The Next 700 Data Description Languages

Kathleen Fisher
AT&T Labs Research
kfisher@research.att.com

Yitzhak Mandelbaum
Princeton University
yitzhakm@CS.Princeton.EDU

David Walker
Princeton University
dpw@CS.Princeton.EDU

Abstract

In the spirit of Landin, we present a calculus of dependent types to serve as the semantic foundation for a family of languages called *data description languages*. Such languages, which include PADS, DATASCRIPT, and PACKETTYPES, are designed to facilitate programming with *ad hoc data*, *i.e.*, data not in well-behaved relational or XML formats. In the calculus, each type describes the physical layout and semantic properties of a data source. In the semantics, we interpret types simultaneously as the in-memory representation of the data described and as parsers for the data source. The parsing functions are robust, automatically detecting and recording errors in the data stream without halting parsing. We show the parsers are type-correct, returning data whose type matches the simple-type interpretation of the specification. We also prove the parsers are "error-correct," accurately reporting the number of physical and semantic errors that occur in the returned data. We use the calculus to describe the features of various data description languages, and we discuss how we have used the calculus to improve PADS.

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.

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

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 hierachy
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

Figure 1. Selected ad hoc data sources.

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.

1.1 Promising Solutions

To address these challenges, researchers have begun to develop high-level languages for describing and processing ad hoc data. For

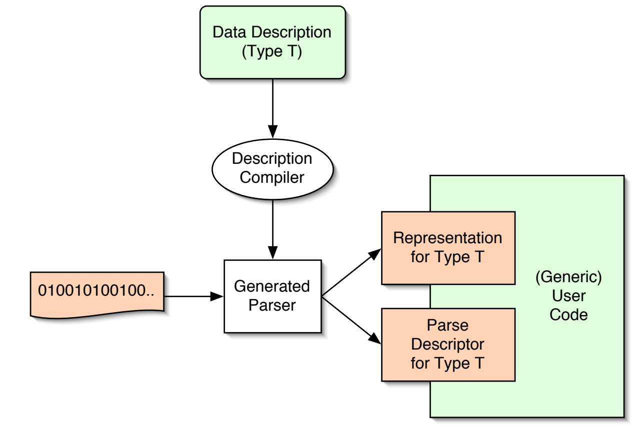

Figure 2. Architecture of PADS system.

instance, McCann and Chandra introduced PACKETTYPES [14], a specification language designed to help systems programmers process the binary data associated with networking protocols. Godmar Back developed DATASCRIPT [1], a scripting language with explicit support for specifying and parsing binary data formats. DATASCRIPT has been used to manipulate Java jar files and ELF object files. The developers of Erlang have also introduced language extensions that they refer to as *binaries* [17, 10] to aid in packet processing and protocol programming. At CMU, Eger is in the process of developing a language of Bit-Level Types [4] for specifying file formats such as ELF, JPEG, and MIDI as well as packet layouts. Finally, we are part of a group developing PADS [6], another system for specifying 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.

While differing in many details, these languages 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 changes, as they frequently do, by extending 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 naturally used to describe sequences of data objects. And, really, what else could an array type describe? Similarly, 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[5]. 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, BLT, 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.
>
> — J. P. Landin, *The Next 700 Programming Languages*, 1965.

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

The primary goal of this paper is to begin to understand the family of ad hoc data processing languages. We do so, as Landin did, by developing a semantic framework for defining, comparing,

and contrasting languages in our domain. This semantic framework revolves around the definition of a data description calculus (DDC). This calculus uses types from a dependent type theory to describe various forms of ad hoc data: base types to describe atomic pieces of data and type constructors to describe richer structures. We show how to give a denotational semantics to DDC by interpreting types as parsing functions that map external representations (bits) to data structures in a typed lambda calculus. More precisely, these parsers produce both internal representations of the external data and parse descriptors that pinpoint errors in the original source.

For many domains, researchers have a solid understanding of what makes a "good" or "bad" language. For instance, a good 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 "good" or "bad." 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 all errors in the parsed data be accurately recorded in the parse descriptor. We adopted this criterion because it ensures that data consumers can rely on the integrity of data marked as error-free.

To study and compare PADS, DATASCRIPT, and/or some other data description language, we advocate translating the language into DDC. The translation decomposes the relatively complex, high-level descriptions of the language in question into a series of lower-level DDC descriptions, which have all been formally defined. We have done this decomposition for IPADS, an idealized version of the PADS language that captures the essence of the actual implementation. We have also analyzed many of the features of 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 we had available to us.

To our delight, the process of giving PADS a semantics in this framework has had additional benefits. In particular, since we defined the semantics by reviewing the existing implementation, we found (and fixed!) a couple of subtle bugs. The semantics has also raised several design questions that we are continuing to study. It has also helped us explore important extensions. In particular, driven by examples found in biological data [3, 15], we decided to add recursion to PADS. We used our semantic framework to study the ramifications of this addition.

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

- We define a semantic framework for understanding and comparing data description languages such as PADS, PACKETTYPES, DATASCRIPT, and BLT. No one has previously given a formal semantics to any of these languages.
- At the center of the framework is DDC, a calculus of data descriptions based on dependent type theory. We give a denotational semantics to DDC 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 parsers maintain this property.
- We define IPADS, an idealized version of the PADS programming language that captures its essential features, and show how to give it a semantics by translating it into DDC. 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 to experiment with a definition and implementation strategy for recursive data types, a feature not found in any existing ad hoc data description language that we are aware of. Recursive types are essential for representing tree-shaped hierarchical data [3, 15]. We have integrated recursion into PADS, using our theory as a guide.

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. Section 6 discusses encodings of IPADS, PACKETTYPES and DATASCRIPT in DDC and Section 7 explains how we have already made use of our semantics in practice. Sections 8 and 9 discuss related work and conclude.

2. IPADS: An Idealized DDL

In this section, we define IPADS, an idealized data description language. IPADS captures the essence of PADS in a fashion similar to the way that MinML [11] captures the essence of ML or Featherweight Java [12] captures the essence of Java. The main goal of this section is to introduce the reader to the form and function of IPADS by giving its syntax and walking through a couple of examples. Though the syntax differs, the structure of PADS' relatives BLT, PACKETTYPES, and DATASCRIPT are similar. Later sections will show how to give a formal semantics to IPADS.

Preliminary Concepts. Like PADS, PACKETTYPES, DATASCRIPT, and BLT, IPADS data descriptions are types. These types specify both the external data format (a sequence of bits or characters) and a mapping into a data structure in the host programming language. In PADS, 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 32-bit integer), characters (**`Pchar`**), strings (**`Pstring`**), dates (**`Pdate`**), IP addresses (**`Pip`**), and others. In general, these base types may parameterized. For instance, we will assume **`Pstring`** is parameterized by an argument that signals termination of the string. For example, **`Pstring`**`(" ")` describes any sequence of characters terminated by a space. (Note that we do not consider the space to be part of the parsed string; it will be part of the next object.) Similarly, **`Puint16_FW`**`(3)` is an unsigned 16-bit integer described in exactly 3 characters in the data source. In general, we write $C(e)$ for a base type parameterized by a (host language) expression e.

When interpreted as a parser, each of these base types reads the external data source and generates a pair of data structures in the host language. The first data structure is the *internal representation* and the second is the *parse descriptor*, which contains meta-data 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 [13], 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 **Peor**), and a description of the terminator (in this case, the end-of-file marker). PADS 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. In general, later fields in a **Pstruct** may reference fields appearing earlier.

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** 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-

```
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(Peor, Peof)
```

Figure 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
```

Figure 4. Sample Common Web Log Data. Each record is broken with a newline for formatting purposes.

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. The expression in the **Pwhere** clause is an arbitrary host language expression with boolean type.

A Recursive* IPADS *Example. Figure 5 presents a second IPADS example. In this example, IPADS describes the Newick format, a flat representation of tree-structured data. The leaves of the trees are names that describe an "entity". In our variant of Newick, leaf names may be omitted. If the leaf name does appear, it is followed by a colon and a number. The number describes the "distance" from the parent node. Microbiologists often use distances to describe the number of genetic mutations that have to occur to move from the parent to the child. An internal tree node may have any number of (comma-separated) children within parentheses. Distances follow the closed-paren of the internal tree node.

The Newick format and other formats that describe tree-shaped hierarchies [3, 15] 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. The rest of the format description uses types we have seen before such as **Pstruct**, **Parray**, and **Punion**.

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. 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**.

```
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; *)
```

Figure 5. IPADS Newick Format Description

Types	t	::=	$C(e)$ | **Plit** c
		|	**Pfun**$(x:\sigma) = t$ | $t\ e$
		|	**Pstruct**$\{\overrightarrow{x{:}t}\}$ | **Punion**$\{\overrightarrow{x{:}t}\}$
		|	t **Pwhere** $x.e$ | **Popt** t | t **Parray**(t,t)
		|	**Pcompute** $e{:}\sigma$ | α | **Prec** $\alpha.t$
Programs	p	::=	t | $\alpha = t;\ p$ | **Prec** $\alpha = t;\ p$

Figure 6. IPADS Syntax

Kinds	κ	::=	T | $\sigma \to \kappa$
Types	τ	::=	unit | bottom | $C(e)$ | $\lambda x.\tau$ | $\tau\ e$
		|	$\Sigma\, x{:}\tau.\tau$ | $\tau + \tau$ | $\tau \,\&\, \tau$ | $\{x{:}\tau \mid e\}$ | τ seq(τ, e, τ)
		|	α | $\mu\alpha.\tau$ | compute$(e{:}\sigma)$ | absorb(τ) | scan(τ)

Figure 7. DDC Syntax

Bits	B	::=	$\cdot$ | $0\,B$ | $1\,B$
Constants	c	::=	$()$ | true | false | 0 | 1 | -1 | ...
		|	none | B | ω | ok | err | fail | ...
Values	v	::=	c | fun $f\ x = e$ | (v, v)
		|	inl v | inr v | $[\vec{v}]$
Operators	op	::=	$=$ | $<$ | not | ...
Expressions	e	::=	c | x | $op(e)$ | fun $f\ x = e$ | $e\ e$
		|	let $x = e$ in e | if e then e else e
		|	(e, e) | $\pi_i\, e$ | inl e | inr e
		|	case e of (inl $x \Rightarrow e$ | inr $x \Rightarrow e$)
		|	$[\vec{e}]$ | e @ e | $e\,[e]$
Base Types	a	::=	unit | bool | int | none
		|	bits | offset | errcode
Types	σ	::=	a | α | $\sigma \to \sigma$ | $\sigma * \sigma$ | $\sigma + \sigma$
		|	σ seq | $\forall\alpha.\sigma$ | $\mu\alpha.\sigma$

Figure 8. Host Language

3. A Data Description Calculus

At the heart of our work is a data description calculus (DDC), designed to capture the core features of data description languages. Consequently, the syntax of DDC is at a significantly lower level of abstraction than that of IPADS. Like IPADS, however, DDC presents a type-based model. Each DDC 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. 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. We base our calculus on a dependent type theory because as we have seen, it is common in data description languages for expressions to appear within types.

3.1 DDC Syntax

Figure 7 shows the syntax of DDC. As with IPADS, expressions e and types σ belong to the host language, defined in Section 3.2. The most basic types are `unit` and `bottom`, both of which consume no input and return unit as their representation. The difference between them is that the former always succeeds, while the latter always fails, a distinction recorded in the associated parse descriptors. The syntax $C(e)$ denotes a base type C parameterized by expression e. The syntax α denotes a type variable introduced in a recursive type.

We provide abstraction $\lambda x.\tau$ and application $\tau\, e$ so that we may parameterize types by expressions. Dependent product types $\Sigma\, x{:}\tau_1.\tau_2$ describe a sequence of values in which the second type may refer to the value of the first. Sum types $\tau_1 + \tau_2$ express flexibility in the data format, as they parse data matching either τ_1 or τ_2. Sum-type parsers are deterministic, transforming the data according to τ_1 when possible and only attempting to use τ_2 if there is an error in τ_1. Intersection types $\tau_1 \,\&\, \tau_2$ describe data that match both τ_1 and τ_2. They transform a single set of bits to produce a pair of values, one from each type. Set types $\{x{:}\tau \mid e\}$ transform data according to the underlying type τ and then check that the constraint e holds when x is bound to the parsed value.

The type τ `seq`(τ_s, e, τ_t) represents a sequence of values of type τ. The type τ_s specifies the type of the separator found between elements of the sequence. For sequences without separators, we use `unit` as the separator type. Expression e is a boolean-valued function that examines the parsed sequence after each element is read to determine if the sequence has completed. For example, a function that checks if the sequence has 100 elements would terminate a sequence when it reaches length 100. The type τ_t denotes the type of the terminator expected after the last sequence element. For sequences without terminators, we use `bottom` for τ_t.

Recursive types $\mu\alpha.\tau$ describe recursive data formats. The name α can be used in τ to refer to the recursive type and causes a recursive call to τ's parser wherever it appears.

DDC also has a number of "active" types. These types describe actions to be taken during parsing rather than strictly describing the data format. Type `compute`$(e{:}\sigma)$ allows us to include an element in the parsed output that does not appear in the data stream (although it is likely dependent on elements that do), based on the value of expression e. In contrast, type `absorb`(τ) 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 `scan`(τ), 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 "recovery" type τ is found.

3.2 Host Language

In Figure 8, we present the host language of DDC, an extension of the simple-typed polymorphic lambda calculus. We use this host language both to encode the parsing semantics of DDC and to write the expressions that can appear within DDC 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 types. Our expressions include arbitrary length sequences $[\vec{e}]$, sequence append $e \mathbin{@} e'$, and sequence indexing $e\,[i]$.

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`; polymorphic types $\forall\alpha.\sigma$ and type variables α; and recursive types $\mu\alpha.\sigma$.

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

We use standard judgments for the static semantics ($\Gamma \vdash e : \sigma$) and operational semantics ($e \hookrightarrow e'$) of the host language. Details appear in Appendix A.

3.3 Example

As an example, we present an abbreviated description of the common log format as it might appear in DDC. 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 non-dependent products and introduce a number of type abbreviations in the form `name` $= \tau$ before giving the type that describes the data source.

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

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

$$\mathtt{response_t} = \lambda\mathtt{x}.\{\mathtt{y}{:}\mathtt{Puint16_FW(x)} \mid 100 \le \mathtt{y}\ \mathtt{and}\ \mathtt{y} < 600\}$$

$$\begin{array}{ll} \multicolumn{2}{l}{\mathtt{entry_t} =} \\ \quad\Sigma\,\mathtt{client}{:}\mathtt{Pip}. & \mathtt{S}(\text{“ ”}) \quad * \\ \quad\Sigma\,\mathtt{remoteid}{:}\mathtt{authid_t}. & \mathtt{S}(\text{“ ”}) \quad * \\ \quad\Sigma\,\mathtt{response}{:}\mathtt{response_t}\ 3. & \\ \qquad\mathtt{compute}(\mathtt{getdomain\ client} = \text{“edu”}{:}\mathtt{bool}) & \end{array}$$

$$\mathtt{entry_t\ seq}(\mathtt{S}(\text{“}\backslash\mathtt{n}\text{”}), \lambda\mathtt{x}.\mathtt{false}, \mathtt{bottom})$$

In the example, we define type constructor `S` to encode literals with a set-type. We also use the following informal translations: **`Pwhere`** becomes a set-type, **`Pstruct`** a series of dependent products, **`Punion`** a series of sums, and **`Parray`** a sequence. For this example, we assume that **`Peor`** is a newline, and therefore specify the sequence separator as such. As the array terminates at the end of the file, we use λ`x.false` and `bottom` to indicate the absence of termination condition and terminator, respectively.

4. DDC Semantics

The primitives of DDC are deceptively simple. Each captures a simple concept, often familiar from type theory. However, in reality, each primitive is multi-faceted. Each 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 meta-data. We give semantics to DDC types using three 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, detailing the types of the data's in-memory representation and parse descriptor. The function $[\![\cdot]\!]$ describes the *parsing semantics* of DDC, 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.

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

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

Figure 10. Representation Types

We begin with a kinding judgment that checks if a type is well formed. We then formalize the three-fold semantics of DDC types.

4.1 DDC Kinding

The kinding judgment defined in Figure 9 determines well-formed DDC types, assigning kind T to basic types and kind $\sigma \to \kappa$ to type abstractions. We use two contexts to express our kinding judgment:

$$\begin{array}{lcl} \Gamma & ::= & \cdot \mid \Gamma, x{:}\sigma \\ \mathrm{M} & ::= & \cdot \mid \mathrm{M}, \alpha{=}\mu\alpha.\tau \end{array}$$

Context Γ is a finite partial map that binds expression variables to their types. Context M is an ordered list of mappings between type variables and recursive types. This context serves two purposes: first, to ensure the well-formedness of types with free type variables; and second, to provide mappings between recursive type variables and their associated types. This second purpose leads us to consider a context M to be a substitution from type variables to types. Application of such a substitution has the form $\mathrm{M}(\tau)$.

To ensure that recursive types are well-formed, we cannot allow types such as $\mu\alpha.\alpha$. More generally, we need to ensure that recursive type variables are separated from their binder by at least one basic primitive, such as a product or sum, a condition called *contractiveness*. To this end, we annotate every judgment with a contractiveness indicator, one of y, n, or c. A y indicates the type is contractive, an n indicates it is not, and a c indicates it may be either. We consider $n < y$.

As the rules are otherwise mostly straightforward, we highlight just two of them. We use the function $\mathcal{B}_{\text{kind}}$ to assign kinds to base types. While their kind does not differentiate them from type abstractions, base types are not well formed when not applied. Once applied, all base types have kind T. The product rule 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. Note that we apply M to the type of the first component before translation, thereby closing it, as open DDC types do not translate into well-formed host types.

$\boxed{\mathrm{M};\Gamma \vdash_c \tau : \kappa}$

$$\frac{\vdash \Gamma\ \mathsf{ok}}{\mathrm{M};\Gamma \vdash_y \texttt{unit} : \mathsf{T}}\ \text{Unit} \qquad \frac{\vdash \Gamma\ \mathsf{ok}}{\mathrm{M};\Gamma \vdash_y \texttt{bottom} : \mathsf{T}}\ \text{Bottom} \qquad \frac{\Gamma \vdash e : \sigma \quad (\mathcal{B}_{\text{kind}}(C) = \sigma \to \mathsf{T})}{\mathrm{M};\Gamma \vdash_y C(e) : \mathsf{T}}\ \text{Const}$$

$$\frac{\mathrm{M};\Gamma, x{:}\sigma \vdash_c \tau : \kappa}{\mathrm{M};\Gamma \vdash_c \lambda x.\tau : \sigma \to \kappa}\ \text{Abs} \qquad \frac{\mathrm{M};\Gamma \vdash_c \tau : \sigma \to \kappa \quad \Gamma \vdash e : \sigma}{\mathrm{M};\Gamma \vdash_c \tau\, e : \kappa}\ \text{App} \qquad \frac{\mathrm{M};\Gamma \vdash_c \tau : \mathsf{T} \quad \mathrm{M};\Gamma, x{:}[\![\mathrm{M}(\tau)]\!]_{\text{rep}} * [\![\mathrm{M}(\tau)]\!]_{\text{PD}} \vdash_{c'} \tau' : \mathsf{T}}{\mathrm{M};\Gamma \vdash_y \Sigma\, x{:}\tau.\tau' : \mathsf{T}}\ \text{Prod}$$

$$\frac{\mathrm{M};\Gamma \vdash_c \tau : \mathsf{T} \quad \mathrm{M};\Gamma \vdash_{c'} \tau' : \mathsf{T}}{\mathrm{M};\Gamma \vdash_y \tau + \tau' : \mathsf{T}}\ \text{Sum} \qquad \frac{\mathrm{M};\Gamma \vdash_c \tau : \mathsf{T} \quad \mathrm{M};\Gamma \vdash_{c'} \tau' : \mathsf{T}}{\mathrm{M};\Gamma \vdash_y \tau \,\&\, \tau' : \mathsf{T}}\ \text{Intersection} \qquad \frac{\mathrm{M};\Gamma \vdash_c \tau : \mathsf{T} \quad \Gamma, x{:}[\![\mathrm{M}(\tau)]\!]_{\text{rep}} * [\![\mathrm{M}(\tau)]\!]_{\text{PD}} \vdash e : \texttt{bool}}{\mathrm{M};\Gamma \vdash_y \{x{:}\tau \mid e\} : \mathsf{T}}\ \text{Set}$$

$$\frac{\begin{array}{c}\mathrm{M};\Gamma \vdash_c \tau : \mathsf{T} \qquad \mathrm{M};\Gamma \vdash_{c_s} \tau_s : \mathsf{T} \qquad \mathrm{M};\Gamma \vdash_{c_t} \tau_t : \mathsf{T} \\ \Gamma \vdash e : [\![\tau_m]\!]_{\text{rep}} * [\![\tau_m]\!]_{\text{PD}} \to \texttt{bool} \quad (\tau_m = \mathrm{M}(\tau\, \texttt{seq}(\tau_s, e, \tau_t)))\end{array}}{\mathrm{M};\Gamma \vdash_y \tau\, \texttt{seq}(\tau_s, e, \tau_t) : \mathsf{T}}\ \text{Seq} \qquad \frac{\vdash \Gamma\ \mathsf{ok} \quad \alpha \in \mathsf{dom}(\mathrm{M})}{\mathrm{M};\Gamma \vdash_n \alpha : \mathsf{T}}\ \text{Var} \qquad \frac{\mathrm{M}, \alpha{=}\mu\alpha.\tau;\Gamma \vdash_y \tau : \mathsf{T}}{\mathrm{M};\Gamma \vdash_y \mu\alpha.\tau : \mathsf{T}}\ \text{Rec}$$

$$\frac{\Gamma \vdash e : \sigma}{\mathrm{M};\Gamma \vdash_y \texttt{compute}(e{:}\sigma) : \mathsf{T}}\ \text{Compute} \qquad \frac{\mathrm{M};\Gamma \vdash_c \tau : \mathsf{T}}{\mathrm{M};\Gamma \vdash_y \texttt{absorb}(\tau) : \mathsf{T}}\ \text{Absorb} \qquad \frac{\mathrm{M};\Gamma \vdash_c \tau : \mathsf{T}}{\mathrm{M};\Gamma \vdash_y \texttt{scan}(\tau) : \mathsf{T}}\ \text{Scan}$$

Figure 9. DDC Kinding Rules

4.2 Representation Semantics

In Figure 10, we present the representation type of each DDC primitive. While the primitives are dependent types, the mapping to the host language erases the dependency because the host language does not have dependent types. For DDC types in which expressions appear, the translation drops the expressions to remove the dependency. With these expressions gone, variables become useless, so we drop variable bindings as well, as in product and set types. Similarly, as type abstraction and application are only relevant for dependency, we translate them according to their underlying types.

In more detail, the DDC 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. Set types produce sums, where a left branch indicates the data satisfies the constraint and the right indicates it does not. In the latter case, the parser returns the offending data rather than `none` because the error is semantic rather than syntactic. Sequences produce a host language sequence paired with its length. Recursive types generate recursive representations. Note that the host type uses the same variable name as the DDC type, and so the type corresponding to the type variable α is exactly α. 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 11, we give the parse descriptor type for each DDC 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 meta-data. For types with neither subcomponents nor special meta-data, we use `unit` as the body type.

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

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

Figure 11. Parse Descriptor Types

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 `absorb` PD type is `unit` as with its representation. We assume that just as the user does not want the represenation to be kept, so too the parse descriptor. The `scan` parse descriptor is either `unit`, in case no match was found, or records the number of bits skipped before the type was matched along with the type's corresponding parse descriptor.

4.3 Parsing Semantics of the DDC

The parsing semantics of a type τ 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. Figure 12 specifies the host language types of the parsers generated from

$\boxed{[\![\tau]\!] = e}$

```
⟦unit⟧ = λ(B, ω).(ω, R_unit(), P_unit(ω))
⟦bottom⟧ = λ(B, ω).(ω, R_bottom(), P_bottom(ω))
⟦C(e)⟧ = λ(B, ω).B_imp(C) (e) (B, ω)
⟦λx.τ⟧ = λx.⟦τ⟧
⟦τ e⟧ = ⟦τ⟧ e
⟦Σ x:τ.τ′⟧ =
  λ(B, ω).
    let (ω′, r, p) = ⟦τ⟧ (B, ω) in
    let x = (r, p) in
    let (ω″, r′, p′) = ⟦τ′⟧ (B, ω′) in
    (ω″, R_Σ(r, r′), P_Σ(p, p′))
⟦τ + τ′⟧ =
  λ(B, ω).
    let (ω′, r, p) = ⟦τ⟧ (B, ω) in
    if isOk(p) then (ω′, R_+left(r), P_+left(p))
    else let (ω′, r, p) = ⟦τ′⟧ (B, ω) in
    (ω′, R_+right(r), P_+right(p))
⟦τ & τ′⟧ =
  λ(B, ω).
    let (ω′, r, p) = ⟦τ⟧ (B, ω) in
    let (ω″, r′, p′) = ⟦τ′⟧ (B, ω) in
    (max(ω′, ω″), R_&(r, r′), P_&(p, p′))

⟦{x:τ | e}⟧ =
  λ(B, ω).
    let (ω′, r, p) = ⟦τ⟧ (B, ω) in
    let x = (r, p) in
    let c = e in
    (ω′, R_set(c, r), P_set(c, p))
⟦τ seq(τ_s, e, τ_t)⟧ =
  λ(B, ω).
    letfun isDone (ω, r, p) =
      EoF(B, ω) or e (r, p) or
      let (ω′, r′, p′) = ⟦τ_t⟧(B, ω) in
      isOk(p′)
    in
    letfun continue (ω, ω′, r, p) =
      if ω = ω′ or isDone (ω′, r, p) then (ω′, r, p)
      else let (ω_s, r_s, p_s) = ⟦τ_s⟧ (B, ω′) in
      let (ω_e, r_e, p_e) = ⟦τ⟧ (B, ω_s) in
      continue (ω, ω_e, R_seq(r, r_e), P_seq(p, p_s, p_e))
    in
    let r = R_seq_init() in
    let p = P_seq_init(ω) in
    if isDone (ω, r, p) then (ω, r, p)
    else let (ω_e, r_e, p_e) = ⟦τ⟧ (B, ω) in
    continue (ω, ω_e, R_seq(r, r_e), P_seq(p, P_unit(ω), p_e))

⟦α⟧ = f_α
⟦μα.τ⟧ =
  fun f_α (B, ω) =
    let (ω′, r, p) = ⟦τ⟧ (B, ω) in
      (ω′, r, p)
⟦compute(e:σ)⟧ =
  λ(B, ω).(ω, R_compute(e), P_compute(ω))
⟦absorb(τ)⟧ =
  λ(B, ω).
    let (ω′, r, p) = ⟦τ⟧ (B, ω) in
    (ω′, R_absorb(p), P_absorb(p))
⟦scan(τ)⟧ =
  λ(B, ω).
    letfun try i =
      let (ω′, r, p) = ⟦τ⟧ (B, ω + i) in
      if isOk(p) then
      (ω′, R_scan(r), P_scan(i, p)) else
      if i = scanMax then
      (ω, R_scan_err(), P_scan_err(ω)) else
      try (i + 1)
    in try 0
```

Figure 13. DDC Semantics

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

$$[\![\tau{:}\mathsf{T}]\!]_{\text{PT}} = \texttt{bits} * \texttt{offset} \rightarrow \texttt{offset} * [\![\tau]\!]_{\text{rep}} * [\![\tau]\!]_{\text{PD}}$$
$$[\![\tau{:}\sigma \rightarrow \kappa]\!]_{\text{PT}} = \sigma \rightarrow [\![\tau{:}\kappa]\!]_{\text{PT}}$$

Figure 12. Host Language Types for Parsing Functions

```
fun R_unit () = ()
fun P_unit ω = ((0, ok, (ω, ω)), ())
fun R_bottom () = none
fun P_bottom ω = ((1, fail, (ω, ω)), ())

fun R_Σ (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))
```

Figure 14. Selected Constructor 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. The full collection of such constructor functions appears in Appendix B.

well-kinded DDC types. Note that parameterized DDC types require their arguments before they can parse any input.

Figure 13 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 types.

For any type, there are three steps to parsing: parse the subcomponents of the type (if any), assemble the resultant representation, and tabulate meta-data based on subcomponent meta-data (if any). For the sake of clarity, we have factored the latter two steps into separate representation and PD constructor functions which we define for each type. For 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 products is $\mathtt{H}_{\Sigma}$. Selected constructors are shown in Figure 14. We have also factored out some commonly occuring code into "built-in" functions, explained as needed and defined formally in Appendix B.

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

With this background, we can now discuss selected portions of the semantics. The semantics of `unit` and `bottom` show that they do not consume any input, *i.e.*, they do not change the offset. 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 pairs 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 descrip-

tor to the variable x before parsing the second element, implicitly mapping the DDC 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.

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 mutually recursive functions `isDone` and `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`.

We translate recursive types into recursive functions with a special function name corresponding to the name of the bound type variable. Recursive type variables translate to these special names.

5. Meta-theory

One of the most difficult, and perhaps most interesting, challenges of our work on DDC was determining what properties we wanted to hold. What are the "correct" invariants of data description languages? While there are many well-known desirable invariants for programming languages, the meta-theory of data description languages has been uncharted. We present the following two properties as critical invariants of our theory. We feel that they should hold, in some form, for any data description language.

- **Parser Type Correctness**: For a DDC 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.
- **Parser Error Correlation**: For any representation and PD output by a parsing function, the errors reported in the PD will be correlated with the errors present in the representation.

In formalizing these properties, we assume that DDC base types, their kinds, representation and parse descriptor types, and parsers satisfy the properties we desire to hold of the rest of the calculus. Appendix C contains a formal statement of these assumptions.

To prove our type correctness theorem by induction, we must account for the fact that any free recursive type variables in a DDC type τ will become free function variables in $[\![\tau]\!]$. To that end, we define the function $[\![\mathrm{M}]\!]_{\mathrm{PT}}$, which maps recursive variable contexts M to typing contexts Γ:

$$[\![\cdot]\!]_{\mathrm{PT}} = \cdot$$
$$[\![\mathrm{M}, \alpha{=}\mu\alpha.\tau]\!]_{\mathrm{PT}} = [\![\mathrm{M}]\!]_{\mathrm{PT}}, \mathtt{f}_\alpha{:}[\![\mathrm{M}(\mu\alpha.\tau){:}\mathsf{T}]\!]_{\mathrm{PT}}$$

We also apply M to τ to close any open references to recursive types before determining the corresponding parser type.

Theorem 1 (Type Correctness)
If $\Gamma \vdash M$ ok *and* $M; \Gamma \vdash_c \tau : \kappa$ *then* $\Gamma, [\![M]\!]_{PT} \vdash [\![\tau]\!] : [\![M(\tau){:}\kappa]\!]_{PT}$.

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

Corollary 2 (Type Correctness of Closed Types)
If $\vdash_y \tau : \kappa$ *then* $\vdash [\![\tau]\!] : [\![\tau{:}\kappa]\!]_{PT}$.

Normalized Types	ν	::=	$\mathtt{unit} \mid \mathtt{bottom} \mid C(e) \mid \lambda x.\tau \mid \Sigma\, x{:}\tau.\tau$
		$\mid$	$\tau + \tau \mid \tau \,\&\, \tau \mid \{x{:}\tau \mid e\} \mid \tau\, \mathtt{seq}(\tau, e, \tau)$
		$\mid$	$\mathtt{compute}(e{:}\sigma) \mid \mathtt{absorb}(\tau) \mid \mathtt{scan}(\tau)$
Types	τ	::=	$\nu \mid \tau\, e \mid \alpha \mid \mu\alpha.\tau$

$$\frac{\tau \hookrightarrow \tau'}{\tau\, e \hookrightarrow \tau'\, e} \qquad \frac{e \hookrightarrow e'}{\nu\, e \hookrightarrow \nu\, e'} \qquad \frac{}{(\lambda x.\tau)\, v \hookrightarrow \tau[v/x]} \qquad \frac{}{\mu\alpha.\tau \hookrightarrow \tau[\mu\alpha.\tau/\alpha]}$$

Figure 15. DDC Normalized Syntax and Normalization Rules

We start our formalization of the error-correlation property by defining representation and PD correlation. Informally, a representation and a PD are correlated when the number of errors recorded in the PD is at least as many as the number of errors in the representation and semantic errors, *i.e.*, constraint violations, are properly reported. Formally, we define correlation with the relation $\mathrm{Corr}_\tau(r, p)$. As the encoding of errors in the representation and the meaning of error counts in the PD are dependent on the DDC type that produced them, correlation is type directed. However, some types, such as type application, are not immediately useful in determining correlation, and must be normalized. In Figure 15, we define the set of normalized types ν and give normalization rules. For clarity, we define $\mathrm{Corr}^*{}_\tau(r, p)$ as a "helper" relation that normalizes types τ before checking for correlation. The relation $\mathrm{Corr}_\tau(r, p)$, then, is defined only for normalized types of kind T. Type abstractions are excluded as they cannot directly produce representations and PDs.

Definition 3
$\mathrm{Corr}^*{}_\tau(r, p)$ *iff if* $\tau \hookrightarrow^* \nu$ *then* $\mathrm{Corr}_\nu(r, p)$.

In the following definition, we abbreviate $p.h.nerr$ as $p.nerr$. and use `pos` to denote the function which returns zero when passed zero and one otherwise.

Definition 4 (Representation and PD Correlation Relation)
$\mathrm{Corr}_\tau(r, p)$ *iff exactly one of the following is true:*

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

- $\tau = \texttt{scan}(\tau')$, $r = \texttt{inl}\ r'$, $p = (h, \texttt{inl}\ (i, p'))$, $h.nerr = \texttt{pos}(i) + \texttt{pos}(p'.nerr)$, *and* $\mathrm{Corr}^*{}_{\tau'}(r', p')$.
- $\tau = \texttt{scan}(\tau')$, $r = \texttt{inr none}$, $p = (h, \texttt{inr}\ ())$, *and* $h.nerr = 1$.

Definition 5 specifies the exact property we require of parsing functions. At base kind, we require that any representation and PD returned by the parser must be correlated. At higher kind, we require that the function preserve the property of error correlation. Hence, the definition is a simple form of logical relation. Lemma 6 states that any well-formed type of base kind is error-correlated.

Definition 5 (Error Correlation Relation)
$\mathrm{EC}(\tau : \kappa)$ *iff exactly one of the following is true:*

- $\kappa = \mathsf{T}$ *and if* $[\![\tau]\!]\ (B, \omega) \hookrightarrow^* (\omega', r, p)$ *then* $\mathrm{Corr}^*{}_\tau(r, p)$
- $\kappa = \sigma \to \kappa'$ *and if* $\vdash v : \sigma$ *then* $\mathrm{EC}(\tau\ v : \kappa')$

Lemma 6 (Error Correlation at Base Kind)
If $\vdash_y \tau : \mathsf{T}$ *and* $[\![\tau]\!]\ (B, \omega) \hookrightarrow^* (\omega', r, p)$ *then* $\mathrm{Corr}^*{}_\tau(r, p)$.

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

Theorem 7 (Error Correlation)
If $\vdash_y \tau : \kappa$ *then* $\mathrm{EC}(\tau : \kappa)$.

PROOF. By induction on the size of the kind κ. For $\kappa = \mathsf{T}$, we use Lemma 6. □

Corollary 8
If $\mathrm{Corr}^*{}_\tau(r, p)$ *and* $p.h.nerr = 0$ *then there are no syntactic or semantic errors in the representation data structure* r.

6. Encoding DDLs in DDC

We can better understand the data description languages mentioned earlier by translating their constructs into the types of DDC. We start with the translation of IPADS, which captures many of the common features of DDLs. We then discuss features of PADS, DATASCRIPT, and PACKETTYPES that are not found in IPADS.

6.1 IPADS Translation

We formalize the translation from IPADS to DDC, described informally in Section 3.3, with two judgments: $p \Downarrow \tau$ prog indicates that the IPADS program p is encoded as DDC type τ, while $t \Downarrow \tau$ does the same for IPADS types t.

As much of the translation is straightforward, we present only selected rules in Figure 16. Notice we add `bottom` as the last branch of the DDC sum when translating **`Punion`** so that the parse will fail if none of the branches match rather than returning the result of the last branch. In the translation of **`Pwhere`**, we only check the constraint if the underlying value parsed with no errors. For **`Parray`**s, we add simple error recovery by scanning for the separator type. This behaviour allows us to easily skip erroneous elements. We use the `scan` type in the same way for **`Plit`**, as literals often appear as field separators in **`Pstruct`**s. We also absorb the literal as its value is known statically. We use the function $\mathrm{Ty}(c)$ to determine the correct type for the particular literal. For example, a string literal would require a **`Pstring`** type.

6.2 Beyond IPADS

We now give semantics to three features not found in IPADS: PADS switched unions, PACKETTYPES overlays, and DATASCRIPT arrays.

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

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

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

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

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

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

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

Next, we consider the *overlay* construct found in PACKETTYPES. An overlay allows us "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." [14]. 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. It can be encoded as **`Pcompute`**`(():unit)`,

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

$$\frac{t \Downarrow \tau}{t \Downarrow \tau\ \mathrm{prog}} \qquad \frac{p[t/\alpha] \Downarrow \tau\ \mathrm{prog}}{\alpha = t;\ p \Downarrow \tau\ \mathrm{prog}} \qquad \frac{p[\texttt{Prec}\ \alpha.t/\alpha] \Downarrow \tau\ \mathrm{prog}}{\texttt{Prec}\ \alpha = t;\ p \Downarrow \tau\ \mathrm{prog}}$$

$\boxed{t \Downarrow \tau}$

$$\frac{t_i \Downarrow \tau_i}{\texttt{Punion}\{x_1{:}t_1 \ldots x_n{:}t_n\} \Downarrow \tau_1 + \cdots + \tau_n + \texttt{bottom}}$$

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

$$\frac{t \Downarrow \tau \quad t_{sep} \Downarrow \tau_s \quad t_{term} \Downarrow \tau_t \quad (f = \lambda \mathrm{x}.\texttt{false})}{t\ \texttt{Parray}(t_{sep}, t_{term}) \Downarrow \tau\ \texttt{seq}(\texttt{scan}(\tau_s), f, \tau_t)}$$

$$\frac{t \Downarrow \tau}{\texttt{Popt}\ t \Downarrow \tau + \texttt{unit}} \qquad \frac{\mathrm{Ty}(c) = \tau}{\texttt{Plit}\ c \Downarrow \texttt{scan}(\texttt{absorb}(\{x{:}\tau \mid x = c\}))}$$

Figure 16. Selected Rules for Encoding IPADS in DDC. The full collection of rules appears in Appendix D.

as this type consumes no data and produces a unit value without errors. The overlay creates a new type `IPinFP` where the body field is an `IPPacket` rather than a simple byte array.

We have developed a translation of the overlay syntax into DDC (not shown due to space constraints). Although overlays are conceptually intuitive, we discovered a critical subtlety, not mentioned by the authors, when formalizing their semantics. Any expressions in the original type that refer to the overlayed field may no longer be well typed after applying the overlay. It is therefore necessary for the translation to check the new type for well formedness after the overlay process, which is an easy task in the DDC framework.

Finally, we introduce DATASCRIPT-style arrays for binary data, $t\ [length]$. 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. Otherwise, the parser continues until an element constraint is violated or the input is completely consumed.

Fixed-length arrays can be encoded in a straightforward manner with DDC sequences:

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

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

Unbounded arrays 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 (abbreviated $\mathtt{not}(\tau)$) which succeeds where τ fails and fails where τ succeeds:

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

Unbounded arrays with element type τ can now be encoded as sequences with terminator $\mathtt{not}(\tau)$.

While there are many more features that we can encode, space prevents us from detailing them here. To give a sense of what is possible, we briefly list those features of DATASCRIPT and PACKETTYPES for which we have found encodings in DDC:

- PACKETTYPES: arrays, where clauses, structures, overlays, and alternation.
- DATASCRIPT: set types (enumerations and bitmask sets), arrays, constraints, value-parameterized types (which they refer to as "type parameters"), and (monotonically increasing) labels.

We know of a couple of features from data description languages that we cannot implement in DDC as it stands. An example is a label construct that permits the user to rewind the input. However, we do not view such limitations as particularly troublesome. Like the lambda calculus or pi calculus, DDC is intended to capture many common language features, while providing a convenient and effective basis for extension with new features.

7. Applications of the Semantics

The development of DDC and defining a semantics for IPADS has had a substantial impact on the real PADS implementation.

7.1 Bug Hunting

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

The semantics also helped us avoid potential non-termination of array parsers. In the original implementation of PADS arrays, it was possible to write non-terminating arrays, a bug that was only uncovered when it hung a real program. We have fixed the bug and used the semantics to verify our fix. [1]

7.2 Principled Implementation Extension: Recursion

Unlike the rest of PADS, the semantics of recursive types preceded the implementation. We used the semantics to guide our design decisions in the implementation, particularly in preventing the user from writing down non-contractive types and in implementing the parsers with recursive functions.

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. 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, 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 allowed us to encode the semantics of other PADS features (*e.g.*, literals) elegantly.

Another accident in the PADS implementation is that there is no guarantee that certain features are "safe." This is due on occasion to the fact that the PADS host language is C and on occasion to the desire to implement certain optimizations. As an example, when a semantic error in a **`Pwhere`** clause is detected, the parser sets a flag. However, the C programmer is not forced to check this flag before using the value in question and therefore can unknowingly process invalid data. The semantics of DDC deviates from the C implementation here as it suggests constrained types be implemented as values with a sum type. A typed lambda calculus programmer is required to perform a case on the sum and hence will always be informed of an error. In such cases, the C implementation does not serve as a proper guide for the integration of PADS ideas with a safe language like ML. For this purpose, the DDC is a much more appropriate starting point.

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 panic, 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

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

now seems clear we would like a more robust **Punion**, but we are not currently sure how to design one.

8. Related Work

To our knowledge, we are the first to attempt to specify a semantics for data description languages based on types such as PACKETTYPES, DATASCRIPT or PADS.

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

Parsing Expression Grammars (PEGs), studied in the early 70s [2] and revitalized more recently by Ford [8], evolved from context-free grammars but have deterministic, prioritized choice like DDC as opposed to nondeterministic choice. Though PEGs have syntactic lookahead operators, they may be parsed in linear time through the use of "packrat parsing" techniques [7, 9]. Once again, the dual interpretation of types in DDC both as data descriptions and as classifiers for internal representations make our theory substantially different from the theory of PEGs.

ANTLR [16], a popular programming language parsing tool, uses top-down recursive descent parsing and appears roughly similar in recognition power to PEGs and DDC. ANTLR also allows programmers to place annotations in the grammar definitions to guide construction of an abstract syntax 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 DDC.

9. Conclusion

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

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

We have already used the semantics to identify bugs in the implementation of PADS and to highlight areas where PADS sacrifices safety for speed. In addition, when various biological data sources motivated adding recursion to PADS, we used DDC for design guidance. After adding recursion, PADS can now describe the biological data sources. Finally DDC has provided insight into how to design a safe overlay concept.

Acknowledgments

Thanks to Andrew Appel for suggesting we refer to Landin's seminal paper on the next 700 programming languages.

References

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

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

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

[4] D. Eger. Bit level types. `http://www-2.cs.cmu.edu/~eger/`.

[5] M. F. Fernández, J. Siméon, B. Choi, A. Marian, and G. Sur. Implementing XQuery 1.0: The Galax experience. In *VLDB*, pages 1077–1080. ACM Press, 2003.

[6] K. Fisher and R. Gruber. Pads: A domain specific language for processing ad hoc data. In *PLDI*, pages 295–304. ACM Press, June 2005.

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

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

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

[10] P. Gustafsson and K. Sagonas. Adaptive pattern matching on binary data. In *European Symposium on Programming*, pages 124–139. Springer, Mar. 2004.

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

[12] A. Igarashi, B. Pierce, and P. Wadler. Featherwieght java: a minimal core calculus for java and gj. In *ACM Conference on Object-oriented Programming, Systems, Languages, and Applications*, pages 132–146. ACM Press, 1999.

[13] B. Krishnamurthy and J. Rexford. *Web Protocols and Practice*. Addison Wesley, 2001.

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

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

[16] T. J. Parr and R. W. Quong. ANTLR: A predicated- *ll(k)* parser generator. *Software – practice and experience*, 25(7):789–810, July 1995.

[17] C. Wikström and T. Rogvall. Protocol programming in Erlang using binaries. In *Fifth International Erlang/OTP User Conference*, Oct. 1999.

A. Host Language

A.1 Well-Formedness Rules

$\boxed{\Delta \vdash \sigma\ \mathsf{ok}}$

$$\frac{}{\Delta \vdash a\ \mathsf{ok}} \quad \frac{\Delta \vdash \sigma\ \mathsf{ok} \quad \Delta \vdash \sigma'\ \mathsf{ok}}{\Delta \vdash \sigma \to \sigma'\ \mathsf{ok}} \quad \frac{\Delta \vdash \sigma\ \mathsf{ok} \quad \Delta \vdash \sigma'\ \mathsf{ok}}{\Delta \vdash \sigma * \sigma'\ \mathsf{ok}}$$

$$\frac{\Delta \vdash \sigma\ \mathsf{ok} \quad \Delta \vdash \sigma'\ \mathsf{ok}}{\Delta \vdash \sigma + \sigma'\ \mathsf{ok}} \quad \frac{\Delta \vdash \sigma\ \mathsf{ok}}{\Delta \vdash \sigma\ \mathtt{seq}\ \mathsf{ok}} \quad \frac{\Delta \vdash \sigma\ \mathsf{ok}}{\Delta \vdash \sigma\ \mathtt{error}\ \mathsf{ok}}$$

$$\frac{\Delta, \alpha \vdash \sigma\ \mathsf{ok}}{\Delta \vdash \forall \alpha.\sigma\ \mathsf{ok}} \quad \frac{\alpha \in \Delta}{\Delta \vdash \alpha\ \mathsf{ok}} \quad \frac{\Delta, \alpha \vdash \sigma\ \mathsf{ok}}{\Delta \vdash \mu\alpha.\sigma\ \mathsf{ok}}$$

$\boxed{\Delta \vdash \Gamma\ \mathsf{ok}}$

$$\frac{}{\Delta \vdash \cdot\ \mathsf{ok}} \quad \frac{\Delta \vdash \Gamma\ \mathsf{ok} \quad x \notin \mathsf{dom}(\Gamma) \quad \Delta \vdash \sigma\ \mathsf{ok}}{\Delta \vdash \Gamma, x : \sigma\ \mathsf{ok}}$$

$\boxed{\Gamma \vdash \mathrm{M}\ \mathsf{ok}}$

$$\frac{}{\Gamma \vdash \cdot\ \mathsf{ok}} \quad \frac{\Gamma \vdash \mathrm{M}\ \mathsf{ok} \quad \alpha \notin \mathsf{dom}(\mathrm{M}) \quad \mathrm{M}; \Gamma \vdash_y \mu\alpha.\tau : \mathsf{T}}{\Gamma \vdash \mathrm{M}, \alpha = \mu\alpha.\tau\ \mathsf{ok}}$$

A.2 Typing Rules

Constants are assigned types with the interface $\mathcal{B}_{\text{cty}}$ and operators with $\mathcal{B}_{\text{opty}}$. Some example constants and their types are show below.

$$\mathcal{B}_{\text{cty}}(\mathtt{true}) = \mathtt{bool} \quad \mathcal{B}_{\text{cty}}(\mathtt{false}) = \mathtt{bool}$$
$$\mathcal{B}_{\text{cty}}(\mathtt{none}) = \mathtt{none} \quad \mathcal{B}_{\text{cty}}(B) = \mathtt{bits}$$
$$\mathcal{B}_{\text{cty}}(\omega) = \mathtt{offset}$$

We use contexts Δ to record the names of open type variables and contexts Γ to record the types of expression variables. The syntax of Δ and Γ is as follows:

$$\Delta \ ::= \ . \mid \Gamma, \alpha$$
$$\Gamma \ ::= \ . \mid \Gamma, x{:}\sigma$$

The typing judgment has the form $\Delta; \Gamma \vdash e : \sigma$. When the type-variable context Δ is empty, we write $\Gamma \vdash e : \sigma$ as an abbreviation.

$$\frac{\Delta \vdash \Gamma\ \mathsf{ok}}{\Delta; \Gamma \vdash c : \mathcal{B}_{\text{cty}}(c)}\ \text{Const} \quad \frac{\Delta \vdash \Gamma\ \mathsf{ok} \quad \Gamma(x) = \sigma}{\Delta; \Gamma \vdash x : \sigma}\ \text{Var}$$

$$\frac{\mathcal{B}_{\text{opty}}(op) = \sigma' \rightharpoonup \sigma \quad \Delta; \Gamma \vdash e : \sigma'}{\Delta; \Gamma \vdash op(e) : \sigma}\ \text{Op}$$

$$\frac{\Delta; \Gamma, f{:}\sigma' \to \sigma, x{:}\sigma' \vdash e : \sigma}{\Delta; \Gamma \vdash \mathtt{fun}\ f\ x = e : \sigma' \to \sigma}\ \text{Fun} \quad \frac{\Delta; \Gamma \vdash e : \sigma' \to \sigma \quad \Delta; \Gamma \vdash e' : \sigma'}{\Delta; \Gamma \vdash e\ e' : \sigma}\ \text{App}$$

$$\frac{\Delta; \Gamma \vdash e' : \sigma' \quad \Delta; \Gamma, x{:}\sigma' \vdash e : \sigma}{\Delta; \Gamma \vdash \mathtt{let}\ x = e'\ \mathtt{in}\ e : \sigma}\ \text{Let}$$

$$\frac{\Delta; \Gamma \vdash e : \mathtt{bool} \quad \Delta; \Gamma \vdash e_1 : \sigma \quad \Delta; \Gamma \vdash e_2 : \sigma}{\Delta; \Gamma \vdash \mathtt{if}\ e\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 : \sigma}\ \text{Cond}$$

$$\frac{\Delta; \Gamma \vdash e_1 : \sigma_1 \quad \Delta; \Gamma \vdash e_2 : \sigma_2}{\Delta; \Gamma \vdash (e_1, e_2) : \sigma_1 * \sigma_2}\ \text{Pair} \quad \frac{\Delta; \Gamma \vdash e : \sigma_1 * \sigma_2}{\Delta; \Gamma \vdash \pi_i\, e : \sigma_i}\ \text{Proj}$$

$$\frac{\Delta; \Gamma \vdash e : \sigma \quad \Delta \vdash \sigma'\ \mathsf{ok}}{\Delta; \Gamma \vdash \mathtt{inl}\ e : \sigma + \sigma'}\ \text{InL} \quad \frac{\Delta; \Gamma \vdash e : \sigma' \quad \Delta \vdash \sigma\ \mathsf{ok}}{\Delta; \Gamma \vdash \mathtt{inr}\ e : \sigma + \sigma'}\ \text{InR}$$

$$\frac{\Delta; \Gamma \vdash e : \sigma_l + \sigma_r \quad \Delta; \Gamma, x{:}\sigma_l \vdash e_l : \sigma \quad \Delta; \Gamma, y{:}\sigma_r \vdash e_r : \sigma}{\Delta; \Gamma \vdash \mathtt{case}\ e\ \mathtt{of}\ (\mathtt{inl}\ x \Rightarrow e_l \mid \mathtt{inr}\ y \Rightarrow e_r) : \sigma}\ \text{Case}$$

$$\frac{\Delta \vdash \Gamma\ \mathsf{ok} \quad \Delta \vdash \sigma\ \mathsf{ok}}{\Delta; \Gamma \vdash [] : \sigma\ \mathtt{seq}}\ \text{Empty}$$

$$\frac{\Delta \vdash \Gamma\ \mathsf{ok} \quad \Delta; \Gamma \vdash e_i : \sigma \quad (\text{for } i = 1 \ldots n)}{\Delta; \Gamma \vdash [e_1 \cdots e_n] : \sigma\ \mathtt{seq}}\ \text{Seq}$$

$$\frac{\Delta; \Gamma \vdash e : \sigma\ \mathtt{seq} \quad \Delta; \Gamma \vdash e' : \sigma\ \mathtt{seq}}{\Delta; \Gamma \vdash e\ @\ e' : \sigma\ \mathtt{seq}}\ \text{Append}$$

$$\frac{\Delta; \Gamma \vdash e : \sigma\ \mathtt{seq} \quad \Delta; \Gamma \vdash e' : \mathtt{int}}{\Delta; \Gamma \vdash e\,[e'] : \sigma + \mathtt{unit}}\ \text{Sub}$$

$$\frac{\Delta; \Gamma \vdash e : \sigma[\mu\alpha.\sigma/\alpha]}{\Delta; \Gamma \vdash e : \mu\alpha.\sigma}\ \text{Roll} \quad \frac{\Delta; \Gamma \vdash e : \mu\alpha.\sigma}{\Delta; \Gamma \vdash e : \sigma[\mu\alpha.\sigma/\alpha]}\ \text{Unroll}$$

$$\frac{\Delta, \alpha; \Gamma \vdash v : \sigma \quad (\alpha \notin \mathrm{FTV}(\Gamma))}{\Delta; \Gamma \vdash v : \forall\alpha.\sigma}\ \text{Generalize}$$

$$\frac{\Delta; \Gamma \vdash e : \forall\alpha.\sigma}{\Delta; \Gamma \vdash e : \sigma[\sigma'/\alpha]}\ \text{Instantiate}$$

A.3 Evaluation Rules

$$
\begin{array}{llll}
\text{Evaluation} & E & ::= & [] \mid op(E) \mid E\ e \mid v\ E \\
\text{Contexts} & & \mid & \mathtt{let}\ x = E\ \mathtt{in}\ e \\
 & & \mid & \mathtt{if}\ E\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 \\
 & & \mid & (E, e) \mid (v, E) \mid \pi_i\, E \\
 & & \mid & \mathtt{inl}\ E \mid \mathtt{inr}\ E \\
 & & \mid & \mathtt{case}\ E\ \mathtt{of}\ (\mathtt{inl}\ x \Rightarrow e \mid \mathtt{inr}\ x \Rightarrow e') \\
 & & \mid & [\vec{v}\ E\ \vec{e}] \mid E\ @\ e \mid v\ @\ E \\
 & & \mid & e\,[\mathrm{E}] \mid v\,[\mathrm{E}]
\end{array}
$$

We specify the implementation of an operator with $\mathcal{O}(op, v)$. Most of the rules are standard, although the sequence rules are new. Append appends the contents of the second array to that of the first array, while Sub extracts the element at index i, if i is within the bounds of the array. If not, the expression fails.

$$\frac{\mathcal{O}(op, v) = v'}{op(v) \hookrightarrow v'}\ \text{Op} \quad \frac{(v = \mathtt{fun}\ f\ x = e)}{v\ v' \hookrightarrow e[v/f][v'/x]}\ \text{App}$$

$$\frac{}{\mathtt{let}\ x = v\ \mathtt{in}\ e \hookrightarrow e[v/x]}\ \text{Let}$$

$$\frac{}{\mathtt{if}\ \mathtt{true}\ \mathtt{then}\ e\ \mathtt{else}\ e' \hookrightarrow e}\ \text{IfTrue}$$

$$\frac{}{\mathtt{if}\ \mathtt{false}\ \mathtt{then}\ e\ \mathtt{else}\ e' \hookrightarrow e'}\ \text{IfFalse}$$

$$\frac{}{\pi_1\,(v, v') \hookrightarrow v}\ \text{Proj1} \quad \frac{}{\pi_2\,(v, v') \hookrightarrow v'}\ \text{Proj2}$$

$$\frac{}{\mathtt{case}\ \mathtt{inl}\ v\ \mathtt{of}\ (\mathtt{inl}\ x \Rightarrow e_l \mid \mathtt{inr}\ y \Rightarrow e_r) \hookrightarrow e_l[v/x]}\ \text{CaseL}$$

$$\frac{}{\mathtt{case}\ \mathtt{inr}\ v\ \mathtt{of}\ (\mathtt{inl}\ x \Rightarrow e_l \mid \mathtt{inr}\ y \Rightarrow e_r) \hookrightarrow e_r[v/y]}\ \text{CaseR}$$

$$\frac{}{[\vec{v}_1]\ @\ [\vec{v}_2] \hookrightarrow [\vec{v}_1\ \vec{v}_2]}\ \text{Append} \quad \frac{}{[]\,[i] \hookrightarrow \mathtt{inr}\ ()}\ \text{EmptySub}$$

$$\frac{0 \leq i < n}{[v_0 \ldots v_{n-1}]\,[i] \hookrightarrow \texttt{inl}\ v_i}\ \text{SubIn} \qquad \frac{i \geq n}{[v_0 \ldots v_{n-1}]\,[i] \hookrightarrow \texttt{inr}\ ()}\ \text{SubOut}$$

$$\frac{e \hookrightarrow e'}{E[e] \hookrightarrow E[e']}\ \text{Step}$$

B. Helper Functions

In defining the parsing functions, we use the following helper functions:

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

We define for each DDC type a pair of constructor functions, one to build a representation and another to build a parse descriptor. 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. Array bodies have type `int*int*`(σ `seq`) (for element type σ). We refer to the projections as `neerr`, `length` and `elts`, respectively.

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

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

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

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

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

fun R_set (c, r) = if c then inl r else inr r
fun P_set (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 ())
```

C. Conditions on Base Types

Condition 9 (Conditions on Base-type Interfaces)

1. $\mathsf{dom}(\mathcal{B}_{kind}) = \mathsf{dom}(\mathcal{B}_{imp})$.
2. *If* $\mathcal{B}_{kind}(C) = \sigma \rightarrow \mathsf{T}$ *then* $\mathcal{B}_{opty}(C) = \sigma \rightarrow [\![C(e){:}\mathsf{T}]\!]_{PT}$ *(for any* e *of type* σ*).*
3. $\vdash \mathcal{B}_{imp}(C) : \mathcal{B}_{opty}(C)$.
4. *If* $\vdash v : \sigma$, $\mathcal{B}_{kind}(C) = \sigma \rightarrow \mathsf{T}$ *and* $\mathcal{B}_{imp}(C)\ v\ (B, \omega) \hookrightarrow^* (\omega', r, p)$ *then* $\mathrm{Corr}_{C(v)}(r, p)$*.*

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

D. Complete IPADS Encoding in DDC

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

$$\frac{t \Downarrow \tau}{t \Downarrow \tau \text{ prog}} \qquad \frac{p[t/\alpha] \Downarrow \tau \text{ prog}}{\alpha = t;\ p \Downarrow \tau \text{ prog}} \qquad \frac{p[\mathtt{Prec}\ \alpha.t/\alpha] \Downarrow \tau \text{ prog}}{\mathtt{Prec}\ \alpha = t;\ p \Downarrow \tau \text{ prog}}$$

$\boxed{t \Downarrow \tau}$

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

$$\frac{t_i \Downarrow \tau_i}{\mathtt{Pstruct}\{x_1{:}t_1 \ldots x_n{:}t_n\} \Downarrow \Sigma\, x_1{:}\tau_1.\cdots\Sigma\, x_{n-1}{:}\tau_{n-1}.\tau_n} \qquad \frac{t_i \Downarrow \tau_i}{\mathtt{Punion}\{x_1{:}t_1 \ldots x_n{:}t_n\} \Downarrow \tau_1 + \cdots + \tau_n + \mathtt{bottom}}$$

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

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

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

$$\frac{}{\mathbf{Pcompute}\ e{:}\sigma \Downarrow \texttt{compute}(e{:}\sigma)} \qquad \frac{\mathrm{Ty}(c) = \tau}{\mathbf{Plit}\ c \Downarrow \texttt{scan}(\texttt{absorb}(\{x{:}\tau \mid x = c\}))}$$

$$\frac{}{\alpha \Downarrow \alpha} \qquad \frac{t \Downarrow \tau}{\mathbf{Prec}\ \alpha.t \Downarrow \mu\alpha.\tau}$$