the reader to Visser's Thesis [Visser 1997]. SDF2 differs from most parser generator systems in both its scope and its feature set. SDF2+ASF provides extensive support for specifying algebraic systems and associated properties, including the syntax and semantics. As with DDC$^\alpha$, elements defined in SDF2 have both concrete (raw) and abstract (parsed) interpretations, a property common to algebraic specification languages. Moreover, SDF2 provides language designers with a variety of tools based on a declarative SDF2 specification. Additionally, SDF2 specifications, like DDC$^\alpha$ types, are scannerless – that is, they do not require a separate lexer – and support polymorphic syntax definitions [Visser 1998].

For all of SDF2's power and overlap with DDC$^\alpha$ features, it lacks some of the essential features that make DDC$^\alpha$ uniquely suited to data description languages: support for dependency, an explicit specification of the connection between DDC$^\alpha$ types and the underlying host language, and an explicit accounting of error-handling.

Parser Combinators. Of all parsing technologies, the DDC$^\alpha$ most closely resembles libraries of functional parser combinators, which have been extensively studied in the literature, dating back at least as early as 1975 [Burge 1975]. In particular, the parsing semantics of the DDC$^\alpha$ could rather easily be redefined in terms of monadic parser combinators, like those of the popular Parsec library [Leijen and Meijer 2001]. Indeed, Oury *et al.* [Oury and Swierstra 2008] have presented a reformulation of our theory along these lines by embedding DDC$^\alpha$ into the dependently-typed programming language Agda. However, an essential feature of DDC$^\alpha$ that distinguishes it from parser combinator libraries is its simultaneous interperetation of type declarations as parsers and as internal representation types. Moreover, this dual semantics has quite an impact on the user experience — the "look and feel" of DDC$^\alpha$, and related systems such as PADS and PACKETTYPES, is quite different from Parsec, for instance, because these systems exploit programmer intuitions concerning the meaning of types directly. This makes such languages a good fit for users that have not been exposed to combinator libraries before.

Despite this principal difference, we believe it is important to compare and contrast DDC$^\alpha$ with the literature on parser combinators in some depth. To do so, we begin by noting the salient distinguishing characteristics of many parser combinator libraries and then note where DDC$^\alpha$ fits with regard to these characteristics.

(1) Are alternatives explored depth-first or breadth-first?
(2) How much lookahead is supported?
(3) What are the semantics of choice?
(4) Does the algorithm support ambiguity?
(5) Does the algorithm support left recursion?
(6) How does the parser handle errors in the input?
(7) Does the library support context-sensitive parsing?

The first two questions are closely related, because lookahead is often integrated with the parsing process by speculatively continuing parsing at each branch point. Therefore, the question becomes: is lookahead performed breadth-first or depth-first? DDC$^\alpha$ uses a depth first approach to parsing alternatives – it tries the branches of each alternative in order, choosing the first branch to parse succesfully. Therefore, it supports unlimited lookahead, because each branch can consume arbitrary quantities of input. The depth-first approach to alternatives is quite common in parser combinators [Wadler 1985; Hutton 1992; Hutton and Meijer 1998; Fokker 1995]. Some libraries support a combination of these approaches. Parsec, for example, employs a breadth-first, single-token lookahead as standard, but also allows explicit invocation of depth-first, arbitrary-length lookahead through the `try` combinator [Leijen and Meijer 2001]. Swierstra *et al.* explore more sophisticated breadth-first parsing, based on continuations, in a series of papers [Swierstra and Duponcheel 1996; Swierstra 2001; Hughes and Swierstra 2003].

The third question to consider is the semantics of the choice operator. In DDC$^\alpha$, the choice operator is deterministic and greedy: it accepts the first branch that succeeds, even

if accepting a later branch might ultimately lead to a longer total parse. While this form of choice is limiting, it reflects the reality of what is supported by existing data description languages. This semantics is a common choice in parser combinator libraries. For example `try p <|> q` in Parsec and `p +++ q` in Hutton and Meijer's combinators [Hutton and Meijer 1998], both behave like DDC$^{\alpha}$'s choice operator.

Regarding the question of support for ambiguity, DDC$^{\alpha}$'s support for only deterministic choice removes the possibility of ambiguous grammars. In contrast, many of the basic parser combinator formulations since Wadler [Wadler 1985] support ambiguity and return all possible parses. However, given the efficiency impacts of such an approach, later work tries to limit the amount of ambiguity supported [Swierstra and Alcocer 1999; Leijen and Meijer 2001], provide the user with more fine grained control over its use [Hughes and Swierstra 2003], or generally improve the efficiency of ambiguous parsers [Peake and Seefried 2004; Frost et al. 2008].

Left-recursion is not supported by most parser combinators, and the DDC$^{\alpha}$ is no exception. This shortcoming is mitigated by the fact that most instances of left recursion can be elegantly rewritten using some form of repetition operator, like DDC$^{\alpha}$'s `seq` type or Fokker's `listOf` and `chainl` combinators [Fokker 1995]. However, there are new techniques for directly supporting left recursion in parser combinators [Frost et al. 2008].

Two of the essential features of DDC$^{\alpha}$ parsers are the detailed error reporting through parse descriptors and the robustness to errors. The various parser combinator libraries take a variety of approaches to error handling and reporting, none of them quite like DDC$^{\alpha}$'s. The Parsec library uses predictive parsing to ensure that when an error is encountered, it is clear exactly where in the input the error occurred. However, this advantage comes at the cost of requiring the grammar to be in (almost) LL(1) form. Moreover, when an error occurs, parsing stops. Swierstra, *et al.* [Swierstra and Alcocer 1999], similarly rely on predictive parsing to pinpoint errors, but add error correction – through token insertion and deletion – to increase parser robustness. All corrective actions are reported to the user via error messages. Later variations of this approach [Swierstra and Duponcheel 1996; Swierstra 2001] eliminate the requirement of predictive parsing by performing a breadth first search of all possible parses (including error correcting parses). These combinators also go beyond earlier ones by reporting all errors, with corresponding corrections, in a special-purpose data structure, rather than simple strings.

The approach to error correction taken by Swierstra *et al.* is closely related to that of DDC$^{\alpha}$. Both attempt to recover from errors via a combination of terminal and nonterminal insertion and terminal deletion. Both provide detailed reports as to the nature and location of errors and the corresponding corrective actions. Nevertheless, the differences are significant. First, they differ in their approach to choosing particular insertions and deletions. In DDC$^{\alpha}$, insertions happen implicitly and for any type – when a parser returns data with errors and parsing continues, an insertion has implicitly occured. Deletion points, however, must be marked explicitly, through the `scan` type. In contrast, Swierstra *et al.* completely automate the choice of insertion and deletion, relying, in part, on an analysis of the input grammar. The second difference lies in the nature of the error-reporting data structure. In DDC$^{\alpha}$, that data structure is the parse descriptor, and is specialized to the input grammar. As a result, the error data structure reflects the shape of the output data. In contrast, Swierstra *et al.* employ a single data structure for all error reporting, and relate errors to the raw input data, rather than the structured output data. This difference in error reporting is

necessary, in part, because parse combinators abstract over the structure of the output data.

The final distinguishing characteristic of parser-combinator libraries is support for context sensitive parsing. Leijen and Meijer [Leijen and Meijer 2001] distinguish between *monadic-style* combinators, like Parsec and those of Hutton and Meijer [Hutton and Meijer 1998], which support context sensitive parsing, and *arrow-style* combinators, like those of Swierstra *et al.*, which do not. The conventional wisdom is that monadic-style combinators are not amenable to the analyses employed for arrow-style combinators [Leijen and Meijer 2001; Swierstra and Alcocer 1999]. DDC$^{\alpha}$ is most similiar to monadic-style combinators.

Marshalling and Unmarshalling. Marshalling libraries such as Java's JXM library [JXM 2003] allow programmers to serialize objects on disk in a fixed format. Unmarshalling libraries read this fixed format back into memory. Although useful for saving or otherwise communicating the state of a program, this technology does not help solve the problem of how to interpret data that arrives in a non-standard, ad hoc format.

Languages such as ASDL [ASDL] and ASN.1 [Dubuisson 2001] are somewhat related to marshallers. Both of these languages specify the *logical* in-memory representation of data and then automatically generate a *physical* on-disk representation. Another language in this category is the Hierarchical Data Format 5 (HDF5) [Hierarchical Data Format 5 2007]. This file format allows users to store scientific data, but it does not help users deal with legacy ad hoc formats. Like marshalling tools, ASDL, ASN.1 and related technologies do not help users who need to parse and process non-standard, ad hoc data.

Type-Directed Programming. Type-directed or *generic* programming techniques [Jansson and Jeuring 1997; 1999; Jansson 2000; Hinze 2000; Jansson and Jeuring 2002; Lämmel and Peyton Jones 2003] allow users to define algorithms by induction over the structure of a type rather than by induction (or recursion) over the structure of a value. Of particular interest is the elegant work by Jansson and Jeuring on polytypic data conversions [Jansson and Jeuring 1997; 1999; Jansson 2000; Jansson and Jeuring 2002]. These authors demonstrate how to program a variety of data transformation functions together with their inverses in PolyP, a type-directed extension of Haskell. For instance, they describe a generic compressing printing/parsing algorithm, a generic noncompressing printing/parsing algorithm, and a data extraction algorithm that separates primitive data from its containing structure.

Also of interest is the work of van Weelden *et al* [van Weelden et al. 2005]. They investigated the use of type-directed 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.

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, there are a number of reasons why one might prefer a domain-specific language like PADS or DDC$^{\alpha}$ over a full-blown generic programming framework. From a programmer's perspective, the specialized syntax makes writing descriptions, particularly descriptions with nested literals, regular expressions, functions and dependencies, relatively easy. From an implementer's perspective, PADS is a relatively simple, light-weight language extension: Implementing a PADS-style language for any standard imperative, functional or object-oriented language requires no changes to the underlying host language type system or run-time. In contrast,

type-directed programming languages normally need sophisticated, non-standard type systems or modifications to the run-time to function correctly. In the paper "Generics for the Masses" [Hinze 2004], Hinze cites these complications as his motivation for the design of a new generic programming environment for Haskell, but unfortunately, the new design is still Haskell-specific, as it makes essential use of polymorphic data structures and type classes.

XML-based tools. Rather than programming directly with data in its ad hoc format, it may be useful to convert it first 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, for programmers not familiar with XML, there is a high barrier to entry — not only do they have to learn the ad hoc format, but they must also learn XML and the XML conversion tool. Altogether, this overhead is too heavy for many simple data processing tasks.

Two other related XML-based specification languages are DFDL [DFDL 2005; Beckerle and Westhead 2004] and XDTM [Moreau et al. 2005; Zhao et al. 2005]. Like PADS or PACKETTYPES, DFDL is a language for specifying data formats. It has a rich collection of base types and supports a variety of ambient codings. Early versions of DFDL did not allow dependent constraints, but they were later added, perhaps because PADS had demonstrated how effective they can be. 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 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 database systems, but these products typically support only a limited number of formats. Also, they do not expose a declarative description of the original format for use apart from the database, and they provide only fixed methods for coping with erroneous data. For these reasons, type-based data description languages are 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.

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 type-based 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 type-based 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. 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 33$^{\text{rd}}$ 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.

BERGSTRA, J. A., HEERING, J., AND KLINT, P. 1989. *Algebraic Specification*. ACM Press Frontier Series. ACM Press in co-operation with Addison-Wesley, Chapter 1, 1–66.

BIRD, R. AND MEERTENS, L. 1998. Nested datatypes. In *Proceedings 4th Int. Conf. on Mathematics of Program Construction, MPC'98, Marstrand, Sweden, 15–17 June 1998*, J. Jeuring, Ed. Vol. 1422. Springer-Verlag, Berlin, 52–67.

BIRMAN, A. AND ULLMAN, J. D. 1973. Parsing algorithms with backtrack. *Information and Control 23,* 1 (Aug.), 1–34.

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.

FOKKER, J. 1995. Functional parsers. In *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques-Tutorial Text*. Springer-Verlag, London, UK, 1–23.

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.

FROST, R. A., HAFIZ, R., AND CALLAGHAN, P. 2008. Parser combinators for ambiguous left-recursive grammars. In *Practical Aspects of Declarative Languages*. Lecture Notes in Computer Science. Springer.

GIRARD, J.-Y. 1972. Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur. Ph.D. thesis, Thèse d'état, Université Paris VII. Summary in *Proceedings of the Second Scandinavian Logic Symposium* (J.E. Fenstad, editor), North-Holland, 1971 (pp. 63-92).

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 2007. 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.

HINZE, R. 2004. Generics for the masses. In *ACM International Conference on Functional Programming*. 236–243.

HUGHES, R. J. M. AND SWIERSTRA, S. D. 2003. Polish parsers, step by step. In *ICFP '03: Proceedings of the eighth ACM SIGPLAN international conference on Functional programming*. ACM, New York, NY, USA, 239–248.

HUTTON, G. 1992. Higher-order Functions for Parsing. *Journal of Functional Programming 2,* 3 (July), 323–343.

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. Featherweight Java: a minimal core calculus for Java and GJ. In *ACM Conference on Object-oriented Programming, Systems, Languages, and Applications*. ACM Press, 132–146.

JAMES, R. AND MALPANI, P. 2003. Enter the data definition language: A developer perspective. *.NET Developers's Journal*.

JANSSON, P. 2000. Functional polytypic programming. Ph.D. thesis, Chalmers University of Technology and Göteborg University.

JANSSON, P. AND JEURING, J. 1997. PolyP - a polytypic programming language extension. In *POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM Press, 470–482.

JANSSON, P. AND JEURING, J. 1999. Polytypic compact printing and parsing. In *European Symposium on Programming*. Number 1576 in Lecture Notes in Computer Science. 273–287.

JANSSON, P. AND JEURING, J. 2002. Polytypic data conversion programs. *Science of Computer Programming 43,* 1, 35–75.

JXM 2003. Java XML mapping. `http://jxm.sourceforge.net/manual.html`.

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. In *ACM SIGPLAN Workshop on Types in Language Design and Implementation*. ACM Press, 26–37.

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

LEIJEN, D. AND MEIJER, E. 2001. Parsec: Direct style monadic parser combinators for the real world. Tech. Rep. UU-CS-2001-27, Department of Computer Science, Universiteit Utrecht.

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, 77–84.

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 data 2003. Tree formats. Workshop on Molecular Evolution web site. `http://workshop.molecularevolution.org/resources/fileformats/tree_formats.php`.

OURY, N. AND SWIERSTRA, W. 2008. The power of pi. In *ICFP '08: Proceedings of the 13th ACM SIGPLAN international conference on Functional programming*. ACM, New York, NY, USA, 39–50.

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

PEAKE, I. AND SEEFRIED, S. 2004. A combinator parser for earley's algorithm. `http://goanna.cs.rmit.edu.au/~ipeake/pubs/earley-cps.pdf`. Work in progress.

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

REYNOLDS, J. C. 1974. Towards a theory of type structure. In *Paris Colloquium on Programming*. Springer-Verlag, 141–156.

RUS, T. 1972. ΣS-Algebra of a formal language. *Bulletin Mathematique de la Societe de Science, Bucharest*.

SWIERSTRA, S. D. 2001. Combinator parsers: From toys to tools. In *2000 ACM SIGPLAN Haskell Workshop*. Electronic Notes in Theoretical Computer Science, vol. 41. Elsevier, 38–59.

SWIERSTRA, S. D. AND ALCOCER, P. R. A. 1999. Fast, error correcting parser combinatiors: A short tutorial. In *SOFSEM '99: Proceedings of the 26th Conference on Current Trends in Theory and Practice of Informatics on Theory and Practice of Informatics*. Springer-Verlag, London, UK, 112–131.

SWIERSTRA, S. D. AND DUPONCHEEL, L. 1996. Deterministic, error-correcting combinator parsers. In *Advanced Functional Programming, Second International School-Tutorial Text*. Springer-Verlag, London, UK, 184–207.

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.

VISSER, E. 1997. Syntax definition for language prototyping. Ph.D. thesis, University of Amsterdam.

VISSER, E. 1998. Polymorphic syntax definition. *Theoretical Computer Science 199*, 57–86.

WADLER, P. 1985. How to replace failure with a list of successes. In *Functional Programming Languages and Computer Architecture*. Lecture Notes in Computer Science, vol. 201. Springer-Verlag, 113–128.

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.

A. EXTENDED METATHEORY

In this appendix, we formally state and prove the theorems of Section 5. To start, we state some basic assumptions. First, we assume that all variable names introduced by the parsing semantics function come from a separate syntactic domain from the variables that appear in ordinary expressions. These names are therefore 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. Note that that the interface $\mathcal{B}_{\text{opty}}$ specifies the types of base-type parsers.

Condition 9 (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, we define in two judgments that respectively generalize evaluation ($e \to_k e'$) and normalization ($\tau \to_k \tau'$) to k steps:

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

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

Some useful properties of these judgments follow.

Lemma 10 (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* $e_2 \to_k e_2'$ *then* $v\, e_2 \to_k v\, e_2'$.
(3) *If* $e \to_k e'$ *then* $e\,[\sigma] \to_k e'\,[\sigma]$.
(4) *If* $e_1 \to_i e_2$ *and* $e_2 \to_j e_3$ *then* $e_1 \to_{(i+j)} e_3$.

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

Lemma 11 (Properties of K-step Normalization)
(1) *If* $\tau_1 \to_k \tau_1'$ *then* $\tau_1\, \tau_2 \to_k \tau_1'\, \tau_2$.
(2) *If* $\tau_2 \to_k \tau_2'$ *then* $\nu\, \tau_2 \to_k \nu\, \tau_2'$.
(3) *If* $\tau_1 \to_k \tau_1'$ *then* $\tau_1\, e \to_k \tau_1'\, e$.
(4) *If* $e \to_k e'$ *then* $\nu\, e \to_k \nu\, e'$.
(5) *If* $\tau_1 \to_i \tau_2$ *and* $\tau_2 \to_j \tau_3$ *then* $\tau_1 \to_{(i+j)} \tau_3$.

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

Lemma 12 (K-step Evaluation Inversion)
(1) If $e_1\, e_2 \to_k v$ *then* $k > 0$ *and* $\exists\, i, j, v_1, v_2$ *s.t.* $e_1 \to_i v_1$ *and* $e_2 \to_j v_2$, *with* $i+j < k$.
(2) If $e\,[\sigma] \to_k v$ *then* $\exists\, i, v'$ *s.t.* $e \to_i v'$, *with* $i < k$.
(3) If $(\texttt{fun}\ f\ x = e)\ v \to_k v'$ *then* $e[(\texttt{fun}\ f\ x = e)/f][v/x] \to_{k-1} v'$.
(4) If $\texttt{let}\ x = e\ \texttt{in}\ e' \to_k v$ *then* $\exists\, i, v'$ *s.t.* $e \to_i v'$ *with* $i < k$.
(5) If $\texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \to_k v$ *and* $e \to^* \texttt{true}$ *then* $\exists\, i$ *s.t.* $e_1 \to_i v$ *with* $i < k$.
(6) If $\texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \to_k v$ *and* $e \to^* \texttt{false}$ *then* $\exists\, i$ *s.t.* $e_2 \to_i v$ *with* $i < k$.

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

Lemma 13 (Confluence of Evaluation)
If $e \to_k v$ *and* $e \to_i e'$ *then* $e' \to_{k-i} v$.

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

Lemma 15 (Properties of Expression Equivalence)
(1) If $e \equiv e'$ *and* $e \to_k e_1$ *then* $\exists\, e_1'$ *s.t.* $e' \to_k e_1'$ *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. □

We will also require the following two standard properties of F_ω type equivalence.

Lemma 16 (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:

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

Lemma 17 (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* $[\![\tau[\tau'/\alpha]]\!]_{rep} = [\![\tau]\!]_{rep}[[\![\tau']\!]_{rep}/\alpha_{\texttt{rep}}]$.
(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 $\texttt{parse}_{\alpha}$ cannot appear in any τ – they can only be introduced by the parsing semantics function. For part 7, note that the open variables in $[\![\tau]\!]_{\mathrm{P}}$ are exactly those that are open in τ itself, as none are introduced in the translation. □

We also require a similar commutativity result for the $[\![\,\cdot : \cdot\,]\!]_{\mathrm{PT}}$ function.

Lemma 18
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 17 for T case. □

Lemma 19
The function $[\![\,\cdot\,]\!]_{rep}$ *is total.*

PROOF. By induction on the structure of types. □

We are now in a position to present some standard type-theoretic results for DDC$^{\alpha}$ kinding and normalization, as well as key substitution lemmas.

Lemma 20 (DDC$^{\alpha}$ Preservation)
If $\vdash \tau : \kappa$ *and* $\tau \rightarrow^{*} \nu$ *then* $\vdash \nu : \kappa$.

PROOF. By induction on the kinding derivation. □

Lemma 21 (DDC$^{\alpha}$ Inversion)
All kinding rules are invertable. That is, given a proof of any rule's conclusion we have a proof of the rule's premises.

PROOF. By inspection of the kinding rules; in particular, the fact that they are syntax directed. □

Lemma 22 (DDC$^{\alpha}$ Canonical Forms)
If $\vdash \nu : \kappa$ *then either*

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

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

Lemma 23 (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 24 (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 23 as needed. □

Finally, we state another commutativity property for the semantic functions. In essence, it says that evaluation 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.

Lemma 25 (Commutativity of Evaluation and Semantic Interpretation)
If $\vdash \tau : \kappa$ *and* $[\![\tau]\!]_P \to^* v$ *then* $\exists\, \nu$ *such that*

(1) $\tau \to^* \nu$,
(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. □

A.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 26. Lemma 30 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 26
—$\mathrm{H}(\tau : \mathsf{T})$ *iff* $\exists\, \sigma$ *s.t.* $[\![\tau]\!]_{PD} \equiv \mathtt{pd\_hdr} * \sigma$.
—$\mathrm{H}(\tau : \mathsf{T} \to \kappa)$ *iff* $\exists\, \sigma$ *s.t.* $[\![\tau]\!]_{PD} \equiv \sigma$ *and* $\forall\, \tau'.\, \mathrm{H}(\tau' : \mathsf{T})$ *implies* $\mathrm{H}(\tau\ \tau' : \kappa)$.
—$\mathrm{H}(\tau : \sigma \to \kappa)$ *iff* $\exists\, \sigma'$ *s.t.* $[\![\tau]\!]_{PD} \equiv \sigma'$ *and* $\mathrm{H}(\tau\ e : \kappa)$ *for any expression* e.

Lemma 27
If $\mathrm{H}(\tau : \mathsf{T})$ *then* $\exists\, \sigma\ s.t. [\![\tau]\!]_{PD} = \sigma$.

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

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 28
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. □

Lemma 29
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 30
If $\Delta; \Gamma \vdash \tau : \kappa$ *then* $\mathrm{H}(\tau : \kappa)$.

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

Corollary 31
—*If* $\Delta; \Gamma \vdash \tau : \kappa$ *then* $\exists \sigma. [\![\tau]\!]_{PD} = \sigma$.
—*If* $\Delta; \Gamma \vdash \tau : \mathsf{T}$ *then* $\exists \sigma. [\![\tau]\!]_{PD} \equiv \mathtt{pd\_hdr} * \sigma$.

PROOF. Immediate from definition of $\mathrm{H}(\tau : \kappa)$ and Lemma 30. □

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 32 (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 33 (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 30 and Lemma 16, 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^{α} type. For clarity, we abbreviate $\texttt{pd\_hdr} * \sigma$ as $\sigma\,\texttt{pd}$.

Lemma 34 (Types of Constructors)
—$\mathtt{R}_{\texttt{unit}} : \texttt{unit} \to \texttt{unit}$
—$\mathtt{P}_{\texttt{unit}} : \texttt{offset} \to \texttt{pd\_hdr} * \texttt{unit}$
—$\mathtt{R}_{\texttt{bottom}} : \texttt{unit} \to \texttt{none}$
—$\mathtt{P}_{\texttt{bottom}} : \texttt{offset} \to \texttt{pd\_hdr} * \texttt{unit}$
—$\mathtt{R}_{\Sigma} : \forall \alpha, \beta. \alpha * \beta \to \alpha * \beta$
—$\mathtt{P}_{\Sigma} : \forall \alpha, \beta. \alpha\,\texttt{pd} * \beta\,\texttt{pd} \to (\alpha\,\texttt{pd} * \beta\,\texttt{pd})\,\texttt{pd}$
—$\mathtt{R}_{+\texttt{left}} : \forall \alpha, \beta. \alpha \to \alpha + \beta$
—$\mathtt{R}_{+\texttt{right}} : \forall \alpha, \beta. \beta \to \alpha + \beta$
—$\mathtt{P}_{+\texttt{left}} : \forall \alpha, \beta. \alpha\,\texttt{pd} \to \texttt{pd\_hdr} * (\alpha\,\texttt{pd} + \beta\,\texttt{pd})$
—$\mathtt{P}_{+\texttt{right}} : \forall \alpha, \beta. \beta\,\texttt{pd} \to \texttt{pd\_hdr} * (\alpha\,\texttt{pd} + \beta\,\texttt{pd})$
—$\mathtt{R}_{\&} : \forall \alpha, \beta. \alpha * \beta \to \alpha * \beta$
—$\mathtt{P}_{\&} : \forall \alpha, \beta. \alpha\,\texttt{pd} * \beta\,\texttt{pd} \to \texttt{pd\_hdr} * (\alpha\,\texttt{pd} * \beta\,\texttt{pd})$.
—$\mathtt{R}_{\texttt{con}} : \forall \alpha. \texttt{bool} * \alpha \to \alpha + \alpha$
—$\mathtt{P}_{\texttt{con}} : \forall \alpha. \texttt{bool} * \alpha\,\texttt{pd} \to \texttt{pd\_hdr} * \alpha\,\texttt{pd}$
—$\mathtt{R}_{\texttt{seq\_init}} : \forall \alpha. \texttt{unit} \to \texttt{int} * \alpha\,\texttt{seq}$
—$\mathtt{P}_{\texttt{seq\_init}} : \forall \alpha. \texttt{offset} \to \texttt{pd\_hdr} * (\alpha\,\texttt{pd}\ \texttt{arr\_pd})$
—$\mathtt{R}_{\texttt{seq}} : \forall \alpha. (\texttt{int} * \alpha\,\texttt{seq}) * \alpha \to \texttt{int} * \alpha\,\texttt{seq}$
—$\mathtt{P}_{\texttt{seq}} : \forall \alpha_{elt}, \alpha_{sep}. (\texttt{pd\_hdr} * (\alpha_{elt}\,\texttt{pd}\ \texttt{arr\_pd})) * \alpha_{sep}\,\texttt{pd} * \alpha_{elt}\,\texttt{pd} \to$
$\texttt{pd\_hdr} * (\alpha_{elt}\,\texttt{pd}\ \texttt{arr\_pd})$
—$\mathtt{R}_{\texttt{compute}} : \forall \alpha. \alpha \to \alpha$
—$\mathtt{P}_{\texttt{compute}} : \texttt{offset} \to \texttt{pd\_hdr} * \texttt{unit}$
—$\mathtt{R}_{\texttt{absorb}} : \forall \alpha. \alpha\,\texttt{pd} \to \texttt{unit} + \texttt{none}$
—$\mathtt{P}_{\texttt{absorb}} : \forall \alpha. \alpha\,\texttt{pd} \to \texttt{pd\_hdr} * \texttt{unit}$
—$\mathtt{R}_{\texttt{scan}} : \forall \alpha. \alpha \to \alpha + \texttt{none}$
—$\mathtt{P}_{\texttt{scan}} : \forall \alpha. \texttt{int} * \alpha\,\texttt{pd} \to \texttt{pd\_hdr} * ((\texttt{int} * \alpha\,\texttt{pd}) + \texttt{unit})$

—$\mathtt{R}_{\mathtt{scan\_err}} : \forall\alpha.\mathtt{unit} \to \alpha + \mathtt{none}$
—$\mathtt{P}_{\mathtt{scan\_err}} : \forall\alpha.\mathtt{offset} \to \mathtt{pd\_hdr} * ((\mathtt{int} * \alpha) + \mathtt{unit})$

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 35 (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 36 (Type Correctness of Closed Types)
If $\vdash \tau : \kappa$ *then* $\vdash [\![\tau]\!]_P : [\![\tau{:}\kappa]\!]_{PT}$.

A.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 $\mathtt{pos}$ to denote the function that returns zero when passed zero and one when passed another natural number.

Definition 37 (Canonical Forms I)
$\mathrm{Canon}_\nu(r, p)$ *iff exactly one of the following is true:*

—$\nu = \mathtt{unit}$ *and* $r = ()$ *and* $p.nerr = 0$.
—$\nu = \mathtt{bottom}$ *and* $r = \mathtt{none}$ *and* $p.nerr = 1$.
—$\nu = C(e)$ *and* $r = \mathtt{inl}\ c$ *and* $p.nerr = 0$.
—$\nu = C(e)$ *and* $r = \mathtt{inr}\ \mathtt{none}$ *and* $p.nerr = 1$.
—$\nu = \Sigma\, x{:}\tau_1.\tau_2$ *and* $r = (r_1, r_2)$ *and* $p = (h, (p_1, p_2))$ *and* $h.nerr = \mathtt{pos}(p_1.nerr) + \mathtt{pos}(p_2.nerr)$, $\mathrm{Canon}^*{}_{\tau_1}(r_1, p_1)$ *and* $\mathrm{Canon}^*{}_{\tau_2[(r_1,p_2)/x]}(r_2, p_2)$.

—$\nu = \tau_1 + \tau_2$ *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* $e[(r', p')/x] \to^* \texttt{true}$.

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

—$\nu = \tau_e\ \texttt{seq}(\tau_s, e, \tau_t,)$, $r = (len, [\vec{r_i}])$, $p = (h, (neerr, len', [\vec{p_i}]))$, $neerr = \sum_{i=1}^{len} \texttt{pos}(p_i.nerr)$, $len = len'$, $\mathrm{Canon}^*{}_{\tau_e}(r_i, p_i)$ *(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 38 (Canonical Forms II)
$\mathrm{Canon}^*{}_{\tau}(r, p)$ *iff* $\tau \to^* \nu$ *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 39 (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* $\mathrm{Canon}^*{}_{\tau_2[(r_1,p_1)/x]}(r_2, p_2)$ *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* $e[(r, p)/x] \to^* c$ *then* $\mathrm{Canon}_{\{x:\tau \mid e\}}(\mathtt{R}_{\texttt{set}}(\mathtt{c}, \mathtt{r}), \mathtt{P}_{\texttt{set}}(\mathtt{c}, \mathtt{p}))$

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

—*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* $\mathrm{Canon}_{\mathtt{scan}(\tau)}(\mathtt{R}_{\mathtt{scan}}(\mathtt{r}),\mathtt{P}_{\mathtt{scan}}(\mathtt{i},\mathtt{p}))$.
—$\mathrm{Canon}_{\mathtt{scan}(\tau)}(\mathtt{R}_{\mathtt{scan\_err}}(),\mathtt{P}_{\mathtt{scan\_err}}(\omega))$.

PROOF. By inspection of the constructor functions. □

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

Condition 40 (Base Type Parsers Produce Values in Canonical Form)
If $\vdash v : \sigma$, $\mathcal{B}_{kind}(C) = \sigma \to \mathsf{T}$ *and* $\mathcal{B}_{imp}(C)\ v\ (B,\omega) \to^*\ (\omega', r, p)$ *then* $\mathrm{Canon}_{C(v)}(r,p)$.

Theorem 41 is our final result. It 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.

Theorem 41 (Parsing to Canonical Forms)
If $\vdash \tau : \mathsf{T}$ *and* $[\![\tau]\!]_P\ (B,\omega) \to^*\ (\omega', r, p)$ *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. □