The Theory and Practice of Data Description

Yitzhak H. Mandelbaum

A Dissertation

Presented to the Faculty

of Princeton University

in Candidacy for the Degree

of Doctor of Philosophy

Recommended for Acceptance

By the Department of

Computer Science

September, 2006

Abstract

Massive amounts of useful data are stored and processed in ad hoc formats for which critical tools like parsers and formatters do not exist. Ad hoc data formats are often poorly documented, and the data itself can be very large scale with a significant number of errors, like missing or malformed data and out-of-range values. Traditional databases and XML systems provide rich infrastructure for processing well-behaved data, but are of little help when dealing with data in ad hoc formats.

In this thesis, we discuss our attempts to address the challenges of ad hoc data. We explain the design and implementation of PADS/ML, a new language and system that facilitates generation of data processing tools for ad hoc formats. The PADS/ML design includes features such as dependent, polymorphic and recursive datatypes, which allow programmers to describe the syntax and semantics of ad hoc data in a concise, easy-to-read notation. The PADS/ML implementation compiles these descriptions into ML structures and functors that include types for parsed data, functions for parsing and printing, and auxiliary support for user-specified, format-dependent and format-independent tool generation.

In addition, we present a general theory of data description languages like PADS/C, PADS/ML, DATASCRIPT, and PACKETTYPES. In the spirit of Landin, we present a calculus of dependent types to serve as the semantic foundation for this family of languages. 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. Finally, we discuss how the semantics has impacted the PADS/C and PADS/ML implementations.

Acknowledgments

I would like to thank my advisor, David Walker, for his advice, encouragment and support, and for generally putting up with me, from my first year in graduate school until my last. I also owe Dave a special thanks for his apparently limitless flexibility and understanding of my needs as a parent of young children.

Next, I would like to thank Kathleen Fisher, my "second" advisor. From the first summer that I worked with her at AT&T Labs through the end of my work on my thesis, Kathleen has devoted countless hours to helping me in all aspects of my work. Despite her incredibly busy schedule, she has consistently found time to discuss my questions or concerns, whether about research, grad school, or job hunting. I would also like to thank Kathleen and Andrew Appel for reading my thesis and giving me excellent feedback on both content and style. I owe an additional thanks to Andrew for introducing me to the field of Programming Languages.

I would like to thank Melissa Lawson, our incredible Graduate Coordinator, for all of her help with anything and everything administrative over my five years in graduate school.

I thank my wife, Rachel, for her constant support, patience, and love. And for showing me what it *really* means to work hard. Along with her, I want to thank my wonderful children, Rivka and Batya, for constantly reminding me that there are things in life that are far more important than one's work. I owe a special thank you to my parents, Richard and Paulette Mandelbaum, for helping me in every way, from as early as I can remember until today. Any attempt on my part to list everything that I owe them, or even a fraction of it, would only lessen the significance of their contributions. I would also like to thank my parents-in-law, Gerald and Sunny Katz, for their constant support and generous help with watching, playing with, and caring for our children when we needed them.

Finally, I would like to thank the numerous people who helped me along the way, in ways both big and small: Robert Gruber and Mary Fernández, of AT&T Labs; my fellow Programming-Languages graduate students Jay Ligatti, Limin Jia, Sudhakar Govindavajhala, Dan Dantas, and Frances Spalding; Derek Dreyer, of Toyota Technological Institute at Chicago;and professors Kevin Wayne, Vivek Pai, Kai Li, and J.P. Singh.

This material is based upon work supported by National Science Foundation grants numbers CCF-0238328 and IIS-0612147, and a Sloan Fellowship. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation or the Sloan Foundation.

Contents

Chapter 1

Introduction

1.1 The Challenge of Ad hoc Data

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, mainstream programming languages typically have 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 an ideal world, all data would be in such formats. In reality, however, we are not nearly so fortunate.

Vast amounts of data are maintained in *ad hoc data formats* – nonstandard data formats that lack readily-available tools for common data processing tasks, such as parsing, querying, analysis, or transformation. Every day, network administrators, financial analysts, computer scientists, biologists, chemists, and physicists deal with ad hoc data in a myriad of complex formats. 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.

At AT&T alone, analysts work with numerous data sources, including call detail data [CFP+04], web server logs [KR01], netflows capturing Internet traffic [Net], log files characterizing Internet back-

Name: Use	Representation	Processing Problems
Gene Ontology (GO) [Con]: Gene Product Information	Variable-width ASCII records	White-space ambiguities
SDSS/Reglens Data [MHS+05]: Weak gravitational lensing analysis	Floating point numbers, among others	Repeated multiplicative error
Web server logs (CLF): Measuring web workloads	Fixed-column ASCII records	Race conditions on log entry Unexpected values
AT&T provisioning data (Sirius): Monitoring service activation	Variable-width ASCII records	Unexpected values Corrupted data feeds
AT&T call detail data: Phone call fraud detection	Fixed-width binary records	Undocumented data
AT&T billing data: Monitoring billing process	Cobol	Unexpected values Corrupted data feeds
IP backbone data (Regulus) Monitoring network performance	ASCII	Multiple missing-value representations. Undocumented data
Netflow Monitoring network performance	Data-dependent number of fixed-width binary records	Missed packets
Newick Standard: Immune system response simulation	Fixed-width ASCII records in tree-shaped hierarchy	No known problems
OPRA: Options-market transactions	Mixed binary & ASCII records with data-dependent unions	100-page informal documentation
Palm PDA: Device synchronization	Mixed binary & character with data-dependent constraints	No high-level documentation available

Figure 1.1: Selected ad hoc data sources.

bone resource utilization, amd wire formats for legacy telecommunication billing systems. Biologists manipulate their own data formats, including phylogenies [Newa] (evolutionary trees describing the ancestor/descendent relationships between organisms) and gene ontologies [Con] (shared vocabularies for attributes of genes and gene products). In the financial community, the Options Price Reporting Authority (OPRA) provides financial institutions with last sale information (information about options sales) and current options quotations (up-to-date option price listings) [Aut05].

Figure 1.1 summarizes some of the salient characteristics of these sources and others to give a partial sense of the range and pervasiveness of ad hoc data. It describes ad hoc data formats from several different domains ranging from genomics to cosmology to networking to finance to internal corporate billing information. They include ASCII, binary, and Cobol data formats, with both fixed and variable-width records, ranging in size from relatively small files through network applications which process over a gigabyte per second. Figures 1.2, 1.3, 1.4 and 1.5 provide data fragments from a number of these data sources to provide the user with concrete examples of ad hoc data.

```
format-version: 1.0
date: 11:11:2005 14:24
auto-generated-by: DAG-Edit 1.419 rev 3
default-namespace: gene_ontology
subsetdef: goslim_goa "GOA and proteome slim"

[Term]
id: GO:0000001
name: mitochondrion inheritance
namespace: biological_process
def: "The distribution of mitochondria\, including the mitochondrial
genome\, into daughter cells after mitosis or meiosis\, mediated by
interactions between mitochondria and the cytoskeleton."
[PMID:10873824,PMID:11389764, SGD:mcc]
is_a: GO:0048308 ! organelle inheritance
is_a: GO:0048311 ! mitochondrion distribution
```

Figure 1.2: Ad hoc data in biology. Shown here is a fragment of the Gene Ontology [Con] encoded in the OBO format, including the file header and the first entry in the ontology. The Gene Ontology describes gene products and links genes known to be related.

```
HA00000000START OF TEST CYCLE
aA00000001BXYZ U1AB0000040000100B0000004200
HL00000002START OF OPEN INTEREST
d 00000003FZYX G1AB0000030000300000
HM00000004END OF OPEN INTEREST
HE00000005START OF SUMMARY
f 00000006NYZX B1QB000520001200000700 00B000050000000520000
  00490000005100+00000100B000000053000000525000000535000
HF00000007END OF SUMMARY
k 00000008LYXW B1KB0000065G000000990010000000 1000020000
HB00000009END OF TEST CYCLE
```

Figure 1.3: Ad hoc data in finance. The Options Price Reporting Authority (OPRA) provides last sale information and current options quotations to customers in its own proprietary format. Here we provide an example data fragment – adapted from the OPRA format's manual [Aut05] – describing a simple test transaction.

```
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/dd@grp.org/confirm HTTP/1.0" 200 941
234.200.68.71 - - [15/Oct/1997:18:53:33 -0700]
   "GET /tr/img/gift.gif HTTP/1.0 200 409
240.142.174.15 - - [15/Oct/1997:18:39:25 -0700]
   "GET /tr/img/wool.gif HTTP/1.0" 404 178
188.168.121.58 - - [16/Oct/1997:12:59:35 -0700]
   "GET / HTTP/1.0" 200 3082
214.201.210.19 ekf - [17/Oct/1997:10:08:23 -0700]
   "GET /img/new.gif HTTP/1.0" 304 -
```

Figure 1.4: Ad hoc data from web server logs. Shown here is a six-entry fragment of a web server log, encoded in the Common Log Format. Each entry has been broken into two lines for readability.

```
00000000: 9192 d8fb 8480 0001 05d8 0000 0000 0872  ...............r
00000010: 6573 6561 7263 6803 6174 7403 636f 6d00  esearch.att.com.
00000020: 00fc 0001 c00c 0006 0001 0000 0e10 0027  ...............'
00000030: 036e 7331 c00c 0a68 6f73 746d 6173 7465  .ns1...hostmaste
00000040: 72c0 0c77 64e5 4900 000e 1000 0003 8400  r..wd.I.........
00000050: 36ee 8000 000e 10c0 0c00 0f00 0100 000e  6...............
00000060: 1000 0a00 0a05 6c69 6e75 78c0 0cc0 0c00  ......linux.....
00000070: 0f00 0100 000e 1000 0c00 0a07 6d61 696c  ............mail
00000080: 6d61 6ec0 0cc0 0c00 0100 0100 000e 1000  man.............
00000090: 0487 cf1a 16c0 0c00 0200 0100 000e 1000  ................
000000a0: 0603 6e73 30c0 0cc0 0c00 0200 0100 000e  ..ns0...........
000000b0: 1000 02c0 2e03 5f67 63c0 0c00 2100 0100  ......_gc...!...
000000c0: 0002 5800 1d00 0000 640c c404 7068 7973  ..X.....d...phys
000000d0: 0872 6573 6561 7263 6803 6174 7403 636f  .research.att.co
```

Figure 1.5: Ad hoc data in computer networking. Shown here is a fragment of a DNS packet, as displayed by the `hexdump` program. The left-most column provides byte numbers (in hexadecimal), the center eight columns display the contents in hexidecimal, and the remaining columns display the same content in ASCII, using a '.' when the corresponding byte is unprintable.

Ad hoc data poses a number of challenges to its users. 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. First, documentation for the format may not exist, or it may be out of date. For example, a common phenomenon is for a field in a data source to fall into disuse. After a while, a new piece of information becomes interesting, but compatibility issues prevent data suppliers from modifying the shape of their data, so instead they hijack the unused field, often failing to update the documentation to reflect the change.

Second, such data frequently contain errors, for a variety of reasons: malfunctioning equipment, programming errors, nonstandard 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 valid 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 languages like Perl. Unfortunately, this process is slow, tedious, and often unreliable. Error checking and recovery in these scripts is frequently minimal or nonexistent because, when present, such error-handling 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 can end up intertwined with parsing code, making it difficult to reuse the parsing code for different analyses. In general, while makeshift programs suffice for short-term use, their benefits come at a high cost. Yet, the cost in time and effort of systematically developing analysis software is well beyond what most analysts can afford.

1.2 PADS/C and PADS/ML

We have designed the PADS/ML language to address the challenges of ad hoc data. PADS/ML has evolved from prior work by Fisher and Gruber on PADS/C [1] [FG05]. Both PADS/ML and PADS/C are high-level

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

languages for declaratively describing data sources such that descriptions can be used to automatically generate a suite of tools for processing the data source. Each language supports the description of the physical format of a data source and its semantic constraints. Such constraints might specify ranges on fields in the data source, or relationships between fields. Both languages are general-purpose data description languages, not focused on any particular data encoding or application domain. They support 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.

For both PADS/C and PADS/ML, authors describe data sources declaratively using type declarations. PADS/C types are based on the types of the C programming language and the PADS/C compiler generates tools as libraries of C source code. PADS/ML types are based on the types of the ML language and the PADS/ML compiler generates tools as ML modules. PADS/ML goes beyond PADS/C in a number of ways, most notably with improved support for reusing descriptions and for extending the suite of generated tools. Both of these languages can describe a wide range of real data formats, including all of those mentioned in Table 1.1.

A key benefit of our approach is the high return-on-investment that analysts can derive from describing their data declaratively. While the suite of tools generated from a description varies between PADS/C and PADS/ML (PADS/C currently produces more tools because of its greater maturity), the core tools produced by the compilers for both languages are a parser and a printer for the associated data source. The parser maps raw data into two data structures: a canonical *representation* of the parsed data and a *parse descriptor*, a metadata object detailing properties of the corresponding data representation, including any errors that may have occured during parsing. Parse descriptors provide applications with programmatic access to this meta data. The printer inverts the process, mapping internal data structures and their corresponding parse descriptors back into raw data. In addition, both compilers can generate tools to convert the data into XML, print the data in human-readable form, check whether the data meets its semantic constraints, and summarize the data.

As an example, Figure 1.6 illustrates how PADS/C and PADS/ML parsers are generated and used. In the diagram, a data analyst 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 an in-memory

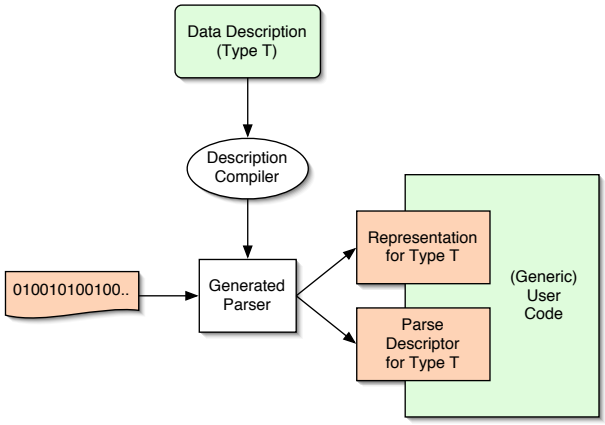

Figure 1.6: Illustration of generation and use of PADS/C and PADS/ML parsers.

representation and parse descriptor. A host-language program (user written or compiler generated) can then analyze, transform or otherwise process the data representation and parse descriptor.

The architecture of PADS/ML and PADS/C helps programmers face the challenges of ad hoc data in multiple ways. Both languages allow programmers to describe both the physical layout of data as well as its deeper semantic properties. Yet, despite this expressiveness, format specifications are easier to write than the equivalent low-level PERL script or C parser, as we will see in Chapter 2. The domain-specific constructs of the language simultaneously ease the programming burden for the user while restricting the range of expressable programs, and, hence, expressable bugs. Once written, such specifications serve as high-level documentation that is more easily read and maintained. For example, 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. Furthermore, as modifying the description is the most effective way to update the generated tools, the documentation and tools are necessarily maintained together.

PADS/C and PADS/ML address the particular challenges of error-handling in two ways. First, both compilers automatically include systematic error handling code in the generated tools. For example, 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. Yet, 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. Second, PADS/C and PADS/ML parsers return parse descriptors so that application-writers can respond to errors in application-specific ways. Based on the parse descriptors, for example, programmers can choose to fix or remove erroneous data, or even halt parsing entirely.

Another benefit of the PADS languages is their basis in type theory, which is especially helpful as ordinary programmers have built up strong intuitions about types. We have exploited these intuitions in the design of PADS/ML and PADS/C to make the syntax and semantics of descriptions particularly easy to understand, even for beginners. For instance, an array type is used to describe sequences of data objects. Similarly, union types are used to describe alternatives in the data source.

Finally, we support large data sources by generating *multiple-entry point parsers*, which provide a separate point to begin parsing for each type declaration in the description. In this way, programmers can choose the granularity at which to parse the data source. For example, a data source that consists of a sequence of records could be parsed in (at least) two ways. The programmer could use the entry point that corresponds to the description of the entire source, in which case the parser would return after parsing the entire source. Or, the programmer could parse the source one record at a time, by using the entry point that corresponds to the description of a single record. Notice that the former strategy might be appropriate for a small data source that would fit entirely into main memory, whereas the latter strategy might be appropriate for a large data source where only one record, or group of records, could fit in memory at once.

As an aside, although the syntax of everyday programming languages might be considered ad hoc, we explicitly exclude programming language syntax from our domain of interest. The challenges related to programming languages are different than those of the kind of data with which we are concerned. For example, programming languages can be (nearly always) described with context-free grammars, whereas other data sources often cannot. It is rarely, if ever, worth compiling a program with errors in it, while we quite often need to be able to process data sources with errors. The tools that we produce from descriptions, like the summary tool and the XML converter, would not be useful for processing a program. Perhaps most importantly, a great deal of work has gone into addressing the challenges of processing programming language syntax, while the challenges presented by other forms of ad hoc data have been largely neglected.

1.3 Thesis Overview

In this thesis, we will discuss the design and implementation of PADS/ML followed by a general theory of data description languages. Chapter 2 presents the PADS/ML language, discusses the process of compiling a PADS/ML description into useful tools, provides example uses of these tools, and discusses PADS/ML's tool development framework. Chapter 3 focuses on the semantics of data description languages. We define the syntax and semantics of a formal data description language, called the *Data Description Calculus* or DDC$^{\alpha}$. We designed the constructs of this language for simplicity and orthogonality, each intended to serve a single purpose. We define the semantics of PADS/C through an elaboration of the complex constructs of PADS/C into the simpler constructs of DDC$^{\alpha}$. Finally, we discuss how DDC$^{\alpha}$ can be used more broadly to define the semantics of other data description languages. We conclude with a discussion of related work for both PADS/ML and DDC$^{\alpha}$. Note that we will not discuss PADS/C in this thesis, except as it relates to our work on PADS/ML and our theory of data description languages, because the design and development of PADS/C were done largely independently of the work presented in this thesis.

The bulk of this thesis is based on two previous works. Chapter 2 is based on the technical report "PADS/ML: A Functional Data Description Language," by Yitzhak Mandelbaum, Kathleen Fisher, David Walker, Mary Fernández, and Artem Gleyzer [MFW^{+}06]. Chapter 3 is based on the published paper "The Next 700 Data Description Languages," by Kathleen Fisher, Yitzhak Mandelbaum and David Walker [FMW06]. Both of these works were edited and expanded considerably for this thesis. At an aesthetic level, we have attempted to unify the works into a coherent whole and we have expanded and revised the explanatory text in numerous places. At a technical level, Chapter 3 extends and revises the theory and metatheory presented in "The Next 700 Data Description Languages" based on new work presented in "PADS/ML: A Functional Data Description Language." The main innovation of the technical report is the ability to define functions from types to types, which are needed to model PADS/ML's polymorphic datatype. In addition, we simplify the overall semantics by making a couple of subtle technical changes. We have eliminated the complicated "contractiveness" constraint from our earlier work, and we now treat recursive type variables as abstract, rather than storing their unfoldings in the kinding context. We have also restated our "error correlation" theorem as a canonical forms lemma. Finally, this thesis contains two important contributions that are not found in either of the other works: first, we include the statements of all

essential lemmas for the meta-theory of DDC$^{\alpha}$, in addition to proofs and proof sketches, where appropriate; second, we specify the formal semantics of PADS/ML polymorphic, recursive datatypes in DDC$^{\alpha}$.

Chapter 2

PADS/ML: A Functional Data Description Language

2.1 Introduction

PADS/ML is a domain-specific language designed to improve the productivity of data analysts, be they computational biologists, physicists, network administrators, healthcare providers, financial analysts, or others. The design of the PADS/ML language was inspired by the type structure of functional programming languages. Specifically, PADS/ML provides dependent, polymorphic recursive datatypes, layered on top of a rich collection of base types, to specify the syntactic structure and semantic properties of data formats. Together, these features enable analysts to write concise, complete, and reusable descriptions of their data. We describe the PADS/ML language using examples from several domains in Section 2.2.

A key benefit of our approach is the high return-on-investment that analysts can derive from describing their data in PADS/ML. In particular, PADS/ML makes it possible to produce automatically a collection of data analysis and processing tools from each description. As a start, the PADS/ML compiler generates from each description a parser and a printer for the associated data source. The parser maps raw data into two data structures: a canonical *representation* of the parsed data and a *parse descriptor*, a metadata object detailing properties of the corresponding data representation. Parse descriptors provide applications with

programmatic access to errors detected during parsing. The printer maps data representations back into raw data, guided by the corresponding parse descriptors.

We have implemented PADS/ML by compiling descriptions into O'CAML code. We use a "types as modules" implementation strategy in which each PADS/ML type becomes a module and each PADS/ML type constructor becomes a functor. We chose ML as the host language because we believe that functional languages lend themselves to data processing tasks more readily than imperative languages such as C or JAVA. In particular, constructs such as pattern matching and higher-order functions make expressing data transformations particularly convenient. Section 2.3 describes our "types as modules" strategy and shows how PADS/ML-generated modules together with functional O'CAML code can concisely express common data-processing tasks such as error filtering and format transformation.

In addition to generating parsers and printers, our framework permits developers to add format-independent, or *generic*, tools without modifying the PADS/ML compiler. A new tool need only match a generic interface, specified as an ML signature. Correspondingly, for each PADS/ML description, the PADS/ML compiler generates a metatool (a functor) that takes a generic tool and specializes it for use with the particular description. Section 2.4 describes the tool framework and gives examples of three generic tools that we have implemented: a data printer useful for description debugging, an accumulator that keeps track of error information for each type in a data source, and a formatter that maps data into XML.

PADS/ML has evolved from previous work on PADS/C [FG05], but PADS/ML differs from PADS/C in three significant ways. First, it is targeted at the ML family of languages. Using ML as the host language simplifies the implementation of many data processing tasks, like data transformation, which benefit from ML's pattern matching and high level of abstraction. Second, unlike PADS/C types, PADS/ML types may be parameterized by other types, resulting in more concise descriptions through code reuse. ML-style datatypes and anonymous nested tuples also help improve the readability and compactness of descriptions. Third, PADS/ML provides significantly better support for the development of format-independent tools with its generic interface against which such tool can be written. In PADS/C, format-independent tools are written as code generators within the compiler, and, therefore, developing a format-independent tool requires understanding and modifying the compiler.

In summary, this chapter discusses the following contributions:

- We have designed and implemented PADS/ML, a novel data-description language that includes dependent polymorphic recursive datatypes. This design allows data analysts to express the syntactic structure and semantic properties of data formats from numerous application domains in a concise and easy-to-read notation.
- Our PADS/ML implementation employs an effective and general "types as modules" compilation strategy that produces robust parser and printer functions as well as auxiliary support for user-specified tool generation.

2.2 Describing Data in PADS/ML

A PADS/ML description specifies the physical layout and semantic properties of an ad hoc data source. These descriptions are composed of types: base types describe atomic data, while structured types describe compound data built from simpler pieces. Examples of base types include ASCII-encoded, 8-bit unsigned integers (`Puint8`) and 32-bit signed integers (`Pint32`), binary 32-bit integers (`Pbint32`), dates (`Pdate`), strings (`Pstring`), zip codes (`Pzip`), phone numbers (`Pphone`), and IP addresses (`Pip`). Semantic conditions for such base types include checking that the resulting number fits in the indicated space. For example, `Pint16` checks that any integers that it parses fit into 16-bits.

Base types may be parameterized by ML values. This mechanism reduces the number of built-in base types and permits base types to depend on values in the parsed data. For example, the base type `Puint16_FW(3)` specifies an unsigned, two-byte integer physically represented by exactly three characters, and the base type `Pstring` takes an argument indicating the character in the source that immediately follows the string, which is called the *terminator character*.

To describe more complex data, PADS/ML provides a collection of type constructors derived from the type structure of functional programming languages like Haskell and ML. We explain these structured types in the following subsections using examples drawn from data sources that we have encountered in practice. However, we do not cover every detail of PADS/ML in this section. For a complete listing of the syntax, please see Appendix B.

```
0|1005022800
9153|9153|1|0|0|0|0||152268|LOC_6|0|FRDW1|DUO|LOC_CRTE|1001476800|LOC_OS_10|1001649601
9152|9151|1|9735551212|0||9085551212|07988|no_ii152272|EDTF_6|0|APRL1|DUO|10|1000295291
```

Figure 2.1: Sirius data used to monitor billing in telecommunications industry.

2.2.1 Simple Structured Types

The bread and butter of a PADS/ML description are the simple structured types: tuples and records for specifying ordered data; lists for specifying homogeneous sequences of data; sum types for specifying alternatives; and singletons for specifying the occurrence of literal characters in the data. We describe each of these constructs as applied to the Sirius data presented in Figure 2.1.

Sirius data summarizes orders for phone service placed with AT&T. Each Sirius data file starts with a timestamp followed by one record per phone service order. Each order consists of a header and a sequence of events. The header has 13 pipe separated fields: the order number, AT&T's internal order number, the order version, four different telephone numbers associated with the order, the zip code of the order, a billing identifier, the order type, a measure of the complexity of the order, an unused field, and the source of the order data. Many of these fields are optional, in which case nothing appears between the pipe characters. The billing identifier may not be available at the time of processing, in which case the system generates a unique identifier, and prefixes this value with the string "no_ii" to indicate that the number was generated. The event sequence represents the various states a service order goes through; it is represented as a newline-terminated, pipe-separated list of state, timestamp pairs. There are over 400 distinct states that an order may go through during provisioning. The sequence is sorted in order of increasing timestamps. From this description, it is apparent that English is a poor language for describing data formats!

Figure 2.2 contains the PADS/ML description for the Sirius data format. The description is a sequence of type definitions. Type definitions precede uses, therefore the description should be read bottom up. The type `Source` describes a complete Sirius data file and denotes an ordered tuple containing a `Summary_header` value followed by an `Orders` value. Other tuple types are defined in the `Summary_header`, `Event`, and `Order` types.

The type `Orders` uses the list type constructor `Plist` to describe a homogenous sequence of values in a data source. The `Plist` constructor takes three parameters: on the left, the type of elements in the list; on the right, a literal *separator* that delimits elements in the list, and a literal *terminator*. In this example,

```
ptype Summary_header = "0|" * Ptimestamp * '\n'

pdatatype Dib_ramp =
  Ramp of Pint
| GenRamp of "no_ii" * Pint

ptype Order_header = {
     order_num : Pint;
'|'; att_order_num : [i:Pint | i < order_num];
'|'; ord_version : Pint;
'|'; service_tn : Pphone Popt;
'|'; billing_tn : Pphone Popt;
'|'; nlp_service_tn : Pphone Popt;
'|'; nlp_billing_tn : Pphone Popt;
'|'; zip_code : Pzip Popt;
'|'; ramp : Dib_ramp;
'|'; order_sort : Pstring('|');
'|'; order_details : Pint;
'|'; unused : Pstring('|');
'|'; stream : Pstring('|');
'|'
}

ptype Event  = Pstring('|') * '|' * Ptimestamp
ptype Events = Event Plist('|', '\n')

ptype Order  = Order_header * Events
ptype Orders = Order Plist('\n', peof)
ptype Source = Summary_header * Orders
```

Figure 2.2: PADS/ML description of Sirius provisioning data.

```
ptype BranchLength = ':' * Pfloat32

pdatatype Tree =
  Interior of '(' * Tree Plist(',',')')  * ')' * BranchLength
| Leaf of Pstring(':') * BranchLength
```

Figure 2.3: PADS/ML description of Newick format.

the type `Orders` is a list of `Order` elements, separated by a newline, and terminated by **`peof`**, a special literal that describes the *end-of-file marker*. Similarly, the `Events` type denotes a sequence of `Event` values separated by vertical bars and terminated by a newline.

Literal characters in type expressions denote singleton types. For example, the `Event` type is a string terminated by a vertical bar, followed by a vertical bar, followed by a timestamp. The singleton type `'|'` means that the data source must contain the vertical bar character at this point in the input stream. String, character, and integer literals can be embedded in a description and are interpreted as singleton types. For example, the singleton type `"0|"` in the `Summary_header` type denotes the "`0|`" string literal.

The type `Order_header` is a record type, which is essentially a tuple type in which each field may have an associated name. The named field `att_order_num` illustrates two other features of PADS/ML: dependencies and constraints. Here, `att_order_num` depends on the previous field, `order_num`, and is constrained to be less than that value. In practice, constraints may be complex, have multiple dependencies, and can specify, for example, the sorted order of records in a sequence. Constrained types have the form `[x:T | e]` where `e` is an arbitrary pure boolean expression[1]. Data satisfies this description if it satisfies `T` and boolean `e` evaluates to true when the parsed representation of the data is substituted for `x`. If the boolean expression evaluates to false, the data contains a *semantic* error.

The datatype definition of `Dib_ramp` specifies two alternatives for a data fragment, either one integer or the fixed string `"no_ii"` followed by one integer. The order of alternatives is significant, that is, the parser attempts to parse the first alternative and only if it fails, will it attempt to parse the second alternative. This semantics differs from similar constructs in regular expressions and context-free grammars, which nondeterministically choose between alternatives.

[1] While PADS/ML requires that the expression be pure, the implementation does not yet enforce this requirement.

```
(((erHomoC:0.28006,erCaelC:0.22089):0.40998,(erHomoA:0.32304,(erpCaelC:0.58815,((erHomoB:\
0.5807,erCaelB:0.23569):0.03586,erCaelA:0.38272):0.06516):0.03492):0.14265):0.63594,\
(TRXHomo:0.65866,TRXSacch: 0.38791):0.32147,TRXEcoli:0.57336)
```

Figure 2.4: Simplified tree-shaped data in Newick Standard format [Newb]. Each '\' indicates a newline that we inserted to improve legibility.

```
2:3004092508||5001|dns1=abc.com;dns2=xyz.com|c=slow link;w=lost packets|INTERNATIONAL
3:|3004097201|5074|dns1=bob.com;dns2=alice.com|src_addr=192.168.0.10;dst_addr=\
192.168.23.10;start_time=1234567890;end_time=1234568000;cycle_time=17412|SPECIAL
```

Figure 2.5: Simplified network-monitoring data. The '\' in the second line indicates a newline that we inserted to improve legibility.

2.2.2 Recursive Types

PADS/ML can describe data sources with recursive structure. One example of recursive data is the Newick Standard format, a flat representation of trees used by biologists [Newa] that employs properly nested parentheses to specify a tree's hierarchy. An interior tree node is represented by a matched pair of parentheses containing a (possibly empty) sequence of children nodes separated by commas. An exterior node is represented by its name. Additionally, every node is annotated with the length of the branch that separates the node from its parent.

Figure 2.3 contains a concise description of Newick Standard. Notice that branch lengths are described as floating point numbers and node names are described as strings terminated by colons. Figure 2.4 contains a small fragment of example data. In this example, the string labels are gene names and the branch length denotes the number of mutations that occur in the antibody receptor genes of B lymphocytes. Steven Kleinstein provided this data. He used similar data in his study of the proliferation of B lymphocytes during an immune response.

2.2.3 Polymorphic Types and Advanced Datatypes

The Regulus project at AT&T monitors network links, sending out alarms when problems are encountered. In Figure 2.5, we show a small fragment of example Regulus data. Each entry records a particular alarm sent out by Regulus. The data in each alarm is encoded largely in name-value pairs. For example, in Line 1 of Figure 2.5, the name-value pair "`dns1=abc.com`" tells us that the domain name of the source of the network link is "`abc.com`." Because of the prevalence of name-value pairs, we would like to describe name-value pairs with a type definition that could be used throughout the description of the Regulus data.

However, while the names in such pairs are always strings, the types of the values can differ. Also, for some pairs, a particular name is specified by the format, while for others, any name is valid. It would seem that we need to define a separate type for each different kind of name-value pair in the data source.

Polymorphic types enable analysts to define reusable descriptions by abstracting type definitions over type and value parameters. Reusable descriptions, in turn, allow for more concise descriptions, because common description elements can be written once and then referenced by name in the remainder of the description. In Figure 2.6, we show a PADS/ML description of the Regulus data format that uses polymorphic types to concisely describe the many kinds of name-value pairs that appear in Regulus data. In particular, we define the polymorphic type `Pnvp`, which abstracts over the type of the value, as a basis for all name-value pair variants. For all polymorphic type definitions, type parameters appear to the left of the type name, as is customary in ML; value parameters and their ML types appear to the right of the type name. In the definition of type `Pnvp`, there is a single type parameter named `Alpha` and no value parameters. Informally, `Pnvp` describes a name-value pair where the value has type `Alpha`.

We use `Pnvp` twice in the Regulus description. First, we define the polymorphic type `Nvp` that uses `Pnvp` to define a name-value pair whose name must match the string argument `name`, but whose value can have any type. Later in the description, the type parameter to `Nvp` is instantiated with IP addresses, timestamps, integers, and strings. Second, we reuse the type `Pnvp` in the definition of the `Generic` variant of the `Info` type. We apply `Pnvp` to the type `SVString` to describe a name-value pair whose value is a string terminated by a semicolon or vertical bar.

To appreciate the utility of polymorphic types, it is helpful to compare the PADS/ML Regulus description in Figure 2.6 with the PADS/C description in Figures 2.2.3 and 2.2.3. In particular, notice that the PADS/C description of this format must define a different type for each variant.

The Regulus description also illustrates the use of *switched* datatypes. A switched datatype selects a variant based on the value of a user-specified O'CAML expression, which typically references parsed data from earlier in the data source. For example, the switched datatype `Info` chooses a variant based on the value of its `alarm_code` parameter. More specifically, if the alarm code is `5074`, the format specification given by the `Details` constructor will be used to parse the current data. Otherwise, the format given by the `Generic` constructor will be used to parse the current data.

```
(* Pstring terminated by semicolon or vertical bar. *)
ptype SVString = Pstring_SE("/;|\\|/")

(* Generic name value pair. *)
ptype (Alpha) Pnvp = Pstring('=') * '=' * Alpha

(* Name value pair with name specified. *)
ptype (Alpha) Nvp(name:string) = [nvp: Alpha Pnvp | fst nvp = name]

ptype Details = {
        source      : Pip Nvp("src_addr");
  ';';  dest        : Pip Nvp("dest_addr");
  ';';  start_time  : Ptimestamp Nvp("start_time");
  ';';  end_time    : Ptimestamp Nvp("end_time");
  ';';  cycle_time  : Puint32 Nvp("cycle_time")
}

pdatatype Info(alarm_code : int) =
  match alarm_code with
    5074 -> Details of Details
  | _    -> Generic of (SVString Pnvp) Plist(';','|')

pdatatype Service =
    DOMESTIC      of "DOMESTIC"
  | INTERNATIONAL of "INTERNATIONAL"
  | SPECIAL       of "SPECIAL"

ptype Alarm = {
        alarm     : [a : Puint32 | a = 2 or a = 3];
  ':';  start     : Ptimestamp Popt;
  '|';  clear     : Ptimestamp Popt;
  '|';  code      : Puint32;
  '|';  src_dns   : SVString Nvp("dns1");
  ';';  dest_dns  : SVString Nvp("dns2");
  '|';  info      : Info(code);
  '|';  service   : Service
}

ptype Source = Alarm Plist('\n',peof)
```

Figure 2.6: PADS/ML description of Regulus data.

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

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

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

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

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

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

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

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

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

Penum Service {
  DOMESTIC,
  INTERNATIONAL,
  SPECIAL
};
```

Figure 2.7: PADS/C description of Regulus data, part 1.

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

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

Figure 2.8: PADS/C description of Regulus data, part 2.

The last feature of interest in the Regulus description is the use of literals as datatype branches. In the `Service` datatype, the string literals in each branch specify to parse the string literal, but omit it from the internal data representation, because the literal can be determined by the datatype constructor.

2.3 From PADS/ML to O'CAML

PADS/ML descriptions are compiled into O'CAML modules that can be used by any O'CAML program. The contents of the generated modules include a parser and printer, a functor to specialize generic tools to the given data source, and type declarations to describe the in-memory representation of the data source and its corresponding parse-descriptor. In this section, we describe the generated modules and their contents in detail, and illustrate their use.

2.3.1 Types as Modules

We use the O'CAML module system to structure the libraries generated by the PADS/ML compiler. Each PADS/ML base type is implemented as an O'CAML module. For each PADS/ML type in a description, the PADS/ML compiler generates an O'CAML module containing the types, functions, and nested modules that implement the PADS/ML type. All the generated modules are grouped into one module that implements the complete description. For example, a PADS/ML description named `sirius.pml` containing three

```
type pd_header = {
  nerr       : int;
  error_code : error_code;
  error_info : error_info;
  span       : span;
}
```

Figure 2.9: The O'CAML type of parse-descriptor headers. The types `error_code`, `error_info`, and `span` are defined in Appendix C.

named types results in the O'CAML file `sirius.ml` defining the module `Sirius`, which contains three submodules, each corresponding to one named type.

Namespace management alone is sufficient motivation to employ a "types as modules" approach, but the power of the ML module system provides substantially more. We implement polymorphic PADS/ML types as functors from (type) modules to (type) modules. Ideally, we would like to map recursive PADS/ML types into recursive modules. Unfortunately, this approach currently is not possible, because O'CAML prohibits the use of functors within recursive modules, and the output of the PADS/ML compiler includes a functor for each type. Instead, we implement recursive types as modules containing recursive datatypes and functions. As there is no theoretical reason to prevent recursive modules from containing functors [Dre05], we pose our system as a challenge to implementers of module systems.

More precisely, a module generated for a monomorphic PADS/ML type matches the signature `S`:

```
module type S = sig
  type rep
  type pd_body
  type pd = Pads.pd_header * pd_body
  val  parse : Pads.handle -> rep * pd
  val  print : rep -> pd -> Pads.handle -> unit
  (* Functor for tool generator ... *)
  module Traverse ...
end
```

The *representation* (`rep`) type describes the in-memory representation of parsed data, while the *parse-descriptor* (`pd`) type describes metadata collected during parsing. Every parse descriptor contains a header and a body. The header includes the number of subcomponents that contain errors, an error code that classifies the errors encountered during parsing (if any), an expanded description of the errors encountered, and the span – the start and end locations – of the raw data in the data source. The body contains the parse descriptors for subcomponents of the corresponding representation. Parse descriptors for base types have a body of type `unit`.

The `parse` function converts the raw data into an in-memory representation and parse descriptor for the representation. The `print` function converts a data representation into a string of data in the original, raw format. The printing is guided by a parse descriptor that corresponds to the data representation. In particular, any data marked by the parse descriptor as syntactically invalid is omitted from the generated string. Other data, though, is printed in its original form or an equivalent, depending on the particular base types included in the description. The module `Traverse` is a functor that takes a format-independent tool and specializes it to the data format described with the PADS/ML type; we defer a description of this functor to Section 2.4.

The module `Pads` contains the built-in types and functions that occur in base-type and generated modules. For example, `Pads.pd_header`, shown in Figure 2.9, is the type of all parse-descriptor headers and `Pads.handle` is an abstract type containing the private data structures PADS/ML uses to manage data sources. A complete listing of the `Pads` interface is provided in Appendix C.

The structure of the representation and parse-descriptor types resembles the structure of the corresponding PADS/ML type, making it easy to see the correspondence between parsed data, its internal representation, and corresponding metadata. For example, given the PADS/ML type of a character and integer separated by a vertical bar:

```
ptype Pair = Pchar * '|' * Pint
```

the compiler generates a module with the signature:

```
module type Pair_sig = sig
  type rep     = Pchar.rep * Pint.rep
  type pd_body = Pchar.pd  * Pint.pd
  type pd      = Pads.pd_header * pd_body
  val  parse   : Pads.handle -> rep * pd
  val  print   : rep -> pd -> Pads.handle -> unit
  ...
end
```

The signature for a polymorphic PADS/ML type uses the signature `S`, defined above. For example, given the polymorphic PADS/ML type `ABPair`:

```
ptype (Alpha,Beta) ABPair = Alpha * '|' * Beta
```

the compiler generates a module with the signature:

```
module type ABPair_sig (Alpha : S) (Beta : S) =
sig
  type rep      = Alpha.rep * Beta.rep
  type pd_body = Alpha.pd * Beta.pd
  type pd       = Pads.pd_header * pd_body
  val  parse    : Pads.handle -> rep * pd
  val  print    : rep -> pd -> Pads.handle -> unit
  ...
end
```

2.3.2 Using the Generated Libraries

Next, we present O'CAML programs that demonstrate how to use PADS/ML modules to compute properties of ad hoc data, to filter it, and to transform it.

Example: Computing Properties

Given the PADS/ML type:

```
ptype IntTriple = Pint * '|' * Pint * '|' * Pint
```

the following O'CAML expression computes the average of the three integers:

```
let ((i1,i2,i3), (pd_hdr, pd_body)) =
 Pads.parse_source IntTriple.parse "input.data"
in match pd_hdr with
     {error_code = Pads.Good} -> (i1 + i2 + i3)/3
   | _ -> raise Pads.Bad_file
```

The `parse_source` function takes a parsing function and a file name, applies the parsing function to the data in the specified file, and returns the resulting representation and parse descriptor. To ensure that the data is valid, the error code in the parse-descriptor header is examined. The error code `Good` indicates that the data is syntactically and semantically valid. Other error codes include `Nest`, indicating an error in a subcomponent, `Syn`, indicating that a syntactic error occurred during parsing, and `Sem`, indicating that the data violates a semantic constraint. The expression above raises an exception if it encounters any of these error codes.

Checking the top-level parse descriptor for errors is sufficient to guarantee that there are no errors in any of the subcomponents. This property holds for all representations and corresponding parse descriptors. This design supports a "pay-as-you-go" approach to error handling. The parse descriptor for valid data need only be consulted once, no matter the size of the corresponding data, and user code only needs to traverse the nested parse descriptors if more precise information about an error is required.

```
open Pads

let classify_order order (pd_hdr, pd_body) (good, bad)=
   match pd_hdr with
    {error_code = Good} -> (order::good, bad)
   | _                  -> (good, order::bad)

let split_orders orders (orders_pd_hdr,order_pds) =
   List.fold_right2 classify_order orders order_pds []

let ((header, orders),(header_pd, orders_pd)) =
   parse_source Sirius.parse "input.txt"

let (good,bad) = split_orders orders orders_pd
```

Figure 2.10: Error filtering of Sirius data

Example: Filtering

Data analysts often need to "clean" their data, by removing or repairing data containing errors, before loading the data into a database or other application. O'CAML's pattern matching and higher-order functions can simplify these tasks. For example, the expressions in Figure 2.10 partition Sirius data into valid orders and invalid orders.

Example: Transformation

Once a data source has been parsed and cleaned, a common task is to transform the data into formats required by other tools, like a relational database or a statistical analysis package. Transformations include removing extraneous literals, inserting delimiters, dropping or reordering fields, and normalizing the values of fields – for example, by converting all times into a specified time zone.

Because relational databases typically cannot directly store fields whose content type varies, one common transformation is to convert such fields into a form that such systems can handle. One option is to partition or "shred" the data into several relational tables, one for each variant of the field. A second option is to create a universal table, with one column for each variant of any field. If a given variant does not occur in a particular field, its value is marked as missing.

In Figure 2.11, we show a partial listing of `RegulusNormal.pml`, a normalized version of the Regulus description from Section 2.2. In this shredded version, `Alarm` has been split into two top-level types `D_alarm` and `G_alarm`. The type `D_alarm` contains all the information concerning alarms with the de-

```
...
ptype Header = {
       alarm : [ a : Puint32 | a = 2 or a = 3];
 ':';  start :  Ptimestamp Popt;
 '|';  clear :  Ptimestamp Popt;
 '|';  code: Puint32;
 '|';  src_dns  :  Nvp("dns1");
 ';';  dest_dns :  Nvp("dns2");
 '|';  service  : Service
}

ptype D_alarm = {
       header : Header;
 '|';  info   : Details
 }

ptype G_alarm = {
       header : Header;
 '|';  info   : (SVString Pnvp) Plist(';','|')
}
```

Figure 2.11: Partial Listing of `RegulusNormal.pml`, a normalized format for Regulus data. All named types not explicitly included in this figure are unchanged from the original Regulus description.

tailed payload, while `G_alarm` contains the information for generic payloads. In the original description, the `info` field identified the type of its payload. In the shredded version, the two different types of records appear in two different data files. Since neither of these formats contains a union, they can be easily loaded into a relational database.

The code fragment in Figure 2.12 shreds Regulus data in the format described by `Regulus.pml` into the format described in `RegulusNormal.pml`. It uses the `info` field of `Alarm` records to partition the data. In the process, we also reorder the fields, putting the `service` field into the common `header`. Notice that the code invokes the `print` functions generated for the `G_alarm` and `D_alarm` types to output the shredded data.

2.4 The Generic Tool Framework

An essential benefit of PADS/ML is that it can provide users with a high return-on-investment for describing their data. While the generated parser and printer alone are enough to justify the user's effort, we aim to increase the return by enabling users to easily construct data analysis tools. However, there is a limit, both in resources and expertise, to the range of tool generators that we can develop. Indeed, new and interesting

```
open Regulus
open RegulusNormal
module A  = Alarm
module DA = D_alarm
module GA = G_alarm
module Header = H

type ('a,'b) Sum = Left of 'a | Right of 'b

let splitAlarm a =
    let h =
       {H.alarm=a.A.alarm; H.start=a.A.start;
        H.clear=a.A.clear; H.code=a.A.code;
        H.src_dns=a.A.src_dns;
        H.dest_dns=a.A.dest_dns;
        H.service=a.A.service}
    in match a with
      {info=Details(d)} ->
      Left {DA.header = h; DA.info = d}
    | {info=Generic(g)} ->
      Right {GA.header = h; GA.info = g}

let process_alarm pads [pads_D; pads_G] =
  let a,a_pd = A.parse pads in
    match (split_alarm a, split_alarm_pd a_pd) with
     (Left  da, Left  da_p) -> DA.print da da_p pads_D
    |(Right ga, Right ga_p) -> GA.print ga ga_p pads_G
    | _ -> ... (* Bug! *)

let _ = process_source process_alarm
             "input.data" ["d_out.data";"g_out.data"]
```

Figure 2.12: Shredding Regulus data based on the `info` field.

data analysis tools are constantly being developed, and we have no hope of integrating even a fraction of them into the PADS/ML system ourselves. Therefore, it is essential that we provide a simple framework for others to develop tool generators.

The techniques of type-directed programming, known variously as *generic* [Hin00] or *polytypic* [JJ96] programming, provide a convenient conceptual starting point in designing a tool framework. In essence, any tool generator is a function from a description to the corresponding tool. As PADS/ML descriptions are types, a tool generator is a type-directed program.

Support for some form of generic programming over data representations and parse descriptors is an essential first step in supporting the development of tool generators. While a full-blown generic programming system like Generic Haskell [HJ03] would be useful in this context, O'CAML lacks a generic programming facility. Yet, we can still achieve some of the benefits of generic programming even without such a facility, as a number of useful data processing tools can be specified generically using only the PADS/ML compiler and the O'CAML module system.

In particular, many of the tools we have encountered perform their computations in a single pass over the representation and corresponding parse descriptor, visiting each value in the data with a pre-, post-, or in-order traversal. This paradigm arises naturally as it scales to very large data sets. It can be abstracted in a manner similar to the generic functions of Lammel and Peyton-Jones [LP03]. For each format description, we generate a format-dependent traversal mechanism that implements a generalized fold over the representation and parse descriptor corresponding to that format. Then, tool developers can write a format-independent, *generic tool* by specifying the behavior of the tool for each PADS/ML type constructor. The traversal mechanism interacts with generic tools through a signature that every generic tool must match.

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

```
module type S = sig
 type state
 ...
 module Int = sig
   val  init    : unit -> state
   val  process : state -> int option -> Pads.pd_header -> state
 end

 module Record : sig
   type partial_state
   val  init          : (string * state) list -> state
   val  start         : state -> Pads.pd_header -> partial_state
   val  project       : state -> string -> state
   val  process_field : partial_state -> string -> state -> partial_state
   val  finish        : partial_state -> state
 end

 module Datatype : sig
   type partial_state
   val  init            : unit -> state
   val  start           : state -> Pads.pd_header -> partial_state
   val  project         : state -> string -> state option
   val  process_variant : partial_state -> string -> state -> partial_state
   val  finish          : partial_state -> state
 end
  ...
end
```

Figure 2.13: Excerpt of generic-tool interface `Generic_tool.S`. The `Int` module is a simplified version of the actual `Int` module in `Generic_tool.S`. A complete listing is provided in Appendix D.

2.4.1 The Generic-Tool Interface

The interface between format-specific traversals and generic tools is specified as an O'CAML signature. For each essential O'CAML built-in type (`int`, `char`, `string`, and `unit`) and for every type constructor in PADS/ML, the signature describes a sub-module that implements the generic tool for that type or type constructor. In addition, it specifies an (abstract) type for auxiliary state that is threaded through the traversal. Figure 2.13 contains an excerpt of the signature that includes the signatures of the `Int`, `Record`, and `Datatype` modules. The signatures of other modules are quite similar. A complete listing of the interface is provided in Appendix D.

The `Int` module contains two functions: `init` to create initial `state` data for an integer field, and `process` to process an integer field based on a parse descriptor and previous state. Notice that the integer argument to `process` is wrapped in an option. This wrapping is necessary because the integer value might

not exist if the parse failed. As the possibility of failure is not limited to integers, the `process` function for all base types receives an optional value argument.

The `Record` module includes a type `partial_state` that allows tools to represent intermediate state in a different form than the general state. The `init` function forms the state of the record from the state of its fields. The `start` function receives the PD header for the data element being traversed and begins processing the element. Function `project` takes a record's state and the name of a field and returns that field's state. Function `process_field` updates the intermediate state of the record based on the name and state of a field, and `finish` converts the finished intermediate state into general tool state. Note that any of these functions could have side effects.

Although the `Datatype` module is similar to the `Record` module, there are some important differences. The `Datatype` `init` function does not start with the state of all the variants. Instead, a variant's state is added during processing so that only variants that have been encountered will have corresponding state. For this reason, `project` returns a `state option`, rather than a `state`. This design is essential for supporting recursive datatypes as trying to initialize the state for all possible variants of the datatype would cause the `init` function to loop infinitely.

The following code snippet gives the signature of the traversal functor as it would appear in the signature `S` from Section 2.3.

```
module Traverse (Tool : Generic_tool.S) :
sig
  val init : unit -> Tool.state
  val traverse : rep -> pd -> Tool.state -> Tool.state
end
```

The functor takes a generic tool generator and produces a format-specific tool with two functions: `init`, to create the initial state for the tool, and `traverse`, which traverses the representation and parse descriptor for the type and updates the given tool state.

2.4.2 Example Tools

We have used this framework to implement a variety of tools useful for processing ad hoc data, including an XML formatter, an accumulator tool for generating statistical overviews of the data, and a data printer for debugging. We briefly describe these tools to illustrate the flexibility of the framework.

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

Figure 2.14: A fragment of the XML output for Sirius.

The XML formatter converts any data with a PADS/ML description into a canonical XML format. This conversion is useful because it allows analysts to exploit the many useful tools that exist for manipulating data in XML. Figure 2.14 shows a sample portion of the output of this tool when run on the Sirius data in Figure 2.1. Appendix E contains a complete listing of the source code of the XML formatter.

The accumulator tool provides a statistical summary of data. Such summaries are useful for developing a quick understanding of data quality. In particular, after receiving a new batch of data, analysts might want to know the frequency of errors, or which fields are the most corrupted. The accumulator tool tracks the distribution of the top n distinct legal values and the percentage of errors. It operates over data sources whose basic structure is a series of records of the same type, providing a summary based on viewing many records in the data source. More complex accumulator programs and a number of other statistical algorithms, like clustering and histogram generation, can easily be implemented using the tool generation infrastructure.

Finally, as an aid in debugging PADS/ML descriptions, we have implemented a simple printing tool. In contrast to the printer generated by the PADS/ML compiler, the output of this tool corresponds to the in-memory representation of the data rather than its original format, which may have concrete syntax or unusual encodings that are not retained in the representation. This format, therefore, is often more readable than the raw data.

2.5 Future Implementation Work and Conclusions

PADS/ML is already an effective, working system for data description and processing. However, there are a number of ways we plan to make it even better.

First, there are a number of properties of data descriptions a programmer might want to infer or verify. For example, it is not hard to write a nonterminating data description by accident. It is also possible to write a description with completely redundant subparts (dead parser code). While these problems might be caught through testing, we would prefer to catch them at compile time. Consequently, we plan to explore development a PADS/ML "type checker" to infer description properties and catch obvious errors.

A second long-term goal is to build a collection of higher-level, format-independent data analysis tools. By "higher-level" tools, we mean tools that perform semantic data analysis as opposed to simpler, low-level syntactic transformation (such as XML conversion) and analysis. Tools in this category include tools

for content-based search, clustering, statistical data modeling, data generation and machine learning. We believe that if we can automatically generate stand-alone, end-to-end tools that perform these functions over arbitrary data, we can have a substantial impact on the productivity of many researchers in fields ranging from computational biology to networking. We hope to provide access to these tools through LaunchPADS, our data visualization environment [DFF+06b, DFF+06a], which currently only interfaces with PADS/C.

Third, as mentioned in Section 2.1, ad hoc data sources are often very large scale. Large data volumes often require that the data be processed without loading it into main memory all at once. The PADS/C language accommodates efficient processing of very large-scale data [FG05] by supporting multiple-entry point parsing, which permits a user to write tools that have fixed memory requirements and that can yield a result in one scan of the data source. The PADS/ML language similarly supports multiple-entry point parsing, but has not yet been tested for performance.

Finally, we hope the PADS/ML system can serve as a stimulating and practical test case for researchers studying functional programming language design and implementation. In particular, our "types as modules" compilation strategy pushes up against the very limits of modern module system design — O'CAML's experimental recursive modules do not allow us to implement recursive types as recursive modules in the way we envision. In addition, future PADS/ML programs might be phrased extremely elegantly as (dependently) type-directed programs, but mainstream languages lack either dependent types or type-directed programming features, or, most commonly, both. Lastly, rather than erasing dependent typing information upon translation of PADS/ML into O'CAML, it would be ideal to preserve the dependency and to verify that data processors preserve necessary data invariants. Unfortunately, sufficiently practical and powerful dependent type systems do not currently exist. So while functional languages are clearly the "programming tools of choice for discriminating hackers," many challenges remain in the domain of ad hoc data processing.

Chapter 3

A Theory of Data Description Languages

> The languages people use to communicate with computers differ in their intended aptitudes, towards either a particular application area, or a particular phase of computer use (high level programming, program assembly, job scheduling, *etc.*). They also differ in physical appearance, and more important, in logical structure. The question arises, do the idiosyncrasies reflect basic logical properties of the situations that are being catered for? Or are they accidents of history and personal background that may be obscuring fruitful developments? This question is clearly important if we are trying to predict or influence language evolution.
>
> To answer it we must think in terms, not of languages, but of families of languages. That is to say we must systematize their design so that a new language is a point chosen from a well-mapped space, rather than a laboriously devised construction.
>
> — P. J. Landin, *The Next 700 Programming Languages*, 1966 [Lan66].

Landin asserts that principled programming language design involves thinking in terms of "families of languages" and choosing from a "well-mapped space." However, 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.

We discovered this problem when we first attempted to specify the semantics of PADS/C. Fisher and Gruber designed and implemented PADS/C [FG05], but specified its semantics only informally, in the PADS/C manual [Pad] and the comments in the source code. When we attempted to formally specify the semantics of PADS/C, we found that there were no existing frameworks suitable for the task. We therefore developed the data description calculus DDC to capture the core features of PADS/C [FMW06]. However, our work on DDC resulted in more than just a semantics for PADS/C. It also enabled us to formally understand other data description languages, like PACKETTYPES and DATASCRIPT(discussed in detail in Section 3.4). Given the broad applicability of DDC$^{\alpha}$, we decided to use it to guide our development of PADS/ML. However, the polymorphic types that we wished to include in PADS/ML could not be formalized with DDC. Therefore, we extended DDC with abstractions over types to create DDC$^{\alpha}$. In the process, we also improved the DDC$^{\alpha}$ theory, as noted in Section 3.2. Then, we used DDC$^{\alpha}$as a basis for the design and implementation of PADS/ML.

In the previous chapter, we discussed PADS/ML. In this chapter, we will begin to understand the family of ad hoc data processing languages, of which PADS/ML and PADS/C are but two members. 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 the data description calculus DDC$^{\alpha}$. This calculus uses types from a dependent type theory to describe various forms of ad hoc data: base types to describe atomic pieces of data and type constructors to describe richer structures. We show how to give a denotational semantics to DDC$^{\alpha}$ by interpreting types as parsing functions that map external representations (bits) to data structures in a typed λ-calculus. More precisely, these parsers produce both internal representations of the external data and parse descriptors that pinpoint errors in the original source.

For many domains, researchers have a solid understanding of what makes a “reasonable” or “unreasonable” language. For instance, a reasonable typed language is one in which values of a given type have a well-defined canonical form and “programs don’t go wrong.” On the other hand, when we began this research, it was not at all clear how to decide whether our data description language and its interpretation were “reasonable” or “unreasonable.” A conventional sort of canonical forms property, for instance, is not relevant as the input data source is not under system control, and, as mentioned in Chapter 1, is frequently buggy. Consequently, we have had to define and formalize a new correctness criterion for the language. Briefly, rather than requiring input data be error-free, we require that the internal data structures produced

by parsing satisfy their specification wherever the parse descriptor reports them to be free of errors. Our invariant allows data consumers to rely on the integrity of internal data structures marked as error-free.

To study and compare the PADS languages and/or other data description languages, we advocate elaborating these languages into DDC$^{\alpha}$. The elaboration decomposes the relatively complex, high-level constructs of the language in question into a composition of lower-level DDC$^{\alpha}$ constructs. We have done this decomposition for IPADS, an idealized version of the PADS/C language that captures the essence of the actual implementation. We have also analyzed key features of PADS/ML, PACKETTYPES and DATASCRIPT using our model. The process of giving semantics to these languages highlighted features that were ambiguous or ill-defined in the documentation we had available to us. For example, we have given a semantics to one of the features of PACKETTYPES, its *overlays*, not found in PADS.

The process of developing DDC$^{\alpha}$ delivered additional benefits, beyond the immediate benefit of obtaining a semantics for PADS/C and its relatives. Most significantly, the semantics served as a clear and effective guide in implementing PADS/ML. In addition, since we defined the semantics by reviewing the existing PADS/C implementation, we found (and fixed!) a couple of subtle bugs. Finally, the semantics has also raised several design questions that we are continuing to study.

In summary, this chapter discusses following theoretical and practical contributions:

- We define a semantic framework for understanding and comparing data description languages such as PADS/C, PADS/ML, PACKETTYPES and DATASCRIPT. Prior to our work (first published in POPL '06 [FMW06]), no one had given a formal semantics to any of these languages.

- At the center of the framework is DDC$^{\alpha}$, a calculus of data descriptions based on dependent type theory. We give a denotational semantics to DDC$^{\alpha}$ by interpreting types both as parsers and, more conventionally, as classifiers for parsed data. By "classfiers," we mean "types" in the usual sense for programming languages.

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

- We define IPADS, an idealized data description language that captures the essential features of a number of real data description languages including PADS/C, PADS/ML, PACKETTYPES and DATASCRIPT. We show how to give IPADS a semantics by elaborating it into DDC$^{\alpha}$. 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 used DDC$^{\alpha}$ to experiment with a definition and implementation strategy for recursive and polymorphic types, features not found in any prior ad hoc data description language of which we are aware. Recursive types are essential for representing tree-shaped hierarchical data [Con, Newa] and polymorphic types allow descriptions to be more concise and more easily reused. We have integrated recursion into PADS/C, and included recursion and polymorphism in PADS/ML, using our theory as a guide.

Sections 3.1, 3.2 and 3.3 explain the syntax, semantics and metatheory of DDC$^{\alpha}$. Section 3.4 introduces the IPADS language and demonstrates with it how to give semantics to high-level data description languages by elaborarting them into DDC$^{\alpha}$. Section 3.5 explains a number of ways in which we have made use of our semantics in practice.

3.1 A Data Description Calculus

At the heart of our work is a data description calculus (DDC$^{\alpha}$), containing simple, orthogonal type constructors designed to capture the core features of data description languages. Consequently, the syntax of DDC$^{\alpha}$ is at a significantly lower level of abstraction than that of PADS/ML. Like PADS/ML, however, DDC$^{\alpha}$ presents a type-based model.

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

Syntactically, the primitives of the calculus are similar to the types found in many dependent type systems, with a number of additions specific to the domain of data description. We base our calculus on a dependent type theory because expressions frequently appear within types in data description languages. However, unlike other dependent type systems, DDC$^{\alpha}$ is not part of a programming language. Therefore, there is no "obvious" choice for an expression language from which to draw the expressions that appear in DDC$^{\alpha}$ types.

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

Figure 3.1: DDC$^{\alpha}$ syntax

However, data description languages tend to draw their expressions from their *host language* – the programming language in which their generated software artifacts are encoded. The host language of PADS/ML, for example, is O'CAML. Therefore, we mimic this design in DDC$^{\alpha}$and choose a single language – a variant of F_{ω}– for expressing both the expressions embedded in types and the interpretations of DDC$^{\alpha}$. This host language is discussed further in Section 3.1.2.

3.1.1 DDC$^{\alpha}$ Syntax

Figure 3.1 shows the syntax of DDC$^{\alpha}$. Expressions e and types σ belong to the host language. We use kinds κ to classify types so that we can ensure their well-formedness. Kind T classifies types that describe data. Kinds $\sigma \to \kappa$ and $\mathsf{T} \to \kappa$ describe functions from values to types and type to types, respectively.

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

We provide abstraction $\lambda x.\tau$ and application $\tau\, e$ so that we may parameterize types by expressions. Dependent 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 describe data matching either τ_1 or τ_2. Unlike regular expressions or context-free grammars, which allow nondeterministic choice, sum-type parsers are deterministic, transforming the data according to τ_1 when possible and *only* attempting to use τ_2 if there is an error in τ_1. Intersection types $\tau_1 \,\&\, \tau_2$ describe data that match both τ_1 and τ_2. They transform a single set of bits to produce a pair of values, one from each type. Constrained types $\{x{:}\tau \mid e\}$ transform data according to the underlying type τ and then check that the constraint e holds when x is bound to the parsed value.

The type $\tau\, \texttt{seq}(\tau_s, e, \tau_t)$ represents a sequence of values of type τ. The type τ_s specifies the type of the separator found between elements of the sequence. For sequences without separators, we use `unit` as the separator type. Expression e is a boolean-valued function that examines the parsed sequence after each

element is read to determine if the sequence has completed. For example, a function that checks if the sequence has 100 elements would terminate a sequence when it reaches length 100. The type τ_t is used when data following the array will signal termination. Commonly, constrained types are used to specify that a particular value terminates the sequence. For example, the type $\{x{:}\texttt{Pchar} \mid x = ';' \}$ specifies that a semicolon terminates the array. However, if no particular value or set of values terminates the array, then a type that never succeeds (like `bottom`) could be used to ensure that the array is not terminated based on τ_t.

Type variables α are abstract descriptions; they are introduced by recursive types and type abstractions. Recursive types $\mu\alpha.\tau$ describe recursive formats, like lists and trees. Type abstraction $\lambda\alpha.\tau$ and application $\tau\,\tau$ allow us to parameterize types by other types. Type variables α always have kind `T`. Note that we call functions from types to types *type abstractions* in contrast to *value abstractions*, which are functions from values to types.

DDC$^\alpha$ also has a number of "active" types. These types describe actions to be taken during parsing rather than strictly describing the data format. Type `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 "synchronization" type τ is found.

3.1.2 Host Language

In Figure 3.2, we present the host language of DDC$^\alpha$, a straightforward extension of F_ω with recursion[1] and a variety of useful constants and operators. We use this host language both to encode the parsing semantics of DDC$^\alpha$ and to write the expressions that can appear within DDC$^\alpha$ itself.

As the calculus is largely standard, we highlight only its unusual features. The constants include bitstrings B; offsets ω, representing locations in bitstrings; and error codes `ok`, `err`, and `fail`, indicating success, success with errors and failure, respectively. We use the constant `none` to indicate a failed parse. Because of its specific meaning, we forbid its use in user-supplied expressions appearing in DDC$^\alpha$ types.

[1]The syntax for `fold` and `unfold`, particularly the choice of annotating `unfold` with a type, is based on the presentation of recursive types in Pierce [Pie02]

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

Figure 3.2: The syntax of the host language, an extension of F_ω with recursion and a variety of useful constants and operators.

Our expressions include arbitrary length sequences $[\vec{e}]$, sequence append $e\ @\ e'$, and sequence indexing $e\,[e']$.

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

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

The static semantics ($\Gamma \vdash e : \sigma$), operational semantics ($e \rightarrow e'$), and type equivalence ($\sigma \equiv \sigma'$) are those of F_ω extended with recursive functions and iso-recursive types and are entirely standard. See, for example, Pierce [Pie02].

We only specify type abstraction over terms and application when we feel it will clarify the presentation. Otherwise, the polymorphism is implicit. We also omit the usual type and kind annotations on functions, with the expectation the reader can construct them from context.

.

3.2 DDC$^\alpha$ Semantics

The primitives of DDC$^\alpha$ are deceptively simple. Each captures a simple concept, often familiar from type theory. However, in reality, each primitive is multifaceted. Except for abstractions, each type simultaneously describes a collection of valid bit strings, two datatypes in the host language – one for the data representation itself and one for its parse descriptor – and a transformation from bit strings, including invalid ones, into data and corresponding metadata.

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

DDC$^\alpha$ abstractions are a special case of DDC$^\alpha$ types in that they do not directly describe data, but rather are conventional functions that enable the writing of more concise data descriptions. Therefore, their meaning, and that of application, can be expressed independently of any particular semantic interpretation of DDC$^\alpha$. We do so with a small-step *normalization* judgment $\tau \rightarrow \tau'$. However, despite our assigning them a semantics with the normalization judgment, we still interpret them in the other semantic interpretations of DDC$^\alpha$. We do so because the implementations of PADS/C and PADS/ML do not normalize types before translating them, but, rather, translate abstraction and application directly into the host language. The role of normalization, then, is only to provide users with a simple and direct explanation of the meaning of abstraction and application.

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

Table 3.1: DDC^{α} Functions and judgments defined in this section.

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

Table 3.2: F_{ω} judgments referenced in this section.

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

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

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

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

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

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

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

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

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

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

Figure 3.3: DDC$^\alpha$ kinding rules

We begin with a kinding judgment that checks if a type is well formed. We then specify the normalization semantics after which we formalize the three-fold semantics of DDC$^\alpha$ types. For reference, Table 3.1 lists all the functions and judgments defined in this section and a brief description of each. Additionally, Table 3.2 lists all of the F_ω judgments that we reference.

3.2.1 DDC$^{\alpha}$ Kinding

The kinding judgment defined in Figure 3.3 determines well-formed DDC$^{\alpha}$ types. We use two contexts to express our kinding judgment:

$$\Gamma \quad ::= \cdot \mid \Gamma, x{:}\sigma$$

$$\Delta \quad ::= \cdot \mid \Delta, \alpha{:}\mathsf{T}$$

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

$$[\![\cdot]\!]_{\text{rep}} = \cdot \qquad\qquad [\![\cdot]\!]_{\text{PD}} = \cdot$$

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

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

As the rules are mostly straightforward, we highlight just a few of them. In rule BASE, we use the function $\mathcal{B}_{\text{kind}}$ to assign kinds to base types. Base types must be fully applied to arguments of the right type. Once fully applied, all base types have kind T. Rule DEPSUM, for dependent sums, shows that the name of the first component is bound to a pair of a representation and corresponding PD. The semantic functions defined in the next section determine the type of this pair. Type abstractions and recursive types (rules TYABS and REC) restrict their type variable to kind T. This restriction simplifies the metatheory of DDC$^{\alpha}$with little practical impact. Finally, with the introduction of potentially open host types, we must now check in rule COMPUTE that the only (potentially) open type variables in σ are the representation-type variables bound (implicitly) in Δ.

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

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

Figure 3.4: Revised DDC$^\alpha$ Syntax

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

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

Figure 3.5: DDC$^\alpha$ weak-head normalization

the rule for type variables (TyVar) now has a standard form, and the premise of the rule for recursive types (Rec) is now nearly identical to the premise of the rule for type abstractions (TyAbs).

3.2.2 DDC$^\alpha$ Normalization

To specify the rules of normalization, we must first refactor the syntax of DDC$^\alpha$ by distinguishing the subset of weak-head normal types (ν) from all types τ, as shown in Figure 3.4. In addition, we must define type and value substitution for DDC$^\alpha$. The notation $\tau'[\tau/\alpha]$ denotes standard capture-avoiding substitution of types into types, except for constructs that contain an F_ω expression e or type σ. For those constructs, the alternative substitution $[[\![\tau]\!]_{\text{rep}}/\alpha_{\texttt{rep}}][[\![\tau]\!]_{\text{PDb}}/\alpha_{\texttt{PDb}}]$ is applied to the subcomponent expression or type. For example,

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

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

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

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

Figure 3.6: Representation-type interpretation function.

Normalization of DDC$^{\alpha}$ is based on a standard call-by-value small-step semantics of the lambda calculus. We present the rules of the normalization judgment in Figure 3.5.

3.2.3 Representation Semantics

In Figure 3.6, we present the representation type of each DDC$^{\alpha}$ 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$^{\alpha}$ 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 constrained types. Similarly, as value abstraction and application are only relevant for dependency, we translate them according to their underlying types.

In more detail, the DDC$^{\alpha}$ type `unit` consumes no input and produces only the `unit` value. Correspondingly, `bottom` consumes no input, but uniformly fails, producing the value `none`. The function $\mathcal{B}_{\text{type}}$ maps each base type to a representation for successfully parsed data. Note that this representation does not depend on the argument expression. As base type parsers can fail, we sum this type with `none` to produce the actual representation type. Intersection types produce a pair of values, one for each sub-type, because

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

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

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

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

Figure 3.7: Parse-descriptor type interpretation function

the representations of the subtypes need not be identical nor even compatible. Constrained types produce sums, where a left branch indicates the data satisfies the constraint and the right indicates it does not. In the latter case, the parser returns the offending data rather than `none` because the error is semantic rather than syntactic. Sequences produce a host language sequence paired with its length.

A type variable α in DDC$^\alpha$ is mapped to a corresponding type variable $\alpha_{\texttt{rep}}$ in F_ω. Recursive types generate recursive representation types with the type variable named appropriately. Polymorphic types and their application become F_ω type constructors and type application, respectively. The output of a `compute` is exactly the computed value, and therefore shares its type. The output of `absorb` is a sum indicating whether parsing the underlying type succeeded or failed. The type of `scan` is similar, but also returns an element of the underlying type in case of success.

In Figure 3.7, we give the parse descriptor type for each DDC$^\alpha$ type. Each PD type has a header and body. This common shape allows us to define functions that polymorphically process PDs based on their headers. Each header stores the number of errors encountered during parsing, an error code indicating the degree of success of the parse – success, success with errors, or failure – and the span of data described by

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

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

Figure 3.8: $F_\omega$types for parsing functions.

the descriptor. Formally, the type of the header (`pd_hdr`) is `int` $*$ `errcode` $*$ `span`. Each body consists of subdescriptors corresponding to the subcomponents of the representation and any type-specific metadata. For types with neither subcomponents nor special metadata, we use `unit` as the body type.

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

Like other types, DDC$^\alpha$ type variables α are translated into a pair of a header and a body. The body has abstract type $\alpha_{\texttt{PDb}}$. This translation makes it possible for polymorphic parsing code to examine the header of a PD, even though it does not know the DDC$^\alpha$ type it is parsing. DDC$^\alpha$ abstractions are translated into F_ω type constructors that abstract the body of the PD (as opposed to the entire PD) and DDC$^\alpha$ applications are translated into F_ω type applications where the argument type is the PD-body type.

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

3.2.4 Parsing Semantics of the DDC$^{\alpha}$

The parsing semantics of a type τ with kind T is a function that transforms some amount of input into a pair of a representation and a parse descriptor, the types of which are determined by τ. The parsing semantics for types with higher kind are functions that construct parsers, or functions that construct functions that construct parsers, and so forth. Figure 3.8 specifies the host-language types of the functions generated from well-kinded DDC$^{\alpha}$ types. For each (unparameterized) type, the input to the corresponding parser is a bit string to parse and an offset at which to begin parsing. The output is a new offset, a representation of the parsed data, and a parse descriptor.

Figure 3.9 shows the parsing semantics function. For each type, the input to the corresponding parser is a bit string and an offset which indicates the point in the bit string at which parsing should commence. The output is a new offset, a representation of the parsed data, and a parse descriptor. As the bit string input is never modified, it is not returned as an output. In addition to specifying how to handle correct data, each function describes how to transform corrupted bit strings, marking detected errors in a parse descriptor. The semantics function is partial, applying only to well-formed DDC$^{\alpha}$ types.

For any type, there are three steps to parsing: parse the subcomponents of the type (if any), assemble the resultant representation, and tabulate metadata based on subcomponent 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 many of the types. For some types, we additionally factor the PD header construction into a separate function. For example, the representation and PD constructors for `unit` are $\mathtt{R}_{\mathtt{unit}}$ and $\mathtt{P}_{\mathtt{unit}}$, respectively, and the header constructor for dependent sums is $\mathtt{H}_{\Sigma}$. The constructor functions are shown in Figure 3.11 and Figure 3.12. We have also factored out some commonly occuring code into auxiliary functions, explained as needed and defined formally in Figure 3.10.

The PD constructors determine the error code and calculate the error count. There are three possible error codes: `ok`, `err`, and `fail`, corresponding to the three possible results of a parse: it can succeed, parsing the data without errors; it can succeed, but discover errors in the process; or, it can find an unrecoverable error and fail. Note that the purpose of the `fail` code is to indicate to any higher level elements that some form of error recovery is required. Hence, the whole parse is marked as failed exactly when the parse ends in failure. The error count is determined by subcomponent error counts and any errors associated directly with the type itself. If a subcomponent has errors then the error count is increased by one; otherwise it is not increased at all. We use the function `pos`, which maps all positive numbers to 1 (leaving zero as is),

$\boxed{[\![\tau]\!]_{\mathtt{p}} = e}$

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

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

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

$[\![\lambda x.\tau]\!]_{\mathtt{p}} = \lambda x.[\![\tau]\!]_{\mathtt{p}}$

$[\![\tau\ e]\!]_{\mathtt{p}} = [\![\tau]\!]_{\mathtt{p}}\ e$

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

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

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

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

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

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

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

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

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

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

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

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

Figure 3.9: DDC$^{\alpha}$ parsing semantics

```
Eof : bits * offset → bool

scanMax : int

fun max (m, n) = if m > n then m else n

fun pos n = if n = 0 then 0 else 1

fun isOk p = pos(p.h.nerr) = 0

fun isErr p = pos(p.h.nerr) = 1

fun max_ec (ec1, ec2) =
 if ec1 = fail or ec2 = fail then fail
 else if ec1 = err or ec2 = err then err
 else ok
```

Figure 3.10: Auxiliary functions. The type of PD headers is `int * errcode * span`. We refer to the projections using dot notation as `nerr`, `ec` and `sp`, respectively. A span is a pair of offsets, referred to as `begin` and `end`, respectively.

to assist in calculating the contribution of subcomponents to the total error count. Errors at the level of the element itself - such as constraint violation in constrained types - are generally counted individually.

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

Sums first attempt to parse according to the left type, returning the resulting value if it parses without errors. Otherwise, it parses according to the right type. Intersections read both types starting at the same offset. They advance the stream to the maximum of the two offsets returned by the component parsers. The construction of the parse descriptor is similar to that of products. For constrained types, we call the parser for the underlying type τ, bind `x` to the resulting rep and PD, and check whether constraint is satisfied. The

```
fun R_unit () = ()

fun P_unit ω = ((0, ok, (ω, ω)), ())


fun R_bot () = none

fun P_bot ω = ((1, fail, (ω, ω)), ())


fun R_Σ (r1, r2) = (r1, r2)

fun H_Σ (h1, h2) =
 let nerr = pos(h1.nerr) + pos(h2.nerr) in
 let ec = if h2.ec = fail then fail
    else max_ec h1.ec h2.ec in
 let sp = (h1.sp.begin, h2.sp.end) in
    (nerr, ec, sp)

fun P_Σ (p1, p2) = (H_Σ(p1.h, p2.h), (p1, p2))


fun R_+left r = inl r

fun R_+right r = inr r

fun H_+ h = (pos(h.nerr), h.ec, h.sp)

fun P_+left p = (H_+ p.h, inl p)

fun P_+right p = (H_+ p.h, inr p)


fun R_& (r, r') = (r, r')

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

fun P_& (p1, p2) = (H_& (p1.h, p2.h), (p1, p2))
```

Figure 3.11: Constructor functions, part 1. The type of parse descriptor headers is `int` $*$ `errcode` $*$ `span`. We refer to the projections using dot notation as `nerr`, `ec` and `sp`, respectively. A span is a pair of offsets, referred to as `begin` and `end`, respectively.

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

fun P_con (c, p) =
 if c then ((pos(p.h.nerr), p.h.ec, p.h.sp), p)
 else ((1 + pos(p.h.nerr), max_ec err p.h.ec, p.h.sp), p)


fun R_seq_init () = (0, [])

fun P_seq_init ω = ((0, ok, (ω, ω)), (0, 0, []))

fun R_seq (r, r_e) = (r.len + 1, r.elts @ [r_e])

fun H_seq (h, h_s, h_e) =
 let eerr = if h.neerr = 0 and h_e.nerr > 0
   then 1 else 0 in
 let nerr = h.nerr + pos(h_s.nerr) + eerr in
 let ec = if h_e.ec = fail then fail
   else max_ec h.ec h_e.ec in
 let sp = (h.sp.begin, h_e.sp.end) in
   (nerr, ec, sp)

fun P_seq (p, p_s, p_e) =
 (H_seq (p.h, p_s.h, p_e.h),
 (p.neerr + pos(p_e.h.nerr), p.len + 1, p.elts @ [p_e]))


fun R_compute r = r

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


fun R_absorb p = if isOk(p) then inl () else inr none

fun P_absorb p = (p.h, ())


fun R_scan r = inl r

fun P_scan (i, p) =
 let nerr = pos(i) + pos(p'.h.nerr) in
 let ec = if nerr = 0 then ok else err in
 let hdr = (nerr, ec, (p.sp.begin − i, p.sp.end)) in
   (hdr, inl (i, p))

fun R_scan_err () = inr none

fun P_scan_err ω = let hdr = (1, fail, (ω, ω)) in
 (hdr, inr ())
```

Figure 3.12: Constructor functions, part 2.

result indicates whether the data has a semantic error and is used in constructing the representation and PD. For example, the PD constructor will add one to the error count if the constraint is not satisfied. Notice that we advance the stream independent of whether the constraint was satisfied.

Sequences have the most complicated semantics because the number of subcomponents depends upon a combination of the data, the termination predicate, and the terminator type. Consequently, the sequence parser uses 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`. Then, the body of the parser creates an initial sequence representation and parse descriptor and then checks whether the sequence described is empty. If not, it reads an element and creates a new rep and PD for the sequence. Note that it passes the PD for `unit` in place of a separator PD, as no separator is read before the first element. Finally, it continues reading the sequence with a call to `continue`.

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

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

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

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

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

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

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

3.3 Metatheory

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

We present the following two properties as critical invariants of our theory. We feel that they should hold, in some form, for any data description language.

- **Parser Type Correctness**: For a DDC$^{\alpha}$ type τ, the representation and PD output by the parsing function of τ will have the types specified by $[\![\tau]\!]_{\text{rep}}$ and $[\![\tau]\!]_{\text{PD}}$, respectively.
- **Canonical Forms of Parsed Data**: We give a precise characterization of the results of parsers by defining the *canonical forms* of representation-parse descriptor pairs associated with a dependent DDC$^{\alpha}$ type. Of particular relevance to data description, we show that the errors reported in the parse descriptor will accurately reflect the errors present in the representation.

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

$$\frac{}{\tau \rightarrow_0 \tau} \qquad \frac{\tau \rightarrow \tau' \quad \tau' \rightarrow_k \tau''}{\tau \rightarrow_{k+1} \tau''}$$

$$\frac{}{e \rightarrow_0 e} \qquad \frac{e \rightarrow e' \quad e' \rightarrow_k e''}{e \rightarrow_{k+1} e''}$$

Figure 3.13: K-steps normalization and evaluation judgments

conversion of the same variable where it appears in the translation of the type. Last, all constructors, support functions and base-type parsers are closed with respect to user-defined variable names.

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

Condition 1 (Conditions on Base-types)

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

The evaluation of F_ω terms and the normalization of DDC$^\alpha$ types are both defined with a small-step semantics. However, it is useful to be able to reason about terms and types that are related by arbitrary many (k) steps in these semantics, rather than just one. To this end, in Figure 3.3, we define two judgments that respectively generalize evaluation and normalization to k steps. Next, we state some properties of these judgments.

Lemma 2 (Properties of K-step Evaluation)

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

Proof: By induction on the number of steps in evaluation relation. ■

Lemma 3 (Properties of K-step Normalization)

1. *If* $\tau_1 \rightarrow_k \tau_1'$ *then* $\tau_1\ \tau_2 \rightarrow_k \tau_1'\ \tau_2$.
2. *If* $\tau_2 \rightarrow_k \tau_2'$ *then* $\nu\ \tau_2 \rightarrow_k \nu\ \tau_2'$.
3. *If* $\tau_1 \rightarrow_k \tau_1'$ *then* $\tau_1\ e \rightarrow_k \tau_1'\ e$.
4. *If* $e \rightarrow_k e'$ *then* $\nu\ e \rightarrow_k \nu\ e'$.
5. *If* $\tau_1 \rightarrow_i \tau_2$ *and* $\tau_2 \rightarrow_j \tau_3$ *then* $\tau_1 \rightarrow_{(i+j)} \tau_3$.

Proof: By induction on the number of steps in evaluation relation. ■

Lemma 4 (K-step Evaluation Inversion)

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

Proof: By induction on the number of steps in the evaluation relation. ■

Lemma 5 (Confluence of Evaluation)

If $e \rightarrow_k v$ *and* $e \rightarrow_i e'$ *then* $e' \rightarrow_{k-i} v$.

Proof: By induction on the height of the first derivation, using determinacy of single-step evaluation as needed. ■

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

Definition 6 (Expression Equivalence)

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

Lemma 7 (Properties of Expression Equivalence)

1. *If* $e \equiv e'$ *and* $e \rightarrow_k e_1$ *then* $\exists\ e'_1$ *s.t.* $e' \rightarrow_k e'_1$ *and* $e_1 \equiv e'_1$.
2. *If* $e \equiv e'$ *then* $e_1[e/x] \equiv e_1[e'/x]$.
3. *If* $\sigma \equiv \sigma'$ *then* $e[\sigma/\alpha] \equiv e[\sigma'/\alpha]$.
4. $e \equiv e$.
5. *If* $e \equiv e'$ *then* $e' \equiv e$.
6. *If* $e \equiv e'$ *and* $e' \equiv e''$ *then* $e \equiv e''$.

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

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

Lemma 8 (Properties of F_ω Type Equivalence)

1. *If* $\Gamma \vdash \sigma :: \kappa$ *and* $\sigma \equiv \sigma'$ *then* $\Gamma \vdash \sigma' :: \kappa$.
2. *If* $\Gamma, x{:}\sigma, \Gamma' \vdash e : \sigma_1$ *and* $\sigma \equiv \sigma'$ *then* $\Gamma, x{:}\sigma', \Gamma' \vdash e : \sigma_1$.

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

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

Lemma 9 (Commutativity of Substitution and Semantic Interpretation)

1. $[\![\tau[\tau'/\alpha]]\!]_{rep} = [\![\tau]\!]_{rep}\langle\tau'/\alpha\rangle$.
2. *If* $\Delta; \Gamma \vdash \tau : \kappa$ *then* $[\![\tau[\tau'/\alpha]]\!]_{rep} = [\![\tau]\!]_{rep}[[\![\mu\alpha.\tau]\!]_{rep}/\alpha_{\texttt{rep}}]$.

3. *If* $\exists\sigma$ *s.t.* $[\![\tau]\!]_{PD} = \sigma$ *and* $\exists\sigma$ *s.t.* $[\![\tau']\!]_{PD} \equiv \texttt{pd\_hdr} * \sigma$ *then* $[\![\tau[\tau'/\alpha]]\!]_{PD} \equiv [\![\tau]\!]_{PD}\langle\tau'/\alpha\rangle = [\![\tau]\!]_{PD}[[\![\tau]\!]_{PDb}/\alpha_{\texttt{PDb}}]$.

4. *If* $\exists\sigma$ *s.t.* $[\![\tau]\!]_{PD} = \sigma$ *and* $\exists\sigma$ *s.t.* $[\![\tau']\!]_{PD} \equiv \texttt{pd\_hdr} * \sigma$ *then* $[\![\tau[\tau'/\alpha]]\!]_{P} \equiv [\![\tau]\!]_{P}\{\tau'/\alpha\}$.

5. $[\![\tau[v/x]]\!]_{rep} = [\![\tau]\!]_{rep}$.

6. $[\![\tau[v/x]]\!]_{PD} = [\![\tau]\!]_{PD}$.

7. $[\![\tau[v/x]]\!]_{P} = [\![\tau]\!]_{P}[v/x]$.

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

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

Lemma 10

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

Proof: By induction on the size of the kind, using Lemma 9 for T case. ■

Lemma 11

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

Proof: By induction on the structure of types. ■

Next we present some standard type-theoretic results for DDC^{α} kinding and normalization.

Lemma 12 (DDC^{α} Preservation)

If $\vdash \tau : \kappa$ *and* $\tau \rightarrow^{*} \nu$ *then* $\vdash \nu : \kappa$.

Proof: By induction on the kinding derivation. ■

Lemma 13 (DDC$^\alpha$ Inversion)

All kinding rules are invertable.

Proof: By inspection of the kinding rules. ■

Lemma 14 (DDC$^\alpha$ Canonical Forms)

If $\vdash \nu : \kappa$ *then either*

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

Proof: By kinding rules and grammar of normalized types ν. ■

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

Lemma 15 (F_ω Substitution)

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

Proof: These are standard properties of F_ω. They are all proven by induction on the height of the first derivation. ■

Lemma 16 (DDC$^\alpha$ Substitution)

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

Proof: For both parts, by induction on the first derivation, using Lemma 15 as needed. ■

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

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

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

Proof: By induction on the number of steps in the evaluation. Within the induction, we proceed using a case-by-case analysis of the possible structures of type τ. The complete proof is shown in Appendix A. ■

3.3.1 Type Correctness

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

Demonstrating that generated parsers are well formed and have the expected types is nontrivial primarily because the generated code expects parse descriptors to have a particular shape, and it is not completely obvious they do in the presence of polymorphism. Hence, to prove type correctness, we first need to characterize the shape of parse descriptors for arbitrary DDC$^{\alpha}$ types.

The particular shape required is that every parse descriptor be a pair of a header and an (arbitrary) body. The most straightforward characterization of this property is too weak to prove directly, so we instead characterize it as a logical relation in Definition 18. Lemma 22 establishes that the logical relation holds of all well-formed DDC^{α} types by induction on kinding derivations, and the desired characterization follows as a corollary.

Definition 18

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

Lemma 19

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

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

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

Lemma 20

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

Proof: By induction on the structure of the kind. ■

Lemma 21

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

Proof: By induction on the structure of the kind. The proof is detailed in Appendix A. ■

Lemma 22

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

Proof: By induction on the height of the kinding derivation. A number of the more challenging cases are shown in Appendix A. ■

Corollary 23

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

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

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

Definition 24 (DDC$^\alpha$ Kind Interpretation in F_ω)

- $K(\mathsf{T}) = \mathsf{T}$
- $K(\sigma \to \kappa) = K(\kappa)$
- $K(\mathsf{T} \to \kappa) = \mathsf{T} \to K(\kappa)$

Lemma 25 (Representation-Type Well Formedness)

If $\Delta; \Gamma \vdash \tau : \kappa$ *then*

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

Proof: By induction using Lemma 22 and Lemma 8, part 1. ■

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

Lemma 26 (Types of Constructors)

- $\mathtt{R}_{\mathtt{unit}} : \mathtt{unit} \to \mathtt{unit}$
- $\mathtt{P}_{\mathtt{unit}} : \mathtt{offset} \to \mathtt{pd\_hdr} * \mathtt{unit}$

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

- $\mathtt{P_{scan}} : \forall\alpha.\mathtt{int} * \alpha\,\mathtt{pd} \rightarrow \mathtt{pd\_hdr} * ((\mathtt{int} * \alpha\,\mathtt{pd}) + \mathtt{unit})$
- $\mathtt{R_{scan\_err}} : \forall\alpha.\mathtt{unit} \rightarrow \alpha + \mathtt{none}$
- $\mathtt{P_{scan\_err}} : \forall\alpha.\mathtt{offset} \rightarrow \mathtt{pd\_hdr} * ((\mathtt{int} * \alpha) + \mathtt{unit})$

Proof: By typing rules of F_ω. ■

With our next lemma, we establish the type correctness of the generated parsers. We prove the lemma using a general induction hypothesis that applies to open types. This hypothesis must account for the fact that any free type variables in a DDC$^\alpha$ type τ will become free function variables in $[\![\tau]\!]_{\mathrm{p}}$. To that end, we define the function $[\![\Delta]\!]_{\mathrm{PT}}$ which maps type-variable contexts Δ in the DDC$^\alpha$ to value-variable contexts Γ in F_ω.

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

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

Proof: By induction on the height of the kinding derivation. A number of the more challenging cases are shown in Appendix A. ■

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

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

3.3.2 Canonical Forms

DDC$^\alpha$ parsers generate pairs of representations and parse descriptors designed to satisfy a number of invariants. Of greatest importance is the fact that when the parse descriptor reports that there are no errors in a particular substructure, the programmer can count on the representation satisfying all of the syntactic and

semantic constraints expressed by the dependent DDC^α type description. When a parse descriptor and representation satisfy these invariants and correspond properly, we say the pair of data structures is *canonical* or in *canonical form*.

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

A second definition, $\text{Canon}^*{}_\tau(r, p)$ normalizes τ to a ν, thereby eliminating outermost type and value applications. Then, the requirements on ν are given by $\text{Canon}_\nu(r, p)$. For brevity, we write $p.h.nerr$ as $p.nerr$ and use `pos` to denote the function that returns zero when passed zero and one when passed another natural number.

Definition 29 (Canonical Forms I)

$\text{Canon}_\nu(r, p)$ *iff exactly one of the following is true:*

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

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

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

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

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

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

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

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

Definition 30 (Canonical Forms II)

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

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

Lemma 31 (Constructors Produce Values in Canonical Form)

- $\mathrm{Canon}_{\mathtt{unit}}(\mathtt{R_{true}}(), \mathtt{P_{true}}(\omega))$.

- $\mathrm{Canon}_{\mathtt{bottom}}(\mathtt{R_{false}}(), \mathtt{P_{false}}(\omega))$.

- *If* $\mathrm{Canon}^*{}_{\tau_1}(r_1, p_1)$ *and* $\mathrm{Canon}^*{}_{\tau_2[(r,p)/x]}(r_2, p_2)$ *then* $\mathrm{Canon}_{\Sigma\, x:\tau_1.\tau_2}(\mathtt{R_\Sigma}(\mathtt{r_1}, \mathtt{r_2}), \mathtt{P_\Sigma}(\mathtt{p_1}, \mathtt{p_2}))$.

- *If* $\mathrm{Canon}^*{}_{\tau}(r, p)$ *then* $\mathrm{Canon}_{\tau+\tau'}(\mathtt{R_{+left}}(\mathtt{r}), \mathtt{P_{+left}}(\mathtt{p}))$.

- *If* $\mathrm{Canon}^*{}_{\tau}(r, p)$ *then* $\mathrm{Canon}_{\tau'+\tau}(\mathtt{R_{+right}}(\mathtt{r}), \mathtt{P_{+right}}(\mathtt{p}))$.

- *If* $\mathrm{Canon}^*{}_{\tau_1}(r_1, p_1)$ *and* $\mathrm{Canon}^*{}_{\tau_2}(r_2, p_2)$ *then* $\mathrm{Canon}_{\tau_1\ \&\ \tau_2}(\mathtt{R_\&}(\mathtt{r_1}, \mathtt{r_2}), \mathtt{P_\&}(\mathtt{p_1}, \mathtt{p_2}))$.

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

- $\mathrm{Canon}_{\tau\ \mathtt{seq}(\tau_s,e,\tau_t)}(\mathtt{R}_{\mathtt{seq\_init}}(), \mathtt{P}_{\mathtt{seq\_init}}(\omega))$.
- *If* $\mathrm{Canon}_{\tau\ \mathtt{seq}(\tau_s,e,\tau_t)}(r,p)$ *and* $\mathrm{Canon}^*{}_\tau(r',p')$ *then, for any* p'', $\mathrm{Canon}_{\tau\ \mathtt{seq}(\tau_s,e,\tau_t)}(\mathtt{R}_{\mathtt{seq}}(\mathtt{r},\mathtt{r}'), \mathtt{P}_{\mathtt{seq}}(\mathtt{p},\mathtt{p}'',\mathtt{p}'))$.
- $\mathrm{Canon}_{\mathtt{compute}(e:\sigma)}(\mathtt{R}_{\mathtt{compute}}(\mathtt{e}), \mathtt{P}_{\mathtt{compute}}(\omega))$.
- $\mathrm{Canon}_{\mathtt{absorb}(\tau)}(\mathtt{R}_{\mathtt{absorb}}(\mathtt{p}), \mathtt{P}_{\mathtt{absorb}}(\mathtt{p}))$.
- *If* $\mathrm{Canon}^*{}_\tau(r,p)$ *then* $\mathrm{Canon}_{\mathtt{scan}(\tau)}(\mathtt{R}_{\mathtt{scan}}(\mathtt{r}), \mathtt{P}_{\mathtt{scan}}(\mathtt{i},\mathtt{p}))$.
- $\mathrm{Canon}_{\mathtt{scan}(\tau)}(\mathtt{R}_{\mathtt{scan\_err}}(), \mathtt{P}_{\mathtt{scan\_err}}(\omega))$.

Proof: By inspection of the constructor functions. ■

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

Condition 32 (Base Type Parsers Produce Values in Canonical Form)

If $\vdash v : \sigma$, $\mathcal{B}_{kind}(C) = \sigma \to \mathsf{T}$ *and* $\mathcal{B}_{imp}(C)\ v\ (B,\omega) \to^*\ (\omega', r, p)$ *then* $\mathrm{Canon}_{C(v)}(r,p)$.

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

Lemma 33 (Parsing to Canonical Forms)

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

Proof: By induction on the height of the second derivation – that is, the number of steps taken to evaluate. Within the induction, we proceed using a case-by-case analysis of the possible structures of type τ. A number of the more challenging cases are shown in Appendix A. ■

Corollary 34

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

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

3.4 Encoding DDLs in DDC$^{\alpha}$

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

3.4.1 IPADS: An Idealized DDL

IPADS is an idealized version of the PADS/C language that captures many of the common features of DDLs. The two essential differences between PADS/C and IPADS are that many of the compound constructs have been replaced by simpler, orthogonal contstructs, and some of the subtler syntactic features of PADS/C have been eliminated. Though the syntax differs, the structure of PADS/C's relatives PADS/ML, PACKETTYPES and DATASCRIPT are similar. Hence, IPADS serves as an effective idealization of these languages as well. Some features, however, are particular to a given language, and are therefore introduced as separate IPADS extensions, later in this section.

IPADS data descriptions are types. Complex IPADS descriptions are built by using type constructors to glue together a collection of simpler types, with the simplest being base types like those of PADS/ML. 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.

Figure 3.14 summarizes the formal syntax of IPADS. As with DDC$^{\alpha}$, expressions e and types σ are taken from the host language, described in Section 3.1.2. Notice that we use x for host language variables and α for IPADS type variables. $C(e)$ denotes a base type parameterized by a value. **`Pfun`** introduces value-parameterized types and **`Plit`** c describes a literal in the data source. **`Pstruct`**s describe sequences, much like PADS/ML records. **`Punion`** is a simplified version of PADS/ML datatypes, supporting only description of variance in the data source. **`Parray`** describes homogenous sequences like the PADS/ML built-in type `Plist`. However, the separator and terminator of **`Parray`** are specified as types rather than literals. **`Pwhere`** specifies constraints, **`Popt`** allows for an optional element, and **`Prec`** introduces

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

Figure 3.14: The syntax of the IPADS data description langauge.

recursive types. **Pcompute** is identical to `compute` of DDC$^\alpha$. **Palt** is an intersection type; it describes data that is described by all the branches simultaneously and produces a set of values - one from each type.

3.4.2 IPADS Elaboration

We specify the elaboration from IPADS to DDC$^\alpha$ with two judgments: $p \Downarrow \tau$ prog indicates that the IPADS program p is encoded as DDC$^\alpha$ type τ, while $t \Downarrow \tau$ does the same for IPADS types t. These judgments are defined in Figure 3.15.

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

3.4.3 Beyond IPADS

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

PADS/C *switched unions.* A switched union, like a **Punion**, indicates variability in the data format with a set of alternative formats (branches). However, instead of trying each branch in turn, the switched union takes an expression that determines which branch to use. Typically, this expression depends upon data

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

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

$\boxed{t \Downarrow \tau}$

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

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

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

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

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

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

Figure 3.15: Encoding IPADS in DDC$^{\alpha}$

read earlier in the parse. Each branch is preceded by a tag, and the first branch whose tag matches the expression is selected. If none match then the default branch t_{def} is chosen. The syntax of a switched union is **`Pswitch`** e $\{\overrightarrow{e \Rightarrow x{:}t}\ t_{\text{def}}\}$.

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

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

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

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

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

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

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

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

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

Next, we extend IPADS programs to include datatype bindings. Datatype bindings include the name of the type, α, a list of type parameters ($\overrightarrow{\alpha}$), a single value parameter x, and a body that consists of a list

of named variants. As with **Prec** bindings, we do not specify the meaning of datatype bindings in DDC$^{\alpha}$ directly. Rather, we decompose a given datatype into a compound IPADS type, which is then substituted into the remainder of the program.

$$\text{Programs} \quad p \quad ::= \quad ... \mid \textbf{Pdatatype}\ \alpha\ \ (\overrightarrow{\alpha})(x : \sigma) = \{\overrightarrow{x{:}t}\};\ p$$

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

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

Our choice to limit PADS/ML uniform datatypes was based on three factors: first, and foremost, we lacked any compelling examples that demanded nonuniform datatypes; second, recursion over higher-order types significantly complicates both the theory of DDC$^{\alpha}$ and the implementation of PADS/ML; lastly, there is no support in O'CAML for polymorphic recursion [2].

DATASCRIPT *arrays.* Next, we introduce DATASCRIPT-style arrays for binary data, $t\ [\mathit{length}]$. 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. Arrays with the length field specfied can be encoded in a straightforward manner with DDC$^{\alpha}$ sequences:

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

As these arrays have neither separators nor terminators, we use `unit` (always succeeds, parsing nothing) and `bottom` (always fails, parsing nothing), respectively, for separator and terminator. The function f takes

[2] The absence might be due to the fact that type inference for polymorphicly-recursive functions (without type annotations) is undecidable [Hen93].

a pair of array representation and PD and compares the sequence length recorded in the representation to $length$.

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

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

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

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

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

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

PACKETTYPES *overlays.* Finally, we consider the *overlay* construct found in PACKETTYPES. An overlay allows description authors "to merge two type specifications by embedding one within the other, as is done when one protocol is *encapsulated* within another. Overlay[s] introduce additional substructure to an already existing field." [MC00a]. 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), 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 elaboration of the overlay syntax into DDC$^{\alpha}$. In essence, overlays are syntactic sugar: overlaying a subfield of a given type replaces the type of that subfield with a new type. However, despite the essentially syntactic nature of overlays, we discovered a critical subtlety of semantic significance, not mentioned by the PACKETTYPES authors. Any expressions in the original type that refer to the overlayed field may no longer be well typed after applying the overlay. For example, consider extending FPPacket with a field that is constrained to be equal to the checksum of the body:

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

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

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

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

We know of a couple of features from data description languages that we cannot implement in DDC$^{\alpha}$ as it stands. First, we cannot support a label construct that permits the user to rewind the input. Second, DATASCRIPT allows the element type of an array to reference the representation of the array itself [Bac02] (see, in particular, the example in Figure 5). This feature can be useful, for example, if the element type

needs the index of the array element that is currently being processed. DDC$^{\alpha}$ does not support this kind of element-type parameterization. However, we do not view such limitations as particularly troublesome. Like the λ-calculus or π-calculus, DDC$^{\alpha}$ is intended to capture many common language features, while providing a convenient and effective basis for extension with new features.

3.5 Applications of the Semantics

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

3.5.1 Bug Hunting

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

The semantics also helped us avoid potential nontermination of array parsers. In the original implementation of PADS/C arrays, it was possible to write nonterminating arrays, a bug that was only uncovered when it hung a real program. We have fixed the bug and used the semantics to verify our fix. [3]

3.5.2 Principled Language Implementation

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

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

semantics, we struggled to disentangle the invariants related to polymorphism. After we had defined the calculus, we were able to implement type abstractions as O'CAML functors in approximately a week. We hope the calculus will serve as a guide for implementations of PADS in other host languages.

3.5.3 Distinguishing the Essential from the Accidental

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

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

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

3.6 Future Work and Conclusions

In the spirit of Landin, we have taken the first steps toward specifying a semantics for the family of data description languages by defining the data description calculus DDC$^{\alpha}$. This calculus, which is a dependent

type theory with a simple set of orthogonal primitives, is expressive enough to describe the features of PADS/C, PADS/ML, 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 revet the data by-hand.

We have already used the semantics to identify bugs in the implementation of PADS/C and to guide the implementation of PADS/ML. In addition, when various biological data sources [Newa, Con] motivated adding recursion to PADS/C and PADS/ML, we used DDC$^{\alpha}$ for design guidance. After adding recursion, both PADS languages can now describe the biological data sources. Furthermore, the DDC$^{\alpha}$ framework has repeatedly proved useful for sketching the design of new tools before implementing them in PADS/ML. For example, we sketched both the PADS/ML printers and traversal functors for DDC$^{\alpha}$ before implementing them for PADS/ML. This sketching is possible as many tools can be thought of as an interpretation of the DDC$^{\alpha}$ types, much like the parser.

Our work on DDC$^{\alpha}$ has suggested a number of possible directions for future work, of which we will briefly describe two. First, we have begun work on studying the semantics of data printers. We have formalized our printer semantics as a semantic interpretation of DDC$^{\alpha}$ (based on the printer sketches discussed above), and we have stated and proven some of their basic properties [MFW^{+}06]. We have also begun to consider which properties we might expect to hold of the interaction between the parser and printer of any given description.

Second, we would like to enhance our support for expressing error recovery mechanisms in DDC$^{\alpha}$. The `scan` type provides a very simple error recovery mechanism that is similar to the *local* error recovery mechanisms of many early versions of the YACC parser generator [App98]. These mechanisms operate, in essence, by deleting input tokens until a particular *synchronizing* token is found. However, the choice of where and when to attempt error recovery, and which synchronizing tokens to use, is not made automatically, but must be specified within the grammar itself with special error recovery rules. Yet, more advanced error recovery mechanisms exist that take a substantially different approach to error recovery. For example, *global error repair* "finds the smallest set of insertions and deletions that would turn the source string into a syntactically correct string, even if the insertions or deletions are not at a point where an LL or LR parser

would first report an error" [App98]. In addition, global error repair does not depend on explicit error recovery rules, but instead uses a single, uniform mechanism for the entire grammar.

We would like to support such global error repair in the DDC$^{\alpha}$ framework. However, adding a new set of type constructors to DDC$^{\alpha}$ would be insufficient, as it would still require that error recovery be specified as part of the description and would be limited to local recovery due to the orthogonal nature of types. Instead, support for a global mechanism would likely require that we parameterize the parsing semantics itself by an error recovery mechanism. Furthermore, as the exact operation of the error repair, including the choice of which tokens to insert or delete, depends on the particular description, we hypothesize that the error recovery mechanism itself should be specified as an interpretation of DDC$^{\alpha}$.

Chapter 4

Related Work and Conclusions

In this thesis we have described the PADS/ML data description language and DDC$^{\alpha}$, a low-level, data description calculus with which we can specify the semantics of PADS/ML and other data description languages. We now review a selection of the related work and offer some concluding remarks.

4.1 Related Work

To give an overview of the related work, it is important to distinguish between PADS/ML – a real, implemented data description language, and DDC$^{\alpha}$ – a formalism for understanding data description languages. We will begin our discussion with an overview of work related to the PADS/ML language, followed by an overview of the work related to DDC$^{\alpha}$.

4.1.1 PADS/ML

As we discussed in a number of places in this thesis, PADS/ML evolved from prior work by Fisher and Gruber on PADS/C [FG05]. For the reader's convenience, we review the differences between the two languages here. First, PADS/C is targeted at the C language, while PADS/ML is targeted at the ML family of languages. Using ML as the host language simplifies the implementation of many data processing tasks, like data transformation, which benefit from ML's pattern matching and high level of abstraction. Second, unlike PADS/C types, PADS/ML types may be parameterized by other types, resulting in more concise descriptions through code reuse. ML-style datatypes and anonymous nested tuples also help improve the

readability and compactness of descriptions. Third, the generic tool architecture of PADS/ML delivers a number of benefits over the fixed architecture of PADS/C. In PADS/C, all tools are generated from within the compiler. Therefore, developing a new tool generator requires understanding and modifying the compiler. Furthermore, the user selects the set of tools to generate when compiling the description. In PADS/ML, tool generators can be developed independent of the compiler and they can be developed more rapidly because the boilerplate code to traverse data need not be replicated for each tool generator. In addition, the user can choose which tools to generate for a given data format on a program-by-program basis. This flexibility is possible because tool generation is simply the composition of the desired generic tool modules with the traversal functor. A final difference between PADS/C and PADS/MLis that PADS/C is more mature than PADS/ML. However, we are actively developing PADS/ML and expect that this will only be a temporary difference.

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

More modern compiler construction tools alleviate several of the problems of Lex and Yacc by providing more built-in programming support. For instance, DEMETER's class dictionaries [Lie88] can generate parsers that construct internal parse trees as well as traversal functions, much like the traversal functions generated by PADS/ML. Similarly, the ANTLR parser generator [PQ95] allows the user to add annotations to a grammar to direct construction of a parse tree. However, all nodes in the abstract syntax tree have a single type, hence the guidance is rather crude when compared with the richly-typed structures that can be constructed using PADS/ML. The SABLE/CC compiler construction tool [Agn98] goes beyond ANTLR by producing LALR(1) parsers along with richly-typed ASTs quite similar to those of PADS/ML. Also like PADS/ML, descriptions do not contain actions. Instead, actions are only performed on the generated ASTs.

Yet, for all of their advantages over LEX and YACC, DEMETER, ANTLR, SABLE/CC, and other such tools differ from PADS/ML in a number of significant ways. None of these tools have dependent and polymorphic data descriptions or a formal semantics. They are based around grammars, rather than types, which forces

users to familiarize themselves with a new formalism. They target only ASCII and UNICODE sources. Finally, their error handling strategies are different than those of PADS/ML and they do not provide the programmer with programmatic access to errors, as PADS/ML does with parse descriptors. That said, such a laundry list of differences risks obscuring the more essential difference – that these tools are targeted at a different domain. PADS/C and PADS/ML generate tools specificly suited to processing ad hoc data (like the accumulator), whereas the others generate tools suited to the processing and analysis of programs.

There are parallels between PADS/ML types and some of the elements of parser combinator libraries found in languages like Haskell [Bur75, HM98]. Likewise, there are libraries to help programmers generate printers. Each of these technologies is very useful in its own domain, but PADS/ML is broader in its scope than each of them: a single PADS/ML description is sufficient to generate *both* a parser and a printer. And a statistical error analysis, a format debugger, an XML translator, and in the future, a query engine [FFGM06], a content-based search engine [LJW+06, Oh06], more statistical analyses, *etc.* Combinator libraries are not designed to generate such a range of artifacts from a single specification. Indeed, the proper way to think about combinator libraries in relation to PADS/ML is that they might serve as an alternative implementation strategy for some of the generated tools.

The networking community has developed a number of domain-specific languages that are substantially closer to PADS/ML than either compiler-construction tools or combinator libraries. These include PACKETTYPES [MC00a], DATASCRIPT [Bac02] and Bro's [Pax99] packet processing language for parsing and printing binary data. Like PADS/C and PADS/ML, these languages have a type-directed approach to describing ad hoc data and permit the user to define semantic constraints. In contrast to our work, these systems handle only binary data and assume the data is error-free or halt parsing if an error is detected. Not only are ASCII formats a common part of many software monitoring systems, parsing nonbinary data poses additional challenges because of the need to handle delimiter values and to express richer termination conditions on sequences of data. PacketTypes and DataScript also focus exclusively on the parsing/printing problem, whereas our work exploits the declarative nature of our data descriptions to automatically generate other useful tools and programming libraries.

Our support for generic tools is related to generic programming [JJ96, Hin00, LP03] and design patterns like the visitor. Both are technologies that can facilitate the implementation of type-directed data structure traversals. Lammel and Peyton Jones' original "scrap your boilerplate" article [LP03] provides a detailed summary of the trade-offs between different techniques. We investigated using one of these techniques

before implementing the generator for PADS/ML traversal functors. However, we found that most advanced techniques for functional programming languages require features, like type classes, that are only present in variants of Haskell. The generated PADS/ML traversal functors are less flexible than some of these traversal techniques, but they suffice for helping us program the tools we have implemented, and for many more tools for ad hoc data that we are considering implementing. In addition, these techniques support only standard functional-programming types, whereas PADS/ML consists of dependent types (specialized to the domain of ad hoc data processing). However, at this point, this distinction is more in priniciple than in practice, as we currently provide only minimal support for PADS/ML's dependent type constructors in the generic tool interface.

Perhaps one of the most closely related works on generic programming is that of van Weelden *et al* [vWSP05], as it relates to the generated parser and printers, rather than only to the generic tool support. The authors investigate the use of polytypic programming to produce a parser for a language based only on the specification of its AST type(s). In this way, the AST types themselves serve as the grammar for the language. They also investigate applying this approach to other compiler-related analyses, like scope checking and type inference. However, while their "types-as-grammar" approach is clearly related to PADS/ML, they are using standard functional-programming types, and they are targeting the domain of programming languages. Dependent types like those of PADS/ML and support for ad hoc data processing are beyond the scope of their work.

There are a number of tools designed to deal with converting ad hoc data formats into XML and various related tasks. For instance, XSugar [BMS05] allows users to specify an alternative non-XML syntax for XML languages using a context-free grammar. This tool automatically generates conversion tools between XML and non-XML syntax. The Binary Format Description language (BFD) [MC00b] is a fragment of XML that allows programmers to specify binary and ASCII formats. BFD is able to convert the raw data into XML-tagged data where it can then be processed using XML-processing tools. While both these tools are useful for many tasks, conversion to XML is not always the answer. Such conversion often results in an 8-10 times blowup in data size over the native form. PADS/ML, on the other hand, avoids this blowup by processing data in its native form. The conversion process also does not directly help programmers get their hands on the data.

DFDL is a data format specification language with an XML-based syntax and type structure [Dat05, BW04]. DFDL is a language *specification*, not an entire system or an implementation. Like the PADS/ML

language, DFDL has a rich collection of base types and supports a variety of ambient codings. In terms of expressiveness, we believe the DFDL consortium has added dependency and semantic constraints to match the expressiveness of PADS/C. However, because the specification is still under development, we cannot give a more detailed comparison at this point.

A somewhat different class of languages includes ASN.1 [Dub01] and ASDL [WAKS97]. Both of these systems specify the *logical* in-memory representation of data and then automatically generate a *physical* on-disk representation. Although useful for many purposes, this technology does not help process data that arrives in predetermined, ad hoc formats. Another language in this category is the Hierarchical Data Format 5 (HDF5) [Hie]. This file format allows users to store scientific data, but it does not help users deal with legacy ad hoc formats like PADS/ML does.

XDTM [MZF+05, ZDF+05] uses XML Schema to describe the locations of a collection of sources spread across a local file system or distributed across a network of computers. However, XDTM has no means of specifying the contents of files, so XDTM and PADS/ML solve complementary problems. The METS schema [MET03] is similar to XDTM as it describes metadata for objects in a digital library, including a hierarchy such objects.

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

4.1.2 DDC$^{\alpha}$

To the best of our knowledge, our work on DDC$^{\alpha}$ is the first to provide a formal interpretation of dependent types as parsers and to study the properties of these parsers including error correctness and type safety. Of course, there are other formalisms for defining parsers, most famously, regular expressions and context-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 [BU73] and revitalized more recently by Ford [For04], evolved from context-free grammars but have deterministic, prioritized choice like DDC$^{\alpha}$ as opposed to nondeterministic choice. Though PEGs have syntactic lookahead operators, they may be parsed in linear time through the use of "packrat parsing" techniques [For02, Gri06]. Once again, our multiple interpretations of types in DDC$^{\alpha}$ makes our theory substantially different from the theory of PEGs.

As with PADS/ML, there are many parallels between DDC$^{\alpha}$ and *parser combinators* [Bur75, HM98]. In particular, DDC$^{\alpha}$'s dependent sum construct is reminiscent of the bind operator in the monadic formulation of parser combinators. Indeed, we can model dependent sums in Haskell as:

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

Parser combinators, however, are a general approach to specifying recursive descent parsing, whereas we have targeted DDC$^{\alpha}$ to the more-specific domain of parsing ad hoc data. This focus leads to many features not found in parser combinators, including the implicit type/value correspondence, the error response mechanism, and arrays. Each of these features is as fundamental to DDC$^{\alpha}$ as dependent sums. These two approaches demonstrate the idea of a spectrum of domain-specificity in languages. The relationship between parser combinators and DDC$^{\alpha}$ is like the relationship between a general purpose language and parser combinators themselves. That is, while parser combinators form an (embedded) domain-specific language, DDC$^{\alpha}$ constructs form a language that is even more domain-specific.

4.2 Concluding Remarks

Ad hoc data presents its users with a great number of challenges and can be found in a wide variety of disciplines. The general rule seems to be that if an area involves some form of data processing, then there are ad hoc data formats to be found. The problems of ad hoc data processing, therefore, are not a niche interest, but an essential problem in computer science. Moreover, they are not likely to go away anytime soon. The existence of ad hoc data formats is not caused by the shortsightedness or inexperience of data format designers. Rather, new discoveries and new applications often legitimately demand new data formats, yet format standardization is a slow and difficult process. While XML is an extremely flexible

and standardized format, it is not appropriate for all data sources, particularly very large ones. For these data sources, the blow-up in data size and the performance hit of processing the XML can make its use untenable.

We hope that our work on data description languages, as described in this thesis, will make a significant contribution both to data analysts in need of tools like PADS/ML and to computer scientists eager to tackle the many challenges of ad hoc data. Our goal in presenting PADS/ML was not only to describe what we have accomplished, but to inspire and guide other researchers in building versions of PADS for their favorite programming languages. Similarly, our aim in presenting DDC$^{\alpha}$ was not only to provide a semantics to a number of existing data description languages, but to pave the way for a clear understanding of the semantics of future data description languages. We hope that there will be many.

However, our vision for the PADS/ML and PADS/C languages does not stop there. Ultimately, we think that every data source should carry with it its own description. That description would be written in a low-level language (perhaps like DDC$^{\alpha}$), into which descriptions from many other, higher-level descriptions could be compiled. Furthermore, going beyond the PADS/C and PADS/ML languages themselves, we want to allow data consumers to access their data with high-level, intuitive tools that require no programming and free them to focus on their goals. If we can enable 1000 cancer researchers to become just 1% more effective in their work, then we will have "created" (in terms of time) the equivalent of 10 new researchers. Of course, we don't intend to be satisfied with helping just 1000 cancer researchers. Given the large quantity and near ubiquity of existing ad hoc data, we strive to improve the data access of millions of people and for many years to come.

Appendix A

Proofs of Selected Lemmas and Theorems

Proof: Lemma 9, part 3.

Case α': $\quad \exists\, \sigma$ s.t. $[\![\alpha']\!]_{\mathrm{PD}} = \sigma \qquad \exists\, \sigma$ s.t. $[\![\tau']\!]_{\mathrm{PD}} \equiv \mathtt{pd\_hdr} * \sigma$.

If $\alpha' = \alpha$ then $[\![\alpha'[\tau'/\alpha]]\!]_{\mathrm{PD}} = [\![\tau']\!]_{\mathrm{PD}}$. From premise, we know

$$[\![\tau']\!]_{\mathrm{PD}} \equiv \mathtt{pd\_hdr} * [\![\tau']\!]_{\mathrm{PDb}} = (\mathtt{pd\_hdr} * \alpha_{\mathtt{PDb}})[[\![\tau']\!]_{\mathrm{PDb}}/\alpha_{\mathtt{PDb}}] = [\![\alpha']\!]_{\mathrm{PD}}[[\![\tau']\!]_{\mathrm{PDb}}/\alpha_{\mathtt{PDb}}].$$

So, $[\![\alpha'[\tau'/\alpha]]\!]_{\mathrm{PD}} \equiv [\![\alpha']\!]_{\mathrm{PD}}[[\![\tau']\!]_{\mathrm{PDb}}/\alpha_{\mathtt{PDb}}]$. As no variables of the form $\alpha_{\mathtt{rep}}$ appear, the result is equal to $[\![\alpha']\!]_{\mathrm{PD}}[[\![\tau']\!]_{\mathrm{rep}}/\alpha_{\mathtt{rep}}][[\![\tau']\!]_{\mathrm{PDb}}/\alpha_{\mathtt{PDb}}]$.

Proof: Lemma 9, part 4.

Case $\tau\, e$:

$[\![\tau\, e]\!]_{\mathrm{P}}\{\tau'/\alpha\} = [\![\tau]\!]_{\mathrm{P}}\{\tau'/\alpha\}\ e\{\tau'/\alpha\}$. As e cannot contain $\mathtt{parse}_{\alpha}$, $e\{\tau'/\alpha\} = e\langle\tau'/\alpha\rangle$. By induction, $[\![\tau]\!]_{\mathrm{P}}\{\tau'/\alpha\} \equiv [\![\tau[\tau'/\alpha]]\!]_{\mathrm{P}}$. Taken together, we have

$$[\![\tau]\!]_{\mathrm{P}}\{\tau'/\alpha\}\ e\{\tau'/\alpha\} \equiv [\![\tau[\tau'/\alpha]]\!]_{\mathrm{P}}\, e[[\![\tau']\!]_{\mathrm{rep}}/\alpha_{\mathtt{rep}}][[\![\tau']\!]_{\mathrm{PDb}}/\alpha_{\mathtt{PDb}}],$$

which, by the definition of substitution is equal to $[\![(\tau\, e)[\tau'/\alpha]]\!]_{\mathrm{P}}$.

Case $\tau_1\,\tau_2$:

$$[\![\tau_1\,\tau_2]\!]_{\mathrm{P}}\{\tau'/\alpha\} = [\![\tau_1]\!]_{\mathrm{P}}\{\tau'/\alpha\}\ [[\![\tau_2]\!]_{\mathrm{rep}}\{\tau'/\alpha\}][[\![\tau_2]\!]_{\mathrm{PDb}}\{\tau'/\alpha\}][\![\tau_2]\!]_{\mathrm{P}}\{\tau'/\alpha\}$$

By part 2, $[\![\tau_2]\!]_{\mathrm{rep}}\{\tau'/\alpha\} \equiv [\![\tau_2[\tau'/\alpha]]\!]_{\mathrm{rep}}$ and $[[\![\tau_2]\!]_{\mathrm{PDb}}\{\tau'/\alpha\}] \equiv [\![\tau_2[\tau'/\alpha]]\!]_{\mathrm{PDb}}$. So, by Lemma 7, part 3,

$$\begin{aligned}[\![\tau_1]\!]_{\mathrm{P}}\{\tau'/\alpha\}\ [[\![\tau_2]\!]_{\mathrm{rep}}\{\tau'/\alpha\}][[\![\tau_2]\!]_{\mathrm{PDb}}\{\tau'/\alpha\}][\![\tau_2]\!]_{\mathrm{P}}\{\tau'/\alpha\}& \\ \equiv [\![\tau_1]\!]_{\mathrm{P}}\{\tau'/\alpha\}\ [[\![\tau_2[\tau'/\alpha]]\!]_{\mathrm{rep}}][[\![\tau_2[\tau'/\alpha]]\!]_{\mathrm{PDb}}][\![\tau_2]\!]_{\mathrm{P}}\{\tau'/\alpha\}.& \quad \text{(A.1)}\end{aligned}$$

The remainder of this case is analogous to the previous case. It is proven by applying the induction hypothesis to the subcomponent types.

Case $\mu\alpha.\tau$:

Analagous to $\tau_1\,\tau_2$. We use part 2 of the lemma for the type annotations.

Proof: Lemma 17.

Case all but App and TyApp: As $[\![\tau]\!]_{\mathrm{P}}$ is a value and τ is normal, result is immediate.

Case App: $\tau = \tau_1\,e \quad [\![\tau_1\,e]\!]_{\mathrm{P}} \rightarrow_k v$

$\vdash \tau : \kappa \quad [\![\tau_1 e]\!]_{\mathrm{P}} = [\![\tau_1]\!]_{\mathrm{P}} e$

By Lemma 13, $\tau_1 : \sigma \rightarrow \kappa$ and $e : \sigma$. By Lemma 4, $[\![\tau_1]\!]_{\mathrm{P}} \rightarrow_i v_1$ and $e \rightarrow_j v_2$, with $i + j < k$. By induction,

$$\tau_1 \rightarrow^* \nu_1, \quad \text{(A.2)}$$

$$v_1 \equiv [\![\nu_1]\!]_{\mathrm{P}}, \quad \text{(A.3)}$$

$$[\![\tau_1]\!]_{\mathrm{rep}} \equiv [\![\nu_1]\!]_{\mathrm{rep}}, \quad \text{(A.4)}$$

$$[\![\tau_1]\!]_{\mathrm{PD}} \equiv [\![\nu_1]\!]_{\mathrm{PD}}. \quad \text{(A.5)}$$

By (A.2) and DDC^{α} Preservation (Lemma 12), $\nu_1 : \sigma \rightarrow \kappa$. So, by Lemma 14, $\nu_1 = \lambda x.\tau_i$.

By Lemma 2 and (A.3), $[\![\tau_1\,e]\!]_{\mathrm{P}} = [\![\tau_1]\!]_{\mathrm{P}}\,e \rightarrow_{(i+j)} v_1 v_2 \equiv [\![\nu_1]\!]_{\mathrm{P}}\,v_2$.

By Lemma 5, $v_1\ v_2 \rightarrow_{k'} v$, where $k' = k - i - j$. By Lemma 7, part 1, $[\![\nu_1]\!]_{\mathrm{P}}\ v_2 \rightarrow_{k'} v'$ and $v' \equiv v$.

Now, as $\nu_1 = \lambda x.\tau_i$, we have $[\![\nu_1]\!]_{\mathrm{P}} = \lambda x.[\![\tau_i]\!]_{\mathrm{P}}$. By evaluation rules, $\lambda x.[\![\tau_i]\!]_{\mathrm{P}}\ v_2 \rightarrow [\![\tau_i]\!]_{\mathrm{P}}[v_2/x]$ which, by Lemma 9, $= [\![\tau_i[v_2/x]]\!]_{\mathrm{P}}$. So, by Lemma 5, $[\![\tau_i[v_2/x]]\!]_{\mathrm{P}} \rightarrow_{(k'-1)} v'$.

By Lemma 3 and DDC^{α} normalization,

$$\tau_1\ e \rightarrow^* \nu_1\ v_2 = \lambda x.\tau_i\ v_2 \rightarrow \tau_i[v_2/x].$$

So, by DDC^{α} Preservation (Lemma 12), $\vdash \tau_i[v_2/x] : \kappa$.

By induction,

$$\tau_i[v_2/x] \rightarrow^* \nu, \tag{A.6}$$

$$v' \equiv [\![\nu]\!]_{\mathrm{P}}, \tag{A.7}$$

$$[\![\tau_i[v_2/x]]\!]_{\mathrm{rep}} \equiv [\![\nu]\!]_{\mathrm{rep}}, \tag{A.8}$$

$$[\![\tau_i[v_2/x]]\!]_{\mathrm{PD}} \equiv [\![\nu]\!]_{\mathrm{PD}}. \tag{A.9}$$

Now, we prove the four necessary conlcusions in order. First, by Lemma 3, part 5, $\tau_1\ e \rightarrow^* \nu$. Second, as $v' \equiv v$, by Lemma 7 5, $v \equiv [\![\nu]\!]_{\mathrm{P}}$. Third,$[\![\tau_1 e]\!]_{\mathrm{rep}} = [\![\tau_1]\!]_{\mathrm{rep}} \equiv [\![\nu_1]\!]_{\mathrm{rep}} = [\![\lambda x.\tau_i]\!]_{\mathrm{rep}} = [\![\tau_i]\!]_{\mathrm{rep}}$, which, by Lemma 9, $= [\![\tau_i[v_2/x]]\!]_{\mathrm{rep}}$. So, by transitivity of type equivalence, $[\![\tau_1 e]\!]_{\mathrm{rep}} \equiv [\![\tau_i[v_2/x]]\!]_{\mathrm{rep}} \equiv [\![\nu]\!]_{\mathrm{rep}}$. Last, by the same argument, $[\![\tau_1 e]\!]_{\mathrm{PD}} \equiv [\![\tau_i[v_2/x]]\!]_{\mathrm{PD}} \equiv [\![\nu]\!]_{\mathrm{PD}}$

Case TyApp: $\tau = \tau_1\ \tau_2 \quad [\![\tau_1\tau_2]\!]_{\mathrm{P}} \rightarrow_k v$

$\vdash \tau : \kappa \qquad [\![\tau_1\tau_2]\!]_{\mathrm{P}} = [\![\tau_1]\!]_{\mathrm{P}}[[\![\tau_2]\!]_{\mathrm{rep}}][[\![\tau_2]\!]_{\mathrm{PDb}}][\![\tau_2]\!]_{\mathrm{P}}$

The proof for TyApp is similar to App, but more complex due to the more complicated parsing semantics of TyApp. As before, we start by proving our induction hypothesis for the subcomponent types τ_1 and τ_2.

By Lemma 13, $\tau_1 : \mathsf{T} \rightarrow \kappa$ and $\tau_2 : \mathsf{T}$.

By Lemma 4 parts 1 & 2, $[\![\tau_1]\!]_{\mathrm{P}} \rightarrow_i v_1$ and $[\![\tau_2]\!]_{\mathrm{P}} \rightarrow_j v_2$, , with $i + j < k$. By induction,

$$\tau_1 \rightarrow^* \nu_1, \tag{A.10}$$

$$v_1 \equiv [\![\nu_1]\!]_{\mathrm{P}}, \tag{A.11}$$

$$[\![\tau_1]\!]_{\mathrm{rep}} \equiv [\![\nu_1]\!]_{\mathrm{rep}}, \tag{A.12}$$

$$[\![\tau_1]\!]_{\mathrm{PD}} \equiv [\![\nu_1]\!]_{\mathrm{PD}}. \tag{A.13}$$

and

$$\tau_2 \rightarrow^* \nu_2, \tag{A.14}$$

$$v_2 \equiv [\![\nu_2]\!]_{\mathrm{P}}, \tag{A.15}$$

$$[\![\tau_2]\!]_{\mathrm{rep}} \equiv [\![\nu_2]\!]_{\mathrm{rep}}, \tag{A.16}$$

$$[\![\tau_2]\!]_{\mathrm{PD}} \equiv [\![\nu_2]\!]_{\mathrm{PD}}. \tag{A.17}$$

By DDC^{α} Preservation (Lemma 12),

$$\nu_1 : \mathsf{T} \rightarrow \kappa \tag{A.18}$$

$$\nu_2 : \mathsf{T}. \tag{A.19}$$

So, by Lemma 14, $\nu_1 = \lambda\alpha.\tau_i$. By definition of $[\![\,\cdot\,]\!]_{\mathrm{PDb}}$ and (A.17), $[\![\tau_2]\!]_{\mathrm{PDb}} \equiv [\![\nu_2]\!]_{\mathrm{PDb}}$.

Now, by Lemma 2, part 1, and (A.11),

$$\begin{aligned}[\![\tau_1\tau_2]\!]_{\mathrm{P}} &= [\![\tau_1]\!]_{\mathrm{P}}[[\![\tau_2]\!]_{\mathrm{rep}}][[\![\tau_2]\!]_{\mathrm{PDb}}][\![\tau_2]\!]_{\mathrm{P}} \\ &\rightarrow_i v_1[[\![\tau_2]\!]_{\mathrm{rep}}][[\![\tau_2]\!]_{\mathrm{PDb}}][\![\tau_2]\!]_{\mathrm{P}} \equiv [\![\nu_1]\!]_{\mathrm{P}}[[\![\tau_2]\!]_{\mathrm{rep}}][[\![\tau_2]\!]_{\mathrm{PDb}}][\![\tau_2]\!]_{\mathrm{P}}\end{aligned} \tag{A.20}$$

By Lemma 5, $v_1[[\![\tau_2]\!]_{\mathrm{rep}}][[\![\tau_2]\!]_{\mathrm{PDb}}][\![\tau_2]\!]_{\mathrm{P}} \rightarrow_{(k-i)} v$ By Lemma 7 1a., $[\![\nu_1]\!]_{\mathrm{P}}[[\![\tau_2]\!]_{\mathrm{rep}}][[\![\tau_2]\!]_{\mathrm{PDb}}][\![\tau_2]\!]_{\mathrm{P}} \rightarrow_{(k-i)} v'$ and $v' \equiv v$.

Now, as $\nu_1 = \lambda\alpha.\tau_i$, $[\![\nu_1]\!]_{\mathrm{P}} = \Lambda\alpha_{\mathtt{rep}}.\Lambda\alpha_{\mathtt{PDb}}.\lambda\mathtt{parse}_\alpha.[\![\tau_i]\!]_{\mathrm{P}}$. By the evaluation rules,

$$
\begin{aligned}
&(\Lambda\alpha_{\mathtt{rep}}.\Lambda\alpha_{\mathtt{PDb}}.\lambda\mathtt{parse}_\alpha.[\![\tau_i]\!]_{\mathrm{P}})[[\![\tau_2]\!]_{\mathrm{rep}}][[\![\tau_2]\!]_{\mathrm{PDb}}][\![\tau_2]\!]_{\mathrm{P}} \\
&\to (\Lambda\alpha_{\mathtt{PDb}}.\lambda\mathtt{parse}_\alpha.[\![\tau_i]\!]_{\mathrm{P}})[[\![\tau_2]\!]_{\mathrm{rep}}/\alpha_{\mathtt{rep}}][[\![\tau_2]\!]_{\mathrm{PDb}}][\![\tau_2]\!]_{\mathrm{P}} \\
&= (\Lambda\alpha_{\mathtt{PDb}}.\lambda\mathtt{parse}_\alpha.[\![\tau_i]\!]_{\mathrm{P}}[[\![\tau_2]\!]_{\mathrm{rep}}/\alpha_{\mathtt{rep}}])[[\![\tau_2]\!]_{\mathrm{PDb}}][\![\tau_2]\!]_{\mathrm{P}} \\
&\to (\lambda\mathtt{parse}_\alpha.[\![\tau_i]\!]_{\mathrm{P}}[[\![\tau_2]\!]_{\mathrm{rep}}/\alpha_{\mathtt{rep}}])[[\![\tau_2]\!]_{\mathrm{PDb}}/\alpha_{\mathtt{PDb}}][\![\tau_2]\!]_{\mathrm{P}} \\
&= (\lambda\mathtt{parse}_\alpha.[\![\tau_i]\!]_{\mathrm{P}}[[\![\tau_2]\!]_{\mathrm{rep}}/\alpha_{\mathtt{rep}}][[\![\tau_2]\!]_{\mathrm{PDb}}/\alpha_{\mathtt{PDb}}])[\![\tau_2]\!]_{\mathrm{P}}
\end{aligned}
$$

As $[\![\tau_2]\!]_{\mathrm{P}} \to_j v_2$,

$$
\begin{aligned}
&\to_j (\lambda\mathtt{parse}_\alpha.[\![\tau_i]\!]_{\mathrm{P}}[[\![\tau_2]\!]_{\mathrm{rep}}/\alpha_{\mathtt{rep}}][[\![\tau_2]\!]_{\mathrm{PDb}}/\alpha_{\mathtt{PDb}}])\ v_2 \\
&\to [\![\tau_i]\!]_{\mathrm{P}}[[\![\tau_2]\!]_{\mathrm{rep}}/\alpha_{\mathtt{rep}}][[\![\tau_2]\!]_{\mathrm{PDb}}/\alpha_{\mathtt{PDb}}][v_2/\mathtt{parse}_\alpha]
\end{aligned}
$$

By Lemma 7.2,

$$\equiv [\![\tau_i]\!]_{\mathrm{P}}[[\![\tau_2]\!]_{\mathrm{rep}}/\alpha_{\mathtt{rep}}][[\![\tau_2]\!]_{\mathrm{PDb}}/\alpha_{\mathtt{PDb}}][[\![\nu_2]\!]_{\mathrm{P}}/\mathtt{parse}_\alpha]$$

By Lemma 7.3,

$$\equiv [\![\tau_i]\!]_{\mathrm{P}}[[\![\nu_2]\!]_{\mathrm{rep}}/\alpha_{\mathtt{rep}}][[\![\nu_2]\!]_{\mathrm{PDb}}/\alpha_{\mathtt{PDb}}][[\![\nu_2]\!]_{\mathrm{P}}/\mathtt{parse}_\alpha]$$

By DDC$^\alpha$ Inversion (Lemma 13), $\alpha{:}\mathsf{T};\cdot \vdash \tau_i : \kappa$, so by (A.19) and Lemma 22, $\mathrm{H}(\tau_i : \kappa)$ and $\mathrm{H}(\tau_2 : \mathsf{T})$. By definition of H, $\exists\,\sigma$ s.t. $[\![\tau_i]\!]_{\mathrm{PD}} \equiv \mathtt{pd\_hdr} * \sigma$ and $\exists\,\sigma$ s.t. $[\![\tau_2]\!]_{\mathrm{PD}} \equiv \mathtt{pd\_hdr} * \sigma$. so, by Lemma 9, $(A.19) \equiv [\![\tau_i[\nu_2/\alpha]]\!]_{\mathrm{P}}$.

By Lemma 5,

$$[\![\tau_i]\!]_{\mathrm{P}}[[\![\tau_2]\!]_{\mathrm{rep}}/\alpha_{\mathtt{rep}}][[\![\tau_2]\!]_{\mathrm{PDb}}/\alpha_{\mathtt{PDb}}][v_2/\mathtt{parse}_\alpha] \to_{(k-i-(j+3))} v'.$$

By Lemma 7 1a.,

$$[\![\tau_i[\nu_2/\alpha]]\!]_{\mathrm{P}} \to_{(k-i-j-3)} v'' \text{ and } v'' \equiv v'.$$

As $\tau_1 \to^* \nu_1$ and $\tau_2 \to^* \nu_2$, by Lemma 3 and DDC^α normalization,

$$\tau_1\, \tau_2 \to^* \nu_1\, \nu_2 = \lambda\alpha.\tau_i\ \nu_2 \to \tau_i[\nu_2/\alpha].$$

So, by DDC^α Preservation (Lemma 12), $\vdash \tau_i[\nu_2/\alpha] : \kappa$.

By induction,

$$\tau_i[\nu_2/\alpha] \to^* \nu, \tag{A.21}$$

$$v'' \equiv [\![\nu]\!]_{\mathrm{P}}, \tag{A.22}$$

$$[\![\tau_i[\nu_2/\alpha]]\!]_{\text{rep}} \equiv [\![\nu]\!]_{\text{rep}}, \tag{A.23}$$

$$[\![\tau_i[\nu_2/\alpha]]\!]_{\text{PD}} \equiv [\![\nu]\!]_{\text{PD}} \tag{A.24}$$

Now, we prove the four necessary conlcusions in order. First, by Lemma 3, part 5, $\tau_1\, \tau_2 \to^* \nu$

Second, as $v'' \equiv v'$ and $v' \equiv v$, by Lemma 7 5, $v \equiv [\![\nu]\!]_{\mathrm{P}}$.

Third,

$$\begin{aligned}
[\![\tau_1 \tau_2]\!]_{\text{rep}} &= [\![\tau_1]\!]_{\text{rep}} [\![\tau_2]\!]_{\text{rep}} \\
&\equiv [\![\nu_1]\!]_{\text{rep}} [\![\nu_2]\!]_{\text{rep}} \\
&= [\![\lambda\alpha.\tau_i]\!]_{\text{rep}} [\![\nu_2]\!]_{\text{rep}} \\
&\equiv [\![\tau_i]\!]_{\text{rep}} [[\![\nu_2]\!]_{\text{rep}}/\alpha_{\texttt{rep}}]
\end{aligned}$$

which, by Lemma 9,

$$= [\![\tau_i[\nu_2/\alpha]]\!]_{\text{rep}}$$

By transitivity of type equivalence,

$$[\![\tau_1\, \tau_2]\!]_{\text{rep}} \equiv [\![\tau_i[\nu_2/\alpha]]\!]_{\text{rep}} \equiv [\![\nu]\!]_{\text{rep}}.$$

Last,

$$\begin{aligned}
[\![\tau_1\tau_2]\!]_{\mathrm{PD}} &= [\![\tau_1]\!]_{\mathrm{PD}}[\![\tau_2]\!]_{\mathrm{PDb}} \\
&\equiv [\![\nu_1]\!]_{\mathrm{PD}}\,[\![\nu_2]\!]_{\mathrm{PDb}} \\
&= [\![\lambda\alpha.\tau_i]\!]_{\mathrm{PD}}\,[\![\nu_2]\!]_{\mathrm{PDb}} \\
&\equiv [\![\tau_i]\!]_{\mathrm{PD}}[[\![\nu_2]\!]_{\mathrm{PDb}}/\alpha_{\mathrm{PDb}}]
\end{aligned}$$

which, by Lemma 9,

$$\equiv [\![\tau_i[\nu_2/\alpha]]\!]_{\mathrm{PD}}$$

By transitivity of type equivalence,

$$[\![\tau_1\tau_2]\!]_{\mathrm{PD}} \equiv [\![\tau_i[\nu_2/\alpha]]\!]_{\mathrm{PD}} \equiv [\![\nu]\!]_{\mathrm{PD}}$$

Proof: Lemma 21.

Case $\kappa = \mathsf{T}$: $\quad \mathrm{H}(\tau : \mathsf{T}) \qquad \mathrm{H}(\tau' : \mathsf{T})$

By definition of H, $\exists\,\sigma$ s.t. $[\![\tau]\!]_{\mathrm{PD}} \equiv \texttt{pd\_hdr} * \sigma$ and $\exists\,\sigma$ s.t. $[\![\tau']\!]_{\mathrm{PD}} \equiv \texttt{pd\_hdr} * \sigma$. Unpacking the first existential with an arbitrary σ, we have $[\![\tau]\!]_{\mathrm{PD}} \equiv \texttt{pd\_hdr} * \sigma$. By Lemma 9, $[\![\tau[\tau'/\alpha]]\!]_{\mathrm{PD}} \equiv [\![\tau]\!]_{\mathrm{PD}}\langle\tau'/\alpha\rangle$. By Pierce, Lemma 30.3.4 (Type Substitution), $[\![\tau]\!]_{\mathrm{PD}}\langle\tau'/\alpha\rangle \equiv (\texttt{pd\_hdr} * \sigma)\langle\tau'/\alpha\rangle$, so, by transitivity, $[\![\tau[\tau'/\alpha]]\!]_{\mathrm{PD}} \equiv (\texttt{pd\_hdr} * \sigma)\langle\tau'/\alpha\rangle = \texttt{pd\_hdr} * \sigma\langle\tau'/\alpha\rangle$. The last equation is possible as `pd_hdr` is closed. Using $\sigma\langle\tau'/\alpha\rangle$ as a witness, we get $\exists\,\sigma'$ s.t. $[\![\tau[\tau'/\alpha]]\!]_{\mathrm{PD}} \equiv \texttt{pd\_hdr} * \sigma'$. This result gives us $\mathrm{H}(\tau[\tau'/\alpha] : \mathsf{T})$.

Case $\kappa = \mathsf{T} \to \kappa'$: $\quad \mathrm{H}(\tau : \mathsf{T} \to \kappa') \qquad \mathrm{H}(\tau' : \mathsf{T})$

We know that $\exists\,\sigma$ s.t. $[\![\tau']\!]_{\mathrm{PD}} \equiv \texttt{pd\_hdr}{*}\sigma$, and $\exists\,\sigma$ s.t. $[\![\tau]\!]_{\mathrm{PD}} \equiv \sigma$ and for all τ_2 s.t. $\mathrm{H}(\tau_2 : \mathsf{T})$, $\mathrm{H}(\tau\,\tau_2 : \kappa')$.
Want to prove: $\exists\,\sigma$ s.t. $[\![\tau[\tau'/\alpha]]\!]_{\mathrm{PD}} \equiv \sigma$ and for all τ_2 s.t.$\mathrm{H}(\tau_2 : \mathsf{T})$, $\mathrm{H}(\tau[\tau'/\alpha]\ \tau_2 : \kappa')$.

Part 1 (the "exists") uses same argument as case $\kappa = \mathsf{T}$.

Part 2: Assume $\mathrm{H}(\tau_2 : \mathsf{T})$ for some arbitrary τ_2. WLOG, we can assume $\alpha \notin \mathsf{FTV}(\tau_2)$, as, if it is, we can always α-vary it. From given, $\mathrm{H}(\tau\,\tau_2 : \kappa')$ By induction, $\mathrm{H}((\tau\,\tau_2)[\tau'/\alpha] : \kappa')$ As $\alpha \notin \mathsf{FTV}(\tau_2)$, $\tau_2[\tau'/\alpha] = \tau_2$ So we have, $\mathrm{H}(\tau[\tau'/\alpha]\ \tau_2 : \kappa')$.

Case $\kappa = \sigma \to \kappa'$:

Analogous to above.

Proof: Lemma 22.

Case Abs: $\Delta; \Gamma \vdash \lambda x.\tau : \sigma \to \kappa \qquad \Delta; \Gamma, x{:}\sigma \vdash \tau : \kappa$

We want to prove that $\mathrm{H}(\lambda x.\tau : \sigma \to \kappa)$. By induction, $\mathrm{H}(\tau : \kappa)$, so $[\![\tau]\!]_{\mathrm{PD}}$ exists, by which we know $[\![\lambda x.\tau]\!]_{\mathrm{PD}}$ exists. Next, for arbitrary e, we wish to prove that $\mathrm{H}((\lambda x.\tau)e : \kappa)$. Now, $[\![(\lambda x.\tau)e]\!]_{\mathrm{PD}} = [\![(\lambda x.\tau)]\!]_{\mathrm{PD}} = [\![\lambda x.\tau]\!]_{\mathrm{PD}} = [\![\tau]\!]_{\mathrm{PD}}$. So, $[\![(\lambda x.\tau)e]\!]_{\mathrm{PD}} \equiv [\![\tau]\!]_{\mathrm{PD}}$. By Lemma 20, and $\mathrm{H}(\tau : \kappa)$, $\mathrm{H}((\lambda x.\tau)e : \kappa)$.

Case Rec: $\Delta; \Gamma \vdash \mu\alpha.\tau : \mathsf{T} \qquad \Delta, \alpha{:}\mathsf{T}; \Gamma \vdash \tau : \mathsf{T}$

We wish to prove that $\mathrm{H}(\mu\alpha.\tau : \mathsf{T})$, that is $\exists\,\sigma$ s.t. $[\![\mu\alpha.\tau]\!]_{\mathrm{PD}} \equiv \mathtt{pd\_hdr} * \sigma$. That is, $\exists\,\sigma$ s.t. $\mathtt{pd\_hdr} * \mu\alpha_{\mathtt{PDb}}.[\![\tau]\!]_{\mathrm{PD}} \equiv \mathtt{pd\_hdr} * \sigma$. By IH, $\mathrm{H}(\tau : \mathsf{T})$ So, $\exists\,\sigma'$ s.t. $[\![\tau]\!]_{\mathrm{PD}} \equiv \sigma'$. Unpacking the existential with arbitrary σ', $\sigma = \mu\alpha_{\mathtt{PDb}}.\sigma'$ serves as a witness for our desired result.

Case TyAbs: $\Delta; \Gamma \vdash \lambda\alpha.\tau : \mathsf{T} \to \kappa \qquad \Delta, \alpha{:}\mathsf{T}; \Gamma \vdash \tau : \kappa$

We wish to prove that $\mathrm{H}(\lambda\alpha.\tau : \mathsf{T} \to \kappa)$, that is $\exists\,\sigma$ s.t. $[\![\lambda\alpha.\tau]\!]_{\mathrm{PD}} \equiv \sigma$ and for all τ' s.t. $\mathrm{H}(\tau' : \mathsf{T})$, $\mathrm{H}((\lambda\alpha.\tau)\tau' : \kappa)$. First, let's prove that $\exists\,\sigma$ s.t. $[\![\lambda\alpha.\tau]\!]_{\mathrm{PD}} \equiv \sigma$. From derivation, $\Delta, \alpha{:}\mathsf{T}; \Gamma \vdash \tau : \kappa$. By induction, $\mathrm{H}(\tau : \kappa)$, so $\exists\,\sigma'$ s.t. $[\![\tau]\!]_{\mathrm{PD}} \equiv \sigma'$. As $[\![\lambda\alpha.\tau]\!]_{\mathrm{PD}} = \lambda\alpha_{\mathtt{PDb}}.[\![\tau]\!]_{\mathrm{PD}}$, we have $\sigma = \lambda\alpha_{\mathtt{PDb}}.\sigma'$ as a witness. Next, let's prove that for all τ' s.t. $\mathrm{H}(\tau' : \mathsf{T})$, $\mathrm{H}((\lambda\alpha.\tau)\ \tau' : \kappa)$. Assume $\mathrm{H}(\tau' : \mathsf{T})$ for some arbitrary τ'. By earlier induction and Lemma 21, $\mathrm{H}(\tau[\tau'/\alpha] : \kappa)$.

Now, if we can prove that $[\![\tau[\tau'/\alpha]]\!]_{\mathrm{PD}} \equiv [\![(\lambda\alpha.\tau)\ \tau']\!]_{\mathrm{PD}}$, then we can use Lemma 20 to obtain our result. By definition of H, $\exists\,\sigma$ s.t. $[\![\tau']\!]_{\mathrm{PD}} \equiv \mathtt{pd\_hdr} * \sigma$, so, by Lemma 9, $[\![\tau[\tau'/\alpha]]\!]_{\mathrm{PD}} \equiv [\![\tau]\!]_{\mathrm{PD}}\langle\tau'/\alpha\rangle = [\![\tau]\!]_{\mathrm{PD}}[[\![\tau']\!]_{\mathrm{PDb}}/\alpha_{\mathtt{PDb}}]$. By Q-AppAbs, $[\![\tau]\!]_{\mathrm{PD}}[[\![\tau']\!]_{\mathrm{PDb}}/\alpha_{\mathtt{PDb}}] \equiv (\lambda\alpha_{\mathtt{PDb}}.[\![\tau]\!]_{\mathrm{PD}})[\![\tau']\!]_{\mathrm{PDb}}$. So, $[\![\tau[\tau'/\alpha]]\!]_{\mathrm{PD}} \equiv (\lambda\alpha_{\mathtt{PDb}}.[\![\tau]\!]_{\mathrm{PD}})[\![\tau']\!]_{\mathrm{PDb}} = [\![(\lambda\alpha.\tau)]\!]_{\mathrm{PD}}[\![\tau']\!]_{\mathrm{PDb}} = [\![(\lambda\alpha.\tau)\ \tau']\!]_{\mathrm{PD}}$. So, by Lemma 20 and $\mathrm{H}(\tau[\tau'/\alpha] : \kappa)$, we have $\mathrm{H}((\lambda\alpha.\tau)\ \tau' : \kappa)$.

Proof: Lemma 27.

Case Rec: $\Delta; \Gamma \vdash \mu\alpha.\tau : \mathsf{T} \qquad \Delta, \alpha{:}\mathsf{T}; \Gamma \vdash \tau : \mathsf{T}$

We wish to prove that $[\![\Delta]\!]_{F_\omega}, \Gamma, [\![\Delta]\!]_{\mathrm{PT}} \vdash [\![\mu\alpha.\tau]\!]_{\mathrm{P}} : [\![\mu\alpha.\tau{:}\mathsf{T}]\!]_{\mathrm{PT}}$.

By induction,

$$[\![\Delta]\!]_{F_\omega}, \alpha_{\mathtt{rep}} :: \mathsf{T}, \alpha_{\mathtt{PDb}} :: \mathsf{T}, \Gamma, [\![\Delta]\!]_{\mathrm{PT}}, \mathtt{parse}_\alpha{:}[\![\alpha{:}\mathsf{T}]\!]_{\mathrm{PT}} \vdash [\![\tau]\!]_{\mathrm{P}} : [\![\tau{:}\mathsf{T}]\!]_{\mathrm{PT}}$$

From derivation and Lemma 25,

$$[\![\Delta]\!]_{F_\omega}, \Gamma, [\![\Delta]\!]_{\mathrm{PT}} \vdash [\![\mu\alpha.\tau]\!]_{\mathrm{rep}} :: \mathsf{T} \tag{A.25}$$

$$[\![\Delta]\!]_{F_\omega}, \Gamma, [\![\Delta]\!]_{\mathrm{PT}} \vdash [\![\mu\alpha.\tau]\!]_{\mathrm{PD}} :: \mathsf{T} \tag{A.26}$$

$$[\![\Delta]\!]_{F_\omega}, \Gamma, [\![\Delta]\!]_{\mathrm{PT}} \vdash [\![\mu\alpha.\tau]\!]_{\mathrm{PDb}} :: \mathsf{T} \tag{A.27}$$

Let $[S] = \langle \mu\alpha.\tau/\alpha \rangle$. By Type Substitution, TAPL Lemma 30.3.4, part 3:

$$[\![\Delta]\!]_{F_\omega}, \Gamma[S], [\![\Delta]\!]_{\mathrm{PT}}[S], \mathtt{parse}_\alpha{:}[\![\alpha{:}\mathsf{T}]\!]_{\mathrm{PT}}[S] \vdash [\![\tau]\!]_{\mathrm{P}}[S] : [\![\tau{:}\mathsf{T}]\!]_{\mathrm{PT}}[S] \tag{A.28}$$

From the derivation and Lemma 22, $\mathrm{H}(\mu\alpha.\tau : \mathsf{T})$. By definition of H, $\exists\, \sigma$ s.t. $[\![\mu\alpha.\tau]\!]_{\mathrm{PD}} \equiv \mathtt{pd\_hdr} * \sigma$ From this result, (A.26), and Lemma 10,

$$[\![\alpha{:}\mathsf{T}]\!]_{\mathrm{PT}}[S] \equiv [\![\alpha[\mu\alpha.\tau/\alpha]{:}\mathsf{T}]\!]_{\mathrm{PT}} = [\![\mu\alpha.\tau{:}\mathsf{T}]\!]_{\mathrm{PT}}$$

This result, (A.28) and Lemma 8, part 2, give us

$$[\![\Delta]\!]_{F_\omega}, \Gamma[S], [\![\Delta]\!]_{\mathrm{PT}}[S], \mathtt{parse}_\alpha{:}[\![\mu\alpha.\tau{:}\mathsf{T}]\!]_{\mathrm{PT}} \vdash [\![\tau]\!]_{\mathrm{P}}[S] : [\![\tau{:}\mathsf{T}]\!]_{\mathrm{PT}}[S]$$

As $\alpha \notin \mathsf{FTV}(\Delta; \Gamma)$ (can always α-vary to ensure this),

$$[\![\Delta]\!]_{F_\omega}, \Gamma, [\![\Delta]\!]_{\mathrm{PT}}, \mathtt{parse}_\alpha{:}[\![\mu\alpha.\tau{:}\mathsf{T}]\!]_{\mathrm{PT}} \vdash [\![\tau]\!]_{\mathrm{P}}[S] : [\![\tau{:}\mathsf{T}]\!]_{\mathrm{PT}}[S]$$

which is equivalent to:

$$[\![\Delta]\!]_{F_\omega}, \Gamma, [\![\Delta]\!]_{\text{PT}}, \texttt{parse}_\alpha{:}[\![\mu\alpha.\tau{:}\mathsf{T}]\!]_{\text{PT}} \vdash [\![\tau]\!]_{\text{p}}[S] : \texttt{bits} * \texttt{offset} \to \texttt{offset} * [\![\tau]\!]_{\text{rep}}[S] * [\![\tau]\!]_{\text{PD}}[S]$$

By typing (and expanding out $[S]$),

$$[\![\Delta]\!]_{F_\omega}, \Gamma, [\![\Delta]\!]_{\text{PT}} \vdash w : \texttt{offset} \tag{A.29}$$

$$[\![\Delta]\!]_{F_\omega}, \Gamma, [\![\Delta]\!]_{\text{PT}} \vdash r : [\![\tau]\!]_{\text{rep}}[[\![\mu\alpha.\tau]\!]_{\text{rep}}/\alpha_{\texttt{rep}}, [\![\mu\alpha.\tau]\!]_{\text{PDb}}/\alpha_{\texttt{PDb}}] \tag{A.30}$$

$$[\![\Delta]\!]_{F_\omega}, \Gamma, [\![\Delta]\!]_{\text{PT}} \vdash p : [\![\tau]\!]_{\text{PD}}[[\![\mu\alpha.\tau]\!]_{\text{rep}}/\alpha_{\texttt{rep}}, [\![\mu\alpha.\tau]\!]_{\text{PDb}}/\alpha_{\texttt{PDb}}]. \tag{A.31}$$

By Rep. Type Well-Formedness Lemma, $[\![\Delta]\!]_{\text{rep}}, \Gamma \vdash [\![\tau]\!]_{\text{rep}} :: \mathsf{T}$ and $[\![\Delta]\!]_{\text{PD}}, \Gamma \vdash [\![\tau]\!]_{\text{PD}} :: \mathsf{T}$.

So, $\alpha_{\texttt{PDb}} \notin \mathsf{FTV}([\![\tau]\!]_{\text{rep}})$, $\alpha_{\texttt{rep}} \notin \mathsf{FTV}([\![\tau]\!]_{\text{PD}})$ and, therefore,

$$[\![\Delta]\!]_{F_\omega}, \Gamma, [\![\Delta]\!]_{\text{PT}} \vdash r : [\![\tau]\!]_{\text{rep}}[[\![\mu\alpha.\tau]\!]_{\text{rep}}/\alpha_{\texttt{rep}}], \text{ and} \tag{A.32}$$

$$[\![\Delta]\!]_{F_\omega}, \Gamma, [\![\Delta]\!]_{\text{PT}} \vdash p : [\![\tau]\!]_{\text{PD}}[[\![\mu\alpha.\tau]\!]_{\text{PDb}}/\alpha_{\texttt{PDb}}]. \tag{A.33}$$

As $[\![\mu\alpha.\tau]\!]_{\text{rep}} = \mu\alpha_{\texttt{rep}}.[\![\tau]\!]_{\text{rep}}$,

$$[\![\Delta]\!]_{F_\omega}, \Gamma, [\![\Delta]\!]_{\text{PT}} \vdash \texttt{fold}[[\![\mu\alpha.\tau]\!]_{\text{rep}}]\, r : \mu\alpha_{\texttt{rep}}.[\![\tau]\!]_{\text{rep}}(= [\![\mu\alpha.\tau]\!]_{\text{rep}}).$$

As $[\![\mu\alpha.\tau]\!]_{\text{PDb}} = \mu\alpha_{\texttt{PDb}}.[\![\tau]\!]_{\text{PD}}$,

$$[\![\Delta]\!]_{F_\omega}, \Gamma, [\![\Delta]\!]_{\text{PT}} \vdash \texttt{fold}[[\![\mu\alpha.\tau]\!]_{\text{PDb}}]\, p : \mu\alpha_{\texttt{PDb}}.[\![\tau]\!]_{\text{PD}}(= [\![\mu\alpha.\tau]\!]_{\text{PDb}}).$$

Case TyApp:

Analagous to Rec, in that it relies on Lemma 9.

Proof: Lemma 33.

Case `unit`: $\texttt{unit} : \mathsf{T} \qquad [\![\texttt{unit}]\!]_{\text{p}}(B, \omega) \to_k (\omega', r, p)$

We wish to prove that $\mathrm{Canon}^*{}_{\mathtt{unit}}(r,p)$. That is, $\mathtt{unit} \rightarrow^* \nu$ and $\mathrm{Canon}_\nu(r,p)$. As `unit` is normal, it suffices to prove that $\mathrm{Canon}_{\mathtt{unit}}(r,p)$. By the definition of $[\![\mathtt{unit}]\!]_{\mathtt{p}}$, we have $r = \mathtt{R}_{\mathtt{unit}}()$ and $p = \mathtt{P}_{\mathtt{unit}}(\omega)$. By Lemma 31, then, we know that $\mathrm{Canon}_{\mathtt{unit}}(r,p)$.

Case $\tau_1\tau_2$: $\quad \tau_1\tau_2 : \mathsf{T} \qquad [\![\tau_1\ \tau_2]\!]_{\mathtt{p}}(B,\omega) \rightarrow_k (\omega', r, p)$

We wish to prove that $\mathrm{Canon}^*{}_{\tau_1\ \tau_2}(r,p)$. That is, $\tau_1\ \tau_2 \rightarrow^* \nu$ and $\mathrm{Canon}_\nu(r,p)$. We prove each clause in order. First, by Lemma 4, part 1, $[\![\tau_1\ \tau_2]\!]_{\mathtt{p}} \rightarrow_i v$, with $i < k$. By Lemma 4, part 1, we know that $i > 0$. By Lemma 17, $\tau_1\ \tau_2 \rightarrow^* \nu$ and $v \equiv [\![\nu]\!]_{\mathtt{p}}$. Now, we wish to prove the second clause by applying the induction hypothesis to the normal type ν. We therefore aim to show that $[\![\nu]\!]_{\mathtt{p}}(B,\omega) \rightarrow_j (\omega', r, p)$, for some $j < k$. Now, by Lemma 2, $[\![\tau_1\ \tau_2]\!]_{\mathtt{p}}(B,\omega) \rightarrow_i v\ (B,\omega)$, so, by Lemma 5, $v\ (B,\omega) \rightarrow_{(k-i)} (\omega', \mathtt{r}, \mathtt{p})$. Together with Lemma 7, part 1, we have $[\![\nu]\!]_{\mathtt{p}}(B,\omega) \rightarrow_{(k-i)} v_t$ and $v_t \equiv (\omega', r, p)$, which is nearly what we want. But, as (ω', r, p) contains no type annotations, $v_t = (\omega', \mathtt{r}, \mathtt{p})$, so, indeed, $[\![\nu]\!]_{\mathtt{p}}(B,\omega) \rightarrow_j (\omega', r, p)$, with $k - i < k$, as $i > 0$. By Lemma 12 we know that $\nu : \mathsf{T}$, so, by induction, we have $\mathrm{Canon}^*{}_\nu(r,p)$. As ν is normal, this result implies that $\mathrm{Canon}_\nu(r,p)$.

Case $\mu\alpha.\tau$: $\quad \mu\alpha.\tau : \mathsf{T} \qquad [\![\mu\alpha.\tau]\!]_{\mathtt{p}}(B,\omega) \rightarrow_k (\omega', r, p)$

By definition of $[\![\mu\alpha.\tau]\!]_{\mathtt{p}}$ and Evaluation Uniqueness (Lemma 5),

$$
\begin{aligned}
&[\![\mu\alpha.\tau]\!]_{\mathtt{p}}(B,\omega) \\
&\qquad \rightarrow \mathtt{let}\ (\omega', r_1, p_1) = [\![\tau]\!]_{\mathtt{p}}\{\mu\alpha.\tau/\alpha\}\ (\mathtt{B},\omega)\ \mathtt{in}\ (\ldots) \\
&\qquad\qquad \rightarrow_{(k-1)} (\omega', r, p)
\end{aligned}
$$

By Lemma 4, part 3, $[\![\tau]\!]_{\mathtt{p}}\{\mu\alpha.\tau/\alpha\}(B,\omega) \rightarrow_i v'$, with $i < k - 1$. As $\mu\alpha.\tau : \mathsf{T}$ and by DDC^α Inversion (Lemma 13), $\alpha{:}\mathsf{T}; \cdot \vdash \tau : \mathsf{T}$, Lemma 22 gives us $\mathrm{H}(\tau : \mathsf{T})$ and $\mathrm{H}(\mu\alpha.\tau : \mathsf{T})$. By Lemma 19, $\exists\,\sigma$ s.t. $[\![\tau]\!]_{\mathrm{PD}} = \sigma$ and by definition of H, $\exists\,\sigma$ s.t. $[\![\mu\alpha.\tau]\!]_{\mathrm{PD}} \equiv \mathtt{pd\_hdr} * \sigma$. So, by Commutativity of Substitution (Lemma 9), $[\![\tau]\!]_{\mathtt{p}}\{\mu\alpha.\tau/\alpha\} \equiv [\![\tau[\mu\alpha.\tau/\alpha]]\!]_{\mathtt{p}}$. Therefore, by Lemma 7, part 1, $[\![\tau[\mu\alpha.\tau/\alpha]]\!]_{\mathtt{p}}(B,\omega) \rightarrow_i v''$, with $i < k - 1$.

Next, we seek to establish the shape of v''. By DDC^{α} Substitution (Lemma 16), $\tau[\mu\alpha.\tau/\alpha] : \mathsf{T}$. By Type Correctness (Theorem 28),

$$[\![\tau[\mu\alpha.\tau/\alpha]]\!]_{\text{P}} : \texttt{bits} * \texttt{offset} \to \texttt{offset} * [\![\tau[\mu\alpha.\tau/\alpha]]\!]_{\text{rep}} * [\![\tau[\mu\alpha.\tau/\alpha]]\!]_{\text{PD}}.$$

By F_ω typing,

$$[\![\tau[\mu\alpha.\tau/\alpha]]\!]_{\text{P}}(B, \omega) : \texttt{offset} * [\![\tau[\mu\alpha.\tau/\alpha]]\!]_{\text{rep}} * [\![\tau[\mu\alpha.\tau/\alpha]]\!]_{\text{PD}}.$$

So, by F_ω preservation, $v'' : \texttt{offset} * [\![\tau[\mu\alpha.\tau/\alpha]]\!]_{\text{rep}} * [\![\tau[\mu\alpha.\tau/\alpha]]\!]_{\text{PD}}$. and by the F_ω canonical forms lemma, $v'' = (\omega_1, r_1, p_1)$.

Putting these conclusions together, we have $\tau[\mu\alpha.\tau/\alpha] : \mathsf{T}$ and $[\![\tau[\mu\alpha.\tau/\alpha]]\!]_{\text{P}}(B, \omega) \to_i (\omega_1, r_1, p_1)$, with $i < k - 1$. So, by induction, $\text{Canon}^*{}_{\tau[\mu\alpha.\tau/\alpha]}(r_1, p_1)$. By definition of $[\![\mu\alpha.\tau]\!]_{\text{P}}$, $r = \texttt{fold}[[\![\mu\alpha.\tau]\!]_{\text{rep}}]\, r_1$ and $p = (p_1.h, \texttt{fold}[[\![\mu\alpha.\tau]\!]_{\text{PDb}}]\, p_1)$. By definition of canonical forms, $\text{Canon}_{\mu\alpha.\tau}(r, p)$. As $\mu\alpha.\tau$ is normal, we have $\text{Canon}^*{}_{\mu\alpha.\tau}(r, p)$.

Appendix B

Complete PADS/ML Grammar

Below is a complete EBNF grammar of the syntax of PADS/ML. Terminals appear in the standard font, nonterminals are in *italics*, keywords appear in **`bold typewriter`** font, and concrete syntax appears in **`bold typewriter`** font, surrounded by quotes.

exprlit	::=	int | char | string | regexpr	// expression literals
hlexpr	::=	*exprlit* | **":"** *mlexpr* **":"**	// host language expression
compty	::=	*tp* | *hlexpr*	// compound type
pat	::=	int | char | string | "_" | id	// patterns
recomit	::=	*hlexpr* | *tp*	// omitted record fields
recfield	::=	*omitfd* | id **":"** *tp* | id **"="** *hlexpr*	// record fields
tas	::=	*tp* | **"("** *tp* (**","** *tp*)* **")"**	// type arguments
tp	::=	*tas*? tid *hlexpr*?	// type application
	|	*compty* **"*"** *compty* (**"*"** *compty*)*	// tuples
	|	"{" *recfield* (**";"** *recfield*)* **";"**? "}"	// records
	|	**"["** id **":"** *tp* **"|"** *mlexpr* **"]"**	// constraints
dtbranch	::=	uid **of** *hlexpr* | uid **of** *tp* | uid **of omit** *tp*	// datatype branch
dtdefault	::=	**with pdefault** uid? **of** *tp* (**"="** *hlexpr*)?	// datatype default
dt	::=	**"|"** ? *dtbranch* (**"|"** *dtbranch*)* *dtdefault*?	// datatype body
sdtbranch	::=	uid **of** *hlexpr* | uid **of** *tp* (**"="** *hlexpr*)?	// switched datatype branch
	|	uid **of omit** *tp*	
sdt	::=	**"|"** ? *pat* **"->"** *sdtbranch* (**"|"** *pat* **"->"** *sdtbranch*)*	// switched datatype body
tps	::=	tid | **"("** tid (**","** tid)* **")"**	// type parameters
decl	::=	**ptype** *tps*? tid (**"("** id **":"** *mlty* **")"**)? = *tp*?	// type declaration
	|	**pdatatype** *tps*? tid (**"("** id **":"** *mlty* **")"**)? **"="** *dt*	// datatype declaration
	|	**pdatatype** *tps*? tid **"("** id **":"** *mlty* **")"**? **"="**	// switched-datatype
		Pmatch *mlexpr* **with** *sdt*	// declaration
description	::=	(*decl*)+	

Appendix C

PADS/ML Runtime Interface

Below is a listing of the O'CAML interface of the `Pads` module, which contains the PADS/ML runtime system.

```
type pos = int64

type span = pos * pos

type error_code = Good | Maybe | Nest | Syn | Sem

type corrupted = string

type error_info = No_info | Error_span of span | Corrupted_data of corrupted

type pd_header = {
  nerr       : int;
  error_code : error_code;
  error_info : error_info;
  span       : span;
}

(* Abstract handle for PADS/ML state. *)
type handle

type 'a pd = pd_header * 'a

type ('a,'b) parser = handle -> 'a * ('b pd)
type ('a,'b) printer = 'a -> ('b pd) -> handle -> unit

type base_pd_body = unit
type base_pd = base_pd_body pd

exception Runtime_error

val get_pd_hdr : 'a * 'b -> 'a

(* Check whether a pd describes an error-free parse. *)
val pd_is_ok : 'a pd -> bool
```

```
(*
 * Scan for char literal. If the literal is found, returns
 * the number of bytes skipped.
 *)
val p_char_lit_scan1 : handle -> char -> int option

val p_str_lit_scan1 : handle -> string -> int option

val p_int_lit_scan1 : handle -> int -> int option

(* Get the current position in the data source. *)
val get_current_pos : handle -> pos

(* Compare two positions. Compatible with Pervasives.compare*)
val comp_pos: pos -> pos -> int

(* Compare two positions for equality. *)
val eq_pos: pos -> pos -> bool

(* Create span with identical start and end positions. *)
val make_empty_span : handle -> span

(* Create a valid pd header with an empty span. *)
val make_empty_pd : handle -> base_pd

(* Create a valid pd header given a span. *)
val mk_valid_pd_hdr : span -> pd_header

(* Valid pd header with span set to (0,0). *)
val spanless_pd_hdr : pd_header

(* An initialized base pd for use with base-type gen_pd functions. *)
val gen_base_pd : base_pd

(* Initialize the system, relying on defaults. *)
val open_handle : unit -> handle option

(*
 * Initialize the system, specifying that the data source does
 * not use records.
 *)
val open_handle_norec : unit -> handle option

(* Cleanup the system. *)
val close_handle : handle -> unit option
```

```
module IO : sig
  (* Open a file for IO. The second argument is the file name. *)
  val open_file : handle -> string -> unit option

  (* Close the IO system for this pads handle. *)
  val close : handle -> unit option

  (*
   * Check whether current parse is speculative. Speculative parses are
   * used by datatypes to try out the different variants.
   *)
  val is_speculative : handle -> bool

  (* Raise when an error is encounterd during a speculative parse. *)
  exception Speculation_failure
end

(*************************************************************
 * The remaining modules are intended only for generated tools.
 *************************************************************)

module Record : sig
  (* Update an existing hdr with the pd from a subcomponent. *)
  val update_pd_hdr : pd_header -> pd_header -> pd_header

  val parse_first : ('a,'b) parser -> handle -> 'a * ('b pd) * pd_header
  val parse_next : ('a,'b) parser -> pd_header -> handle
                   -> 'a * ('b pd) * pd_header
  val absorb_first : ('a,'b) parser -> handle -> pd_header
  val absorb_next : ('a,'b) parser -> pd_header -> handle -> pd_header

  val absorb_first_char : char -> handle -> pd_header
  val absorb_next_char  : char -> pd_header -> handle -> pd_header

  val absorb_first_string :  string -> handle -> pd_header
  val absorb_next_string  :  string -> pd_header -> handle -> pd_header

  val absorb_first_int : int -> handle -> pd_header
  val absorb_next_int  : int -> pd_header -> handle -> pd_header

  (*
   * Generate a valid parse-descriptor header for a record based
   * on a previous header and the parse descriptor of a record
   * element.
   * For convenience, returns (unmodified) element parse descriptor
   * in addition to the new header.
   *)
  val gen_pd : pd_header -> 'a pd -> 'a pd * pd_header
end
```

```
module Compute : sig
  (* Make a parsing function from a computed value and gen_pd function. *)
  val generate_parser : 'a -> ('a -> 'b pd) -> ('a,'b) parser
end

module Where : sig
  val gen_pd : 'a pd -> bool -> ('a pd) pd
  val make_pd : 'a pd -> bool -> handle -> ('a pd) pd
  val parse_underlying : ('a,'b) parser -> ('a -> bool)
                         -> ('a,('b pd)) parser
end

module Datatype : sig
  val parse_variant  : ('a,'b) parser -> handle -> ('a * ('b pd)) option
  val absorb_variant : ('a,'b) parser -> handle -> span option
  val absorb_char_variant   : char   -> handle -> span option
  val absorb_string_variant : string -> handle -> span option
  val absorb_int_variant    : int    -> handle -> span option

  val parse_case  : ('a,'b) parser -> ('a -> 'c) -> ('b pd -> 'd)
                    -> ('c, 'd) parser
  val absorb_case : ('a,'b) parser -> 'c -> 'd -> ('c, 'd) parser
  val absorb_char_case   : char -> 'a -> 'b -> ('a, 'b) parser
  val absorb_string_case : string -> 'a -> 'b -> ('a, 'b) parser
  val absorb_int_case    : int -> 'a -> 'b -> ('a, 'b) parser

  (* args: gen_rep genpd_fn rep_constructor pd_constructor *)
  val gen_case : 'a -> ('a -> 'b pd) -> ('a -> 'c) -> ('b pd -> 'd)
                 -> ('c,'d) parser

  val make_pd_hdr : pd_header -> pd_header
  val make_rep : 'a -> 'a
  val make_pd : pd_header * 'a -> 'a pd

  val make_err_pd : handle -> 'a -> 'a pd
  (* If speculative, raise exception. Otherwise, return pd. *)
  val handle_error_variant : handle -> 'a -> 'a pd

  val make_gen_pd : handle -> 'a -> 'a pd
  val make_absorb_pd : span -> 'a -> 'a pd

  (*
   * Generate a valid parse descriptor for a datatype given
   * the parse descriptor of the variant and a function to generate
   * the datatype's pd body from the variant's pd.
   *)
  val gen_pd : 'a pd -> ('a pd -> 'b) -> 'b pd
  (* Generate the pd when the variant has no subcomponent. *)
  val gen_pd_empty : 'b -> 'b pd
end
```

Appendix D

Generic-Tool Interface

Below is a listing of the O'CAML interface of the `Generic_tool` module, which contains `S`, the signature of generic tools.

```
(* Common signature of generic tool base type modules. *)
module type BaseType = sig

  (* type of value of this base type. *)
  type t

  (* type of tool state for this base type. *)
  type state

  (* Generate initial state for value of type t. *)
  val init   : unit -> state

  (* Process a value of type t. *)
  val process  : state -> t option -> Pads.pd_header -> state
end

(* Interface with which generic tools must comply. *)
module type S = sig

  (* Data structure built by generic tool during data traversal. *)
  type state

  (* Raise to indicate error condition in the execution of a tool *)
  exception Tool_error of state * string

  (* Initialize the tool. *)
  val init : unit -> unit

  module Int    : BaseType with type t = int    and type state = state
  module Char   : BaseType with type t = char   and type state = state
  module String : BaseType with type t = string and type state = state
  module Unit   : BaseType with type t = unit   and type state = state
```

```
(* Functions for tuples and records. *)
  module Record : sig
    (* Intermediate data used for record processing *)
    type partial_state

    (*
     * Generate initial state of record given states of all
     * components. Component state is labeled with the field name.
     *)
    val init  : (string * state) list -> state

    (* Begin processing the record. *)
    val start : state -> Pads.pd_header -> partial_state

    (* Retrieve state of named field, given state of record. *)
    val project : state -> string -> state

    (* Process named field, given record state and field state. *)
    val process_field : partial_state -> string -> state -> partial_state

    (* Finish processing record. *)
    val finish : partial_state -> state
  end

  module Constraint : sig
    (* Intermediate data used for constraint processing. *)
    type partial_state

    (* Generate initial state from state of constrained element. *)
    val init : state -> state

    (* Start  processing a constraint. *)
    val start : state -> Pads.pd_header -> partial_state

    (* Retrieve the state of constrained element. *)
    val project : state -> state

    (* Process constraint given constraint state and element state. *)
    val process : partial_state -> state -> state
  end
```

```
  module Datatype : sig
    (* Intermediate data used for datatype processing. *)
    type partial_state

    (*
     * Generate initial state of datatype without any state for the
     * branches. Branch state will be added dynamically.
     *)
    val init : unit -> state

    (* Start processing datatype *)
    val start : state -> Pads.pd_header -> partial_state

    (*
     * Retrieve state of named variant. Returs None if no state
     * exists for that variant.
     *)
    val project : state -> string -> state option

    (* Process named variant, given datatype state and variant state. *)
    val process_variant : partial_state -> string -> state -> state

    (* For variants that have no contents. *)
    module Empty : sig
      (* Generate initial state for empty variant. *)
      val init   : unit -> state

      (* Process an empty variant. *)
      val process  : state -> state
    end
  end
end
```

Appendix E

Generic XML Conversion Tool

```
(*********************************************************************
 * XML formatter tool: formats arbitrary data representation as XML.
 * This tool uses the MotionTwin XML-Light library.
 *********************************************************************)

(* State: Collects data values into an Xml.xml structure *)
type state = Xml.xml list
type global_state = state

(*
 * Wrap up top-level list of XML elements into single xml element.
 * This function is specific to the XML formatter and is not specified
 * in the generic tool interface.
 *)
let wrap elements name = Xml.Element(name,[],elements)

exception Tool_error of state * string

(* No initialization needed for this tool. *)
let init () = ()

(* Convert an error code to a string representation. *)
let ec_to_string (ec : Pads.error_code) =
  match ec with
    Pads.Good -> "GOOD"
  | Pads.Maybe -> "MAYBE"
  | Pads.Nest -> "NEST"
  | Pads.Syn -> "SYN"
  | Pads.Sem -> "SEM"

(* Convert a parse descriptor header to an XML representation. *)
let hdr_to_xml (h : Pads.pd_header) =
  Xml.Element ("pd", [],
    [Xml.Element("nerr",[],
        [Xml.PCData (string_of_int h.Pads.nerr)]);
     Xml.Element("error_code",[],
        [Xml.PCData (ec_to_string h.Pads.error_code)])])
```

```
(* Decide whether or not to include the PD header in the XML.*)
let process_hdr pd_hdr =
  match pd_hdr.Pads.error_code with
      Pads.Good -> []
    | _         -> [hdr_to_xml pd_hdr]

let process_base result base_to_string pd_hdr =
  match result with
      Pads.Ok r -> [Xml.Element ("val", [], [Xml.PCData(base_to_string r)])]
    | Pads.Error -> [hdr_to_xml pd_hdr]

module Int = struct
  type t = int
  type state = global_state
  let init _ = []
  let process _ result pd_hdr = process_base result string_of_int pd_hdr
end

module Char = struct
  type t = char
  type state = global_state
  let init _ = []
  let process _ result pd_hdr = process_base result (String.make 1) pd_hdr
end

module String = struct
  type t = string
  type state = global_state
  let init _ = []
  let process _ result pd_hdr = process_base result (fun s -> s) pd_hdr
end

module Unit = struct
  type t = unit
  type state = global_state
  let init () = []
  let process _ result pd_hdr = process_base result (fun () -> "") pd_hdr
end
```

```
module Record = struct
  type partial_state = state

  let init named_states = []

  let start state pd_hdr = process_hdr pd_hdr

  (* Project is a no-op becuase this tool ignores previous state.*)
  let project state field_name = []

  (* Build up list in **reverse** order *)
  let process_field fields field_name state =
     (Xml.Element (field_name, [], state))::fields

  let finish state = List.rev state
end

module Datatype = struct
  type partial_state = state

  let init () = []

  let start state pd_hdr = process_hdr pd_hdr

  let project state variant_name = None

  let process_variant state variant_name variant =
                state @ [Xml.Element (variant_name, [], variant)]

  module Empty = struct
    let init () = []
    let process state = state
  end
end

module Constraint = struct
  type partial_state = state

  let init _ = []

  let start _ pd_hdr = process_hdr pd_hdr

  let project state = []

  let process state sub_state = state @ sub_state
end
```

Bibliography

[Agn98] Étienne Agnon. SableCC: An object oriented compiler framework. Master's thesis, School of Computer Science, McGill University, Montreal, 1998.

[App98] Andrew W. Appel. *Modern Compiler Implementation in ML*. Cambridge University Press, New York, NY, USA, 1998.

[Aut05] Options Price Reporting Authority. Data recipient interface specification, version 1.6. http://www.opradata.com, December 2005.

[Bac02] Godmar 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.

[BMS05] Claus Brabrand, Anders Møller, and Michael I. Schwartzbach. Dual syntax for XML languages. In *Tenth International Symposium on Database Programming Languages*, volume 3774 of *Lecture Notes in Computer Science*, pages 27–41. Springer-Verlag, August 2005.

[BU73] Alexander Birman and Jeffrey D. Ullman. Parsing algorithms with backtrack. *Information and Control*, 23(1), August 1973.

[Bur75] William Burge. *Recursive Programming Techniques*. Addison Wesley, 1975.

[BW04] Mike Beckerle and Martin Westhead. GGF DFDL primer. `http://www.ggf.org/Meetings/GGF11/Documents/DFDL_Primer_v2.pdf`, May 2004. Global Grid Forum.

[CFP+04] Corinna Cortes, Kathleen Fisher, Daryl Pregibon, Anne Rogers, and Frederick Smith. Hancock: A language for analyzing transactional data streams. *ACM Trans. Program. Lang. Syst.*, 26(2):301–338, 2004.

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

[Dat05] Data format description language (DFDL) a Proposal, Working Draft, Global Grid Forum, Aug 2005. `https://forge.gridforum.org/projects/dfdl-wg/document/DFDL_Proposal/en/%2`.

[DFF+06a] Mark Daly, Mary Fernández, Kathleen Fisher, Robert Gruber, Yitzhak Mandelbaum, David Walker, and Xuan Zheng. PADS: An end-to-end system for processing ad hoc data. In *Demo paper for the 2006 ACM SIGMOD International Conference on Management of Data*, June 2006.

[DFF+06b] Mark Daly, Mary Fernández, Kathleen Fisher, Yitzhak Mandelbaum, and David Walker. LaunchPADS: A system for processing ad hoc data. In *Demo paper for the Workshop on Programming Language Technologies for XML*, January 2006.

[Dre05] Derek Dreyer. *Understanding and Evolving the ML Module System*. PhD thesis, CMU, May 2005. CMU-CS-05-131.

[Dub01] Olivier Dubuisson. *ASN.1: Communication between heterogeneous systems*. Morgan Kaufmann, 2001.

[FFGM06] Mary F. Fernández, Kathleen Fisher, Robert Gruber, and Yitzhak Mandelbaum. PADX: Querying large-scale ad hoc data with XQuery. In *Programming Language Technologies for XML*, January 2006.

[FG05] Kathleen Fisher and Robert Gruber. PADS: A domain specific language for processing ad hoc data. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 295–304. ACM Press, June 2005.

[FMW06] Kathleen Fisher, Yitzhak Mandelbaum, and David Walker. The next 700 data description languages. In *ACM Symposium on Principles of Programming Languages*, pages 2 – 15, January 2006.

[For02] Bryan Ford. Packrat parsing:: simple, powerful, lazy, linear time. In *ACM International Conference on Functional Programming*, pages 36–47. ACM Press, October 2002.

[For04] Bryan Ford. Parsing expression grammars: a recognition-based syntactic foundation. In *ACM Symposium on Principles of Programming Languages*, pages 111–122. ACM Press, January 2004.

[Gri06] Robert Grimm. Better extensibility through modular syntax. In *ACM Conference on Programming Language Design and Implementation*, pages 38–51, New York, NY, USA, 2006. ACM Press.

[Hen93] Fritz Henglein. Type inference with polymorphic recursion. *ACM Transactions on Progamming Languages and Systems*, 15(2):253–289, 1993.

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

[Hin00] Ralf Hinze. A new approach to generic functional programming. In *ACM Symposium on Principles of Programming Languages*, pages 119–132, January 2000.

[HJ03] Ralf Hinze and Johan Jeuring. Generic Haskell: Practice and theory. Technical Report UU-CS-2003-015, Institute of Information and Computing Sciences, Utrecht University, 2003.

[HM98] Graham Hutton and Erik Meijer. Monadic Parsing in Haskell. *Journal of Functional Programming*, 8(4):437–444, July 1998.

[JJ96] J. Jeuring and P. Jansson. Polytypic programming. In *Second International School on Advanced Functional Programming*, volume 1129 of *Lecture Notes in Computer Science*, pages 68–114, August 1996.

[KR01] Balachander Krishnamurthy and Jennifer Rexford. *Web Protocols and Practice*. Addison Wesley, 2001.

[Lan66] P. J. Landin. The next 700 programming languages. *Communications of the ACM*, 9(3):157 – 166, March 1966.

[Lie88] Karl Lieberherr. Object-oriented programming with class dictionaries. *Lisp and Symbolic Computation*, 1:185–212, 1988.

[LJW+06] Qin Lv, William Josephson, Zhe Wang, Moses Charikar, and Kai Li. Ferret: A toolkit for content-based similarity search of feature-rich data. In *EuroSys2006*, April 2006.

[LP03] Ralf Lämmel and Simon Peyton Jones. Scrap your boilerplate: a practical design pattern for generic programming. In *ACM Workshop on Types in Language Design and Implementation*, pages 26–37. ACM Press, March 2003.

[MC00a] Peter McCann and Satish Chandra. PacketTypes: Abstract specification of network protocol messages. In *ACM Conference of Special Interest Group on Data Communications*, pages 321–333. ACM Press, August 2000.

[MC00b] J. Myers and A. Chappell. Binary format definition (BFD). `http://collaboratory.emsl.pnl.gov/sam/bfd/`, 2000.

[MET03] METS: An overview and tutorial. `http://www.loc.gov/standards/mets/METSOverview.v2.html`, 2003.

[MFW+06] Yitzhak Mandelbaum, Kathleen Fisher, David Walker, Mary Fernández, and Artem Gleyzer. PADS/ML: A functional data description language. Technical Report TR-761-06, Princeton University, July 2006.

[MHS+05] R. Mandelbaum, C. M. Hirata, U. Seljak, J. Guzik, N. Padmanabhan, C. Blake, M. R. Blanton, R. Lupton, and J. Brinkmann. Systematic errors in weak lensing: application to SDSS galaxy-galaxy weak lensing. *Monthly Notices of the Royal Astronomical Society*, 361:1287–1322, August 2005.

[MZF+05] Luc Moreau, Yong Zhao, Ian Foster, Jens Voeckler, and Micael Wilde. XDTM: The XML data type and mapping for specifying datasets. In *European Grid Conference*, 2005.

[Net] Cisco NetFlow. `http://www.cisco.com/warp/public/732/Tech/nmp/netflow/index.shtml`.

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

[Newb] Tree formats. Workshop on Molecular Evolution web site. `http://workshop.molecularevolution.org/resources/fileformats/tree_forma%ts.php`.

[Oh06] Jin Oh. PADS and CASS utilization for beta coefficient estimation with the single-index model. Princeton University Undergraduate Senior Independent Work, May 2006.

[Pad] PADS manual. `http://www.padsproj.org`.

[Pax99] Vern Paxson. A system for detecting network intruders in real-time. In *Computer Networks*, December 1999.

[Pie02] Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, February 2002.

[PQ95] T. J. Parr and R. W. Quong. ANTLR: A predicated- *ll(k)* parser generator. *Software – practice and experience*, 25(7):789–810, July 1995.

[vWSP05] Arjen van Weelden, Sjaak Smetsers, and Rinus Plasmeijer. Polytypic syntax tree operations. In *Implementation and Application of Functional Languages, 17th International Workshop, IFL 2005, Revised Selected Papers*, volume 4015 of *Lecture Notes in Computer Science*, Dublin, Ireland, September 2005. Springer.

[WAKS97] Daniel C. Wang, Andrew W. Appel, Jeff L. Korn, and Christopher S. Serra. The Zephyr abstract syntax description language. In *USENIX Conference on Domain-Specific Languages*, October 1997. `http://ncstrl.cs.princeton.edu/expand.php?id=TR-554-97`.

[ZDF+05] Yong Zhao, Jed Dobson, Ian Foster, Luc Moreau, and Micael Wilde. A notation and system for expressing and executing cleanly typed workflows on messy scientific data. *ACM SIGMOD Record*, 34(3):37–43, 2005.