datashape Documentation

Release 0.5.4+8.gcae16a8

Continuum Analytics

Jul 25, 2018

Contents

Contents:

CHAPTER 1

Datashape Overview

Datashape is a data layout language for array programming. It is designed to describe in-situ structured data without requiring transformation into a canonical form.

Similar to NumPy, datashape includes `shape` and `dtype`, but combined together in the type system.

1.1 Units

Single named types in datashape are called `unit` types. They represent either a dtype like `int32` or `datetime`, or a single dimension like `var`. Dimensions and a single dtype are composed together in a datashape type.

1.1.1 Primitive Types

DataShape includes a variety of dtypes corresponding to C/C++ types, similar to NumPy.

Bit type	Description
bool	Boolean (True or False) stored as a byte
int8	Byte (-128 to 127)
int16	Two's Complement Integer (-32768 to 32767)
int32	Two's Complement Integer (-2147483648 to 2147483647)
int64	Two's Complement Integer (-9223372036854775808 to 9223372036854775807)
uint8	Unsigned integer (0 to 255)
uint16	Unsigned integer (0 to 65535)
uint32	Unsigned integer (0 to 4294967295)
uint64	Unsigned integer (0 to 18446744073709551615)
float16	Half precision float: sign bit, 5 bits exponent, 10 bits mantissa
float32	Single precision float: sign bit, 8 bits exponent, 23 bits mantissa
float64	Double precision float: sign bit, 11 bits exponent, 52 bits mantissa
complex[float32]	Complex number, represented by two 32-bit floats (real and imaginary components)
complex[float64]	Complex number, represented by two 64-bit floats (real and imaginary components)

Additionally, there are types which are not fully specified at the bit/byte level.

Bit type	Description
string	Variable length Unicode string.
bytes	Variable length array of bytes.
json	Variable length Unicode string which contains JSON.
date	Date in the proleptic Gregorian calendar.
time	Time not attached to a date.
datetime	Point in time, combination of date and time.
units	Associates physical units with numerical values.

Many python types can be mapped to datashape types:

Python type	Datashape
int	int32
bool	bool
float	float64
complex	complex[float64]
str	string
unicode	string
datetime.date	date
datetime.time	time
datetime.datetime	datetime or datetime[tz='<timezone>']
datetime.timedelta	units['microsecond', int64]
bytes	bytes
bytearray	bytes
buffer	bytes

1.1.2 String Types

To Blaze, all strings are sequences of unicode code points, following in the footsteps of Python 3. The default Blaze string atom, simply called "string", is a variable-length string which can contain any unicode values. There is also a fixed-size variant compatible with NumPy's strings, like `string[16, "ascii"]`.

1.2 Dimensions

An asterisk (*) between two types signifies an array. A datashape consists of 0 or more `dimensions` followed by a `dtype`.

For example, an integer array of size three is:

```
3 * int
```

In this type, 3 is is a `fixed` dimension, which means it is a dimension whose size is always as given. Other dimension types include `var`.

Comparing with NumPy, the array created by `np.empty((2, 3), 'int32')` has datashape `2 * 3 * int32`.

1.2.1 Records

Record types are ordered struct dtypes which hold a collection of types keyed by labels. Records look similar to Python dictionaries but the order the names appear is important.

Example 1:

```
{
    name   : string,
    age    : int,
    height : int,
    weight : int
}
```

Example 2:

```
{
    r: int8,
    g: int8,
    b: int8,
    a: int8
}
```

Records are themselves types declaration so they can be nested, but cannot be self-referential:

Example 2:

```
{
    a: { x: int, y: int },
    b: { x: int, z: int }
}
```

1.2.2 Datashape Traits

While datashape is a very general type system, there are a number of patterns a datashape might fit in.

Tabular datashapes have just one dimension, typically `fixed` or `var`, followed by a record containing only simple types, not nested records. This can be intuitively thought of as data which will fit in a SQL table.:

```
var * { x : int, y : real, z : date }
```

Homogenous datashapes are arrays that have a simple dtype, the kind of data typically used in numeric computations. For example, a 3D velocity field might look like:

```
100 * 100 * 100 * 3 * real
```

1.2.3 Type Variables

Type variables are a separate class of types that express free variables scoped within type signatures. Holding type variables as first order terms in the signatures encodes the fact that a term can be used in many concrete contexts with different concrete types.

For example the type capable of expressing all square two dimensional matrices could be written as a datashape with type variable `A`, constraining the two dimensions to be the same:

```
A * A * int32
```

A type capable of rectangular variable length arrays of integers can be written as two free type vars:

```
A * B * int32
```

Note: Any name beginning with an uppercase letter is parsed as a symbolic type (as opposed to concrete). Symbolic types can be used both as dimensions and as data types.

1.2.4 Option

An option type represents data which may be there or not. This is like data with `NA` values in R, or nullable columns in SQL. Given a type like `int`, it can be transformed by prefixing it with a question mark as `?int`, or equivalently using the type constructor `option[int]`

For example a `5 * ?int` array can model the Python data:

```
[1, 2, 3, None, None, 4]
```

CHAPTER 2

DataShape Types

In addition to *defining the grammar*, datashape specifies a standard set of types and some properties those types should have. Type constructors can be classified as `dimension` or `dtype`, and a datashape is always composed of zero or more dimensions followed by a dtype.

2.1 Dimension Types

2.1.1 Fixed Dimension

`fixed[4]`

A dimension whose size is specified. This is the most common dimension type used in Blaze, and `4 * int32` is syntactic sugar for `fixed[4] * int32` in datashape syntax.

2.1.2 Var Dimension

`var`

A dimension whose size may be different across instances. A common use of this is a ragged array like `4 * var * int32`.

2.1.3 Type Variables

`typevar['DimName']`

Constructs a type variable. `DimName` is syntactic sugar for `typevar['DimName']`. This is used for pattern matching types, particularly for function prototypes. For example the datashape `(M * N * int32) -> N * int32` accepts an input with two dimensions that are type variables, and returns a one dimensional array using one of those dimension types.

2.1.4 Ellipsis

`ellipsis`

Constructs an ellipsis for matching multiple broadcast dimensions. `...` is syntactic sugar for `ellipsis`.

`ellipsis['DimVar']`

Constructs a named ellipsis for matching multiple broadcast dimensions. `Dim...` is syntactic sugar for `ellipsis['Dim']`.

2.2 DTypes

2.2.1 Boolean Type

`bool`

A boolean type which may take on two values, `True` and `False`. In Blaze and DyND, this is stored as a single byte which may take on the values 1 and 0.

2.2.2 Default Integer

`int`

This is an alias for `int32`.

2.2.3 Arbitrary-Precision Integer

`bignum` or `bigint`

An integer type which has no minimum or maximum value. This is not implemented in Blaze or DyND presently and the final name for it hasn't been locked down.

2.2.4 Signed Integer Types

`int8` `int16` `int32` `int64` `int128`

Integer types whose behavior follows that of twos-complement integers of the given size.

2.2.5 Unsigned Integer Types

`uint8` `uint16` `uint32` `uint64` `uint128`

Integer types whose behavior follows that of unsigned integers of the given size.

2.2.6 Platform-Specific Integer Aliases

`intptr` `uintptr`

Aliases for `int##` and `uint##` where ## is the size of a pointer type on the platform.

2.2.7 Default Floating Point

`real`

This is an alias for `float64`.

2.2.8 Binary Floating Point

`float16` `float32` `float64` `float128`

Binary floating point types as defined by IEEE 754-2008. Each type corresponds to the `binary##` type defined in the standard.

Note that `float128` is not a C/C++ `long double`, except on such platforms where they coincide. NumPy defines a `float128` on some platforms which is not IEEE `binary128`, and is thus different from DataShape's type of the same name on those platforms.

TODO: Support for C/C++ `long double`. This is tricky given that DataShape intends to be cross-platform, and maybe some inspiration can be taken from HDF5 for specifying them.

2.2.9 Decimal Floating Point

`decimal32` `decimal64` `decimal128`

Decimal floating point types as defined by IEEE 754-2008. These are not implemented in Blaze or DyND presently.

2.2.10 Default Complex

`complex`

This is an alias for `complex[float64]`.

2.2.11 Complex

`complex[float32]`

Constructs a complex number type from a real number type.

2.2.12 Void

`void`

A type which can store no data. It is not intended to be constructed in concrete arrays, but to allow for things like function prototypes with `void` return type.

2.2.13 String

`string`

A unicode string that can be arbitrarily sized. In Blaze and DyND, this is a UTF-8 encoded string.

`string[16]`

A unicode string in a UTF-8 fixed-sized buffer. The string is zero-terminated, but as in NumPy, all bytes may be filled with character data so the buffer is not valid as a C-style string.

`string['utf16']`

A unicode string that can be arbitrarily sized, using the specified encoding. Valid values for the encoding are `'ascii'`, `'utf8'`, `'utf16'`, `'utf32'`, `'ucs2'`, and `'cp###'` for valid code pages.

`string[16, 'utf16']`

A unicode string in a fixed-size buffer of the specified number of bytes, encoded as the requested encoding. The string is zero-terminated, but as in NumPy, all bytes may be filled with character data so the buffer is not valid as a C-style string.

2.2.14 Character

`char`

A value which contains a single unicode code point. Typically stored as a 32-bit integer.

2.2.15 Bytes

`bytes`

An arbitrarily sized blob of bytes. This like `bytes` in Python 3.

`bytes[16]`

A fixed-size blob of bytes. This is not zero-terminated as in the `string` case, it is always exactly the specified number of bytes.

2.2.16 Categorical

`categorical[['low', 'medium', 'high'], type=string, ordered=True]`

Constructs a type whose values are constrained to a particular set. The `type` parameter is optional and is inferred by the first argument. The `ordered` parameter is a boolean indicating whether the values in the set are ordered, so certain functions like min and max work.

Note: The categorical type *assumes* that the input categories are unique.

2.2.17 JSON

`json`

A unicode string which is known to contain values represented as JSON.

2.2.18 Records

`struct[['name', 'age', 'height'], [string, int, real]]`

Constructs a record type with the given field names and types. `{name: string, age: int}` is syntactic sugar for `struct[['name', 'age'], [string, int]]`.

2.2.19 Tuples

`tuple[[string, int, real]]`

Constructs a tuple type with the given types. `(string, int)` is syntactic sugar for `tuple[[string, int]]`.

2.2.20 Function Prototype

`funcproto[[string, int], bool]`

Constructs a function prototype with the given argument and return types. `(string, int) -> bool` is syntactic sugar for `funcproto[[string, int], bool]`.

2.2.21 Type Variables

`typevar['DTypeName']`

Constructs a type variable. `DTypeName` is syntactic sugar for `typevar['DTypeName']`. This is used for pattern matching types, particularly for function prototypes. For example the datashape `(T, T) -> T` accepts any types as input, but requires they have the same types.

2.2.22 Option/Missing Data

`option[float32]`

Constructs a type based on the provided type which may have missing values. `?float32` is syntactic sugar for `option[float32]`.

The type inside the option parameter may also have its own dimensions, for example `?3 * float32` is syntactic sugar for `option[3 * float32]`.

2.2.23 Pointer

```
pointer[target=2 * 3 * int32]
```

Constructs a type whose value is a pointer to values of the target type.

2.2.24 Maps

Represents the type of key-value pairs. This is used for discovering foreign key relationships in relational databases, but is meant to be useful outside of that context as well. For example the type of a column of Python dictionaries whose keys are strings and values are 64-bit integers would be written as:

```
var * map[string, int64]
```

2.2.25 Date, Time, and DateTime

`date`

A type which represents a single date in the Gregorian calendar. In DyND and Blaze, it is represented as a 32-bit signed integer offset from the date `1970-01-01`.

`time` `time[tz='UTC']`

Represents a time in an abstract day (no time zone), or a day with the specified time zone.

Stored as a 64-bit integer offset from midnight, stored as ticks (100 ns units).

`datetime` `datetime[tz='UTC']`

Represents a moment in time in an abstract time zone if no time zone is provided, otherwise stored as UTC but representing time in the specified time zone.

Stored as a 64-bit signed integer offset from `0001-01-01T00:00:00` in ticks (100 ns units), the "universal time scale" from the ICU library. Follows the POSIX convention of ignoring leap seconds.

http://userguide.icu-project.org/datetime/universaltimescale

`units['second', int64]`

A type which represents a value with the units and type specified. Initially only supporting time units, to support the datetime functionality without adding a special "timedelta" type.

Initial valid units are: '100*nanosecond' (ticks as in the datetime storage), 'microsecond', 'millisecond', 'second', 'minute', 'hour', 'day'. Need to decide on valid shortcuts in a context with more physical units, probably by adopting conventions from a good physical units library.

`timetz` `datetimetz`

Represents a time/datetime with the time zone attached to the data. Not implemented in Blaze/DyND.

CHAPTER 3

Pattern Matching DataShapes

DataShape includes type variables, as symbols beginning with a capital letter. For example *A * int32* represents a one-dimensional array of *int32*, where the size or type of the dimension is unspecified. Similarly, *3 * A* represents a size 3 one-dimensional array where the data type is unspecified.

The main usage of pattern matching in the DataShape system is for specifying function signatures. To provide a little bit of motivation, let's examine what happens in NumPy ufuncs, and see how we can represent their behaviors via DataShape types.

3.1 NumPy *ldexp* UFunc Example

We're going to use the *ldexp* ufunc, which is for the C/C++ function with overloads *double ldexp(double x, int exp)* and *float ldexp(float x, int exp)*, computing *x * 2^exp* by tweaking the exponent in the floating point format. (We're ignoring the long double for now.)

These C++ functions can be represented with the DataShape function signatures:

```
(float32, int32) -> float32
(float64, int32) -> float64
```

As a NumPy ufunc, there is an behavior for arrays, where the source arrays are "broadcasted" together, and the function is computed elementwise.

In the simplest case, given two arrays which match, the result is an array of the same size. When one array has size one in a dimension, it gets repeated to match the size of the other dimension. When one array has fewer dimensions, it gets repeated to fill in the outer dimensions. The "broadcast" array shape is the result of combining all these repetitions, and is the shape of the output. Represented as DataShape function signatures, some examples are:

```
(12 * float32, 12 * int32) -> 12 * float32
(10 * float64, 1 * int32) -> 10 * float64
(float32, 3 * 4 * int32) -> 3 * 4 * float32
(3 * float64, 4 * 1 * int64) -> 4 * 3 * float64
```

3.2 Ellipsis for Broadcasting

To represent the general broadcasting behavior, DataShape provides ellipsis type variables.:

```
(A... * float32, A... * int32) -> A... * float32
(A... * float64, A... * int64) -> A... * float64
```

3.2.1 Coercions/Broadcasting as a System of Equations

Let's say as input we get two arrays with datashapes *3 * 4 * float64* and *int32*. We can express this as two systems of coercion equations as follows (using ==> as a "coerces to" operator):

```
# float32 prototype
3 * 4 * float64 ==> A... * float32
int32 ==> A... * int32

# float64 prototype
3 * 4 * float64 ==> A... * float64
int32 ==> A... * int32
```

To solve these equations, we evaluate the legality of each coercion, and accumulate the set of values the *A. . .* type variable must take.:

```
# float32 prototype
float64 ==> float32 # ILLEGAL
3 * 4 * ==> A... *  # "3 * 4 *" in A...
int32 ==> int32     # LEGAL
* ==> A...          # "*" in A...

# float64 prototype
float64 ==> float64 # LEGAL
3 * 4 * ==> A... *  # "3 * 4 *" in A...
int32 ==> int32     # LEGAL
* ==> A...          # "*" in A...
```

The float32 prototype can be discarded because it requires an illegal coercion. In the float64 prototype, we collect the set of all *A. . .* values *{"3 * 4 *", "*"}*, broadcast them together to get *"3 * 4 *"*, and substitute this in the output. Doing all the substitutions in the full prototype produces:

```
(3 * 4 * float64, int32) -> 3 * 4 * float64
```

as the matched function prototype that results.

3.3 Disallowing Coercion

In the particular function we picked, ideally we don't want to allow implicit coercion of the type, because the nature of the function is to "load the exponent" in particular formats of floating point number. Saying *ldexp(True, 3)*, and having it work is kind of weird.

One way to tackle this would be to add an *exact* type, both as a dimension and a data type, which indicates that broadcasting should be disallowed. For the discussion, in addition to *ldexp*, lets introduce a vector magnitude function *mag*, where we want to disallow scalar arrays to broadcast into it.:

```
# ldexp signatures
(A... * exact[float32], A... * int32) -> A... * float32
(A... * exact[float64], A... * int64) -> A... * float64

# mag signatures
(A... * exact[2] * float32) -> A... * float32
(A... * exact[3] * float32) -> A... * float32

# ufunc but disallowing broadcasting
(exact[A...] * int32, exact[A...] * int32) -> A... * int32
```

A possible syntactic sugar (which I'm not attached to, I think this needs some exploration) for this is:

```
# ldexp signatures
(A... * float32=, A... * int32) -> A... * float32
(A... * float64=, A... * int64) -> A... * float64

# mag signatures
(A... * 2= * float32) -> A... * float32
(A... * 3= * float32) -> A... * float32

# ufunc but disallowing broadcasting
(A=.. * int32, A=.. * int32) -> A... * int32
```

3.4 Factoring a Set of Signatures

One of the main things the multiple dispatch in DataShape has to do is match input arrays against a set of signatures very efficiently. We need to be able to hide the abstraction we're creating, and provide performance competitive with, but ideally superior to, what NumPy provides in its ufunc system.

Factoring the set of signatures into two or more stages which are simpler to solve and can prune the possibilities more quickly is one way to do this abstraction hiding. Let's use the *add* function for our example, with the following subset of signatures. We've included the *datetime* signatures to dispel any notion that the signatures will always match precisely.:

```
# add signatures
(A... * int32, A... * int32) -> A... * int32
(A... * int64, A... * int64) -> A... * int64
(A... * float32, A... * float32) -> A... * float32
(A... * float64, A... * float64) -> A... * float64
(A... * timedelta, A... * timedelta) -> A... * timedelta
(A... * datetime, A... * timedelta) -> A... * datetime
(A... * timedelta, A... * datetime) -> A... * datetime
```

Because the broadcasting of all these cases is identical, we can transform this set of signatures into two stages as follows:

```
# broadcasting stage
(A... * X, A... * Y) -> A... * Z

# data type stage matched against (X, Y)
(int32, int32) -> int32
(int64, int64) -> int64
(float32, float32) -> float32
```

(continues on next page)

(continued from previous page)

```
(float64, float64) -> float64
(timedelta, timedelta) -> timedelta
(datetime, timedelta) -> datetime
(timedelta, datetime) -> datetime
```

Let’s work through this example to illustrate how it works.:

```
# Stage 1: Input arrays "3 * 1 * int32", "4 * float32"
#     (A... * X, A... * Y) -> A... * Z
int32 ==> X        # "int32" in X
3 * 1 * ==> A...   # "3 * 1 *" in A...
float32 ==> Y      # "float32" in Y
4 * ==> A...       # "4 *" in A...

# Solution: A... is "3 * 4 *", X is "int32", and Y is "float32"
# Stage 2: Input arrays "int32" and "float32"
#     (int32, int32) -> int32
int32 ==> int32      # LEGAL
float32 ==> int32    # ILLEGAL
#     (float32, float32) -> float32
int32 ==> float32    # LEGAL
float32 ==> float32 # LEGAL
# etc.

# Assume we picked (float32, float32) -> float32
# so the variables are:
# X is "float32"
# Y is "float32"
# Z is "float32"
# giving the solution substituted into stage 1:
(3 * 1 * float32, 4 * float32) -> 3 * 4 * float32
```

CHAPTER 4

DataShape Grammar

The datashape language is a DSL which describes the structure of data, abstracted from a particular implementation in a language or file format. Compared to the Python library NumPy, it combines *shape* and *dtype* together, and introduces a syntax for describing structured data.

Some of the basic features include:

- Dimensions are separated by asterisks.
- Lists of types are separated by commas.
- Types and Typevars are distinguished by the capitalization of the leading character. Lowercase for types, and uppercase for typevars.
- Type constructors operate using square brackets.
- Type constructors accept positional and keyword arguments, and their arguments may be:
  - datashape, string, integer, list of datashape, list of string, list of integer
- In multi-line datashape strings or files, comments start from # characters to the end of the line.

4.1 Some Simple Examples

Here are some simple examples to motivate the idea:

```
# Scalar types
bool
int32
float64

# Scalar types with missing data/NA support
?bool
?float32
?complex

```

(continues on next page)

(continued from previous page)

```
# Arrays
3 * 4 * int32
3 * 4 * int32
10 * var * float64
3 * complex[float64]

# Array of Structures
100 * {
    name: string,
    birthday: date,
    address: {
        street: string,
        city: string,
        postalcode: string,
        country: string
    }
}

# Structure of Arrays
{
    x: 100 * 100 * float32,
    y: 100 * 100 * float32,
    u: 100 * 100 * float32,
    v: 100 * 100 * float32,
}

# Structure with strings for field names
{
    'field 0': 100 * float32,
    'field 1': float32,
    'field 2': float32,
}

# Array of Tuples
20 * (int32, float64)

# Function prototype
(3 * int32, float64) -> 3 * float64

# Function prototype with broadcasting dimensions
(A... * int32, A... * int32) -> A... * int32
```

4.2 Syntactic Sugar

Many syntax elements in datashape are syntax sugar for particular type constructors. For dtypes, this is:

```
{x : int32, y : int16}    =>   struct[['x', 'y'], [int32, int16]]
(int64, float32)          =>   tuple[[int64, float32]]
(int64, float32) -> bool  =>   funcproto[[int64, float32], bool]
DTypeVar                  =>   typevar['DTypeVar']
?int32                    =>   option[int32]
2 * ?3 * int32            =>   2 * option[3 * int32]
```

For dims, this is:

```
3 * int32                =>   fixed[3] * int32
DimVar * int32           =>   typevar['DimVar'] * int32
... * int32              =>   ellipsis * int32
DimVar... * int32        =>   ellipsis['DimVar'] * int32
```

4.3 The DataShape Grammar

Dimension Type Symbol Table:

```
# Variable-sized dimension
var
```

Dimension Type Constructor Symbol Table:

```
# Arrays which are either missing or fully there
# option[3 * int32]
option
```

Data Type Symbol Table:

```
# Numeric
bool
# Two's complement binary integers
int8
int16
int32
int64
int128
# Unsigned binary integers
uint8
uint16
uint32
uint64
uint128
# IEEE 754-2008 binary### floating point binary numbers
float16
float32
float64
float128
# IEEE 754-2008 decimal### floating point decimal numbers
decimal32
decimal64
decimal128
# Arbitrary precision integer
bignum
# Alias for int32
int
# Alias for float64
real
# Alias for complex[float64]
complex
# Alias for int32 or int64 depending on platform
intptr
# Alias for uint32 or uint64 depending on platform
uintptr
```

(continues on next page)

(continued from previous page)

```
# A unicode string
string
# A single unicode code point
char
# A blob of bytes
bytes
# A date
date
# A string containing JSON
json
# No data
void
```

Data Type Constructor Symbol Table:

```
# complex[float32], complex[type=float64]
complex
# string['ascii'], string[enc='cp949']
string
# bytes[size=4,align=2]
bytes
# datetime[unit='minutes',tz='CST']
datetime
# categorical[type=string, values=['low', 'medium', 'high']]
categorical
# option[float64]
option
# pointer[target=2 * 3 * int32]
pointer
```

Tokens:

```
NAME_LOWER : [a-z][a-zA-Z0-9_]*
NAME_UPPER : [A-Z][a-zA-Z0-9_]*
NAME_OTHER : _[a-zA-Z0-9_]*
ASTERISK : \*
COMMA : ,
EQUAL : =
COLON : :
LBRACKET : \[
RBRACKET : \]
LBRACE : \{
RBRACE : \}
LPAREN : \(
RPAREN : \)
ELLIPSIS : \.\.\.
RARROW : ->
QUESTIONMARK : ?
INTEGER : 0(?![0-9])|[1-9][0-9]*
STRING : (?:"(?:[^"\n\r\\]|(?:\\u[0-9a-fA-F]{4})|(?:\\["bfnrt]))*")|(?:\'(?:[^\
↪'\n\r\\]|(?:\\u[0-9a-fA-F]{4})|(?:\\['bfnrt]))*"))*\')
```

Grammar:

```
# Datashape may start with a '?' or not to signal optionality
datashape : datashape_nooption
```

(continues on next page)

(continued from previous page)

```
          | QUESTIONMARK datashape_nooption

# Asterisk-separated list of dimensions, followed by data type
datashape_nooption : dim ASTERISK datashape
                   | dtype

# Dimension Type (from the dimension type symbol table)
dim : typevar
    | ellipsis_typevar
    | type
    | type_constr
    | INTEGER
    | ELLIPSIS

# Data Type (from the data type symbol table)
dtype : typevar
      | type
      | type_constr
      | struct_type
      | funcproto_or_tuple_type

# A type variable
typevar : NAME_UPPER

# A type variable with ellipsis
ellipsis_typevar : NAME_UPPER ELLIPSIS

# A bare type (from the data type symbol table)
type : NAME_LOWER

# Type Constructor (from the data type constructor symbol table)
type_constr : NAME_LOWER LBRACKET type_arg_list RBRACKET

# Type Constructor: list of arguments
type_arg_list : type_arg COMMA type_arg_list
              | type_kwarg_list
              | type_arg

# Type Constructor: list of arguments
type_kwarg_list : type_kwarg COMMA type_kwarg_list
                | type_kwarg

# Type Constructor : single argument
type_arg : datashape
         | INTEGER
         | STRING
         | list_type_arg

# Type Constructor : single keyword argument
type_kwarg : NAME_LOWER EQUAL type_arg

# Type Constructor : single list argument
list_type_arg : LBRACKET RBRACKET
              | LBRACKET datashape_list RBRACKET
              | LBRACKET integer_list RBRACKET
              | LBRACKET string_list RBRACKET
```

(continues on next page)

(continued from previous page)

```
datashape_list : datashape COMMA datashape_list
               | datashape

integer_list : INTEGER COMMA integer_list
             | INTEGER

string_list : STRING COMMA string_list
            | STRING


# Struct/Record type (allowing for a trailing comma)
struct_type : LBRACE struct_field_list RBRACE
            | LBRACE struct_field_list COMMA RBRACE

struct_field_list : struct_field COMMA struct_field_list
                  | struct_field

struct_field : struct_field_name COLON datashape

struct_field_name : NAME_LOWER
                  | NAME_UPPER
                  | NAME_OTHER
                  | STRING

# Function prototype is a tuple with an arrow to the output type
funcproto_or_tuple_type : tuple_type RARROW datashape
                        | tuple_type

# Tuple type (allowing for a trailing comma)
tuple_type : LPAREN tuple_item_list RPAREN
           | LPAREN tuple_item_list COMMA RPAREN

tuple_item_list : datashape COMMA tuple_item_list
                | datashape
```

CHAPTER 5

Release Notes

5.1 Release 0.5.4+8.gcae16a8

Release 0.5.4+8.gcae16a8

Date TBD

5.2 New Features

None

5.3 New Types

- datashape now supports a `categorical` type (#150).

5.4 Experimental Types

Warning: Experimental types are subject to change.

- Records can now be constructed with the new syntax: `R['field0':type0, 'field1':type1, ...]` where each slice object represents a field in the record. `R` acts as an alias for `Record` to make it more pleasant to construct these literal types (#186).

5.5 API Changes

- `datashape` no longer supports Python 2.6 (#189).
- `datashape` no longer support Python 3.3 (#191).
- The default `repr` of `Mono` subclass now prints out the slot names as keyword arguments next to their values (#188). For example

  Instead of

  ```
  >>> from datashape import Decimal
  >>> Decimal(precision=11, scale=2)
  Decimal(11, 2)
  ```

  we have

  ```
  >>> Decimal(precision=11, scale=2)
  Decimal(precision=11, scale=2)
  ```

- Fields are now always constructed with `str` in Record datashapes (#197).

5.6 Bug Fixes

- Makes the parser recognize `null` and `void` (#183).
- Cache the datashape hash value to avoid potentially expensive recomputation during memoization (#184).
- Fix discovery of strings that start with things that look like numbers (#190).
- Makes the parser recognize `object` (#193).
- Make string field names in Record types have the same string type (#200).
- Fix the reprs for `Function` objects (#194).

5.7 Miscellaneous

None

CHAPTER 6

Indices and tables

- genindex
- modindex
- search