ndtypes-python documentation

Release 0.1

Continuum

November 07, 2016

Contents

ndtypes-python implements the Python bindings for libndtypes. It is currently based on DyND-Python.

ndtypes-python will provide the functionality of Datashape while utilizing libndtypes as the backend.

Datashape is a data description language for modern array computing capable of handling complex memory layouts, tagged union data types, pattern matching, polymorphism and much more.

CHAPTER 1

Changelog

1.1 Version 0.1

Initial revision of ndtypes-python.

CHAPTER 2

Datashape

In this section we will be using the `ndt` module (See: ndtypes-python) to demonstrate the capabilities of datashape. The rest of this document assumes that the `ndt` module has been imported:

```
from ndtypes import ndt
```

2.1 Types

The set of all datashape types comprises *dtypes* and *arrays*.

2.2 Dtypes

An important notion in datashape is the `dtype`, which roughly translates to the element type of an array. In datashape, the `dtype` can be of arbitrary complexity and can contain e.g. tuples, records and functions.

2.2.1 Scalars

Scalars are the primitive C/C++ types. Most scalars are fixed-size and platform independent.

Fixed size

Datashape offers a number of fixed-size scalars. Here's how to construct a simple `int64_t` type:

```
>>> ndt.type('int64')
ndt.type('int64')
```

All fixed-size scalars:

void	boolean	signed int	unsigned int	float [2]	complex
`void`	`bool` [1]	`int8`	`uint8`	`float16`	`complex64` [3]
		`int16`	`uint16`	`float32`	`complex128` [4]
		`int32`	`uint32`	`float64`	
		`int64`	`uint64`	`float128`	
		`int128`	`uint128`		

Aliases

Datashape has a number of aliases for scalars, which are internally mapped to their corresponding platform specific fixed-size types. This is how to construct a `size_t`:

```
>>> ndt.type('size')
ndt.type('uint64')
```

Machine dependent aliases:

`intptr`	`intptr_t`
`size`	`size_t`
`uintptr`	`uintptr_t`

Machine independent aliases:

`int`	`int32_t`
`real`	`float64`
`complex`	`complex<float64>`

Since IEEE 754-2008 is required, `sizeof(double) == 8` and `double` maps to `float64`.

Complex

Even though complex numbers are scalars, datashape provides a type constructor syntax for them:

```
>>> ndt.type("complex[float32]")
ndt.type('complex[float32]')

>>> ndt.type("complex64")
ndt.type('complex[float32]')

>>> ndt.type("complex[real]")
ndt.type('complex[float64]')

>>> ndt.type("complex[float64]")
ndt.type('complex[float64]')
```

2.2.2 Chars, strings, bytes

Encodings

Datashape defines the following encodings for strings and characters. Each encoding has several aliases:

[2] IEEE 754-2008 binary floating point types
[1] implemented as `char`
[3] implemented as `complex<float32>`
[4] implemented as `complex<float64>`

canonical form	aliases	
'ascii'	'A'	'us-ascii'
'utf8'	'U8'	'utf-8'
'utf16'	'U16'	'utf-16'
'utf32'	'U32'	'utf-32'
'ucs2'	'ucs_2'	'ucs2'

As seen in the table, encodings must be given in string form:

```
>>> ndt.type("char['ucs2']")
ndt.type("char['ucs2']")
```

Chars

The `char` constructor accepts `'ascii'`, `'ucs2'` and `'utf32'` encoding arguments. `char` without arguments is equivalent to `char[utf32]`.

```
>>> ndt.type("char['ascii']")
ndt.type("char['ascii']")

>>> ndt.type("char['utf32']")
ndt.type('char')

>>> ndt.type("char")
ndt.type('char')
```

UTF-8 strings

The `string` type is a variable length UTF-8 string:

```
>>> t = ndt.type("string")
>>> t.encoding
'utf8'
```

Fixed size strings

The `fixed_string` type takes a length and an optional encoding argument:

```
>>> t = ndt.type("fixed_string[1729]")
>>> t.data_size
1729
>>> t.encoding
'utf8'

>>> t = ndt.type("fixed_string[1729, 'utf16']")
>>> t.data_size
3458
>>> t.encoding
'utf16'
```

Bytes

The *bytes* type is variable length and takes an optional alignment argument. Valid values are powers of two in the range `[1, 16]`.

```
>>> t = ndt.type("bytes")
>>> t.data_alignment
8
>>> t.target_alignment
1

>>> t = ndt.type("bytes[align=2]")
>>> t.data_alignment
8
>>> t.target_alignment
2
```

Fixed size bytes

The `fixed_bytes` type takes a length and an optional alignment argument. The latter is a keyword-only argument in order to prevent accidental swapping of the two integer arguments:

```
>>> t = ndt.type("fixed_bytes[32]")
>>> t.data_size
32
>>> t.data_alignment
1

>>> t = ndt.type("fixed_bytes[128, align=8]")
>>> t.data_size
128
>>> t.data_alignment
8
```

2.2.3 Pointers

Datashape pointers are fully general and can point to types of arbitrary complexity:

```
>>> ndt.type("pointer[int64]")
ndt.type('pointer[int64]')

>>> ndt.type("pointer[10 * {a: int, b: 10 * float64}]")
ndt.type('pointer[10 * {a: int32, b: 10 * float64}]')
```

2.2.4 Option type

The option type provides safe handling of values that may or may not be present. The concept is well-known from languages like ML or SQL.

Two equivalent notations exist:

```
>>> ndt.type("option[complex]")
ndt.type('?complex[float64]')

>>> ndt.type('?complex[float64]')
ndt.type('?complex[float64]')
```

2.2.5 Dtype variables

Dtype variables are used in quantifier free type schemes and pattern matching. The range of a variable extends over the entire type term.

```
>>> ndt.type("T")
ndt.type('T')

>>> ndt.type("10 * 16 * T")
ndt.type('10 * 16 * T')
```

2.2.6 Symbolic constructors

Symbolic constructors stand for any constructor that takes the given datashape argument. Used in pattern matching.

```
>>> ndt.type("T[int32]")
ndt.type('T[int32]')
```

2.2.7 Type kinds

Type kinds denote specific subsets of *dtypes*, *types* or *dimension types*. Type kinds are in the dtype section because of the way the grammar is organized. Currently available are:

type kind	set	specific subset
`Any`	`datashape`	`datashape`
`Scalar`	`dtypes`	`scalars`
`Categorical`	`dtypes`	`categoricals` [5]
`FixedString`	`dtypes`	`fixed_strings`
`FixedBytes`	`dtypes`	`fixed_bytes`
`Fixed`	`dimension kind instances`	`fixed dimensions`

Type kinds are used in *pattern matching*.

2.2.8 Composite types

Datashape has container and function *dtypes*.

Tuples

As usual, the tuple type is the product type of a fixed number of types:

```
>>> ndt.type("(int64, float32, string)")
ndt.type('(int64, float32, string)')
```

Tuples can be nested:

```
>>> ndt.type("(bytes, (int8, fixed_string[10]))")
ndt.type('(bytes, (int8, fixed_string[10]))')
```

[5] currently disabled

Records

Records are equivalent to tuples with named fields:

```
>>> ndt.type("{a: float32, b: float64}")
ndt.type('{a: float32, b: float64}')
```

Functions

In datashape, function types can have positional and keyword arguments. Internally, positional arguments are represented by a tuple and keyword arguments by a record. Both kinds of arguments can be variadic.

Positional-only

This is a function type with a single positional `int32` argument, returning an `int32`:

```
>>> ndt.type("(int32) -> int32")
ndt.type('(int32) -> int32')
```

This is a function type with three positional arguments:

```
>>> ndt.type("(int32, complex128, string) -> float64")
ndt.type('(int32, complex[float64], string) -> float64')
```

Positional-variadic

This is a function type with a single required positional argument, followed by any number of additional positional arguments:

```
>>> ndt.type("(int32, ...) -> int32")
ndt.type('(int32, ...) -> int32')
```

Keyword-only

Keywords are specified inline:

```
>>> ndt.type("(distance: float32, velocity: float32) -> float32")
ndt.type('(distance: float32, velocity: float32) -> float32')
```

Keyword-variadic

This is a function type with a single required keyword argument, followed by any number of additional keyword arguments:

```
>>> ndt.type("(sum: float64, ...) -> float64")
ndt.type('(sum: float64, ...) -> float64')
```

Mixed

Function types can have both positional and keyword arguments, the former must precede the latter:

```
>>> ndt.type("(uint32, uint32, product: float64) -> float64")
ndt.type('(uint32, uint32, product: float64) -> float64')
```

Mixed-variadic

Any combination of positional-variadic and keyword-variadic is permitted.

This function has positional-variadic arguments, followed by keyword arguments:

```
>>> ndt.type("(uint64, ..., scale: uint8) -> uint64")
ndt.type('(uint64, ..., scale: uint8) -> uint64')
```

Positional arguments, followed by keyword-variadic arguments:

```
>>> ndt.type("(uint64, scale: uint8, ...) -> uint64")
ndt.type('(uint64, scale: uint8, ...) -> uint64')
```

Positional-variadic and keyword-variadic:

```
>>> ndt.type("(..., color: uint32, ...) -> uint64")
ndt.type('(..., color: uint32, ...) -> uint64')
```

2.3 Arrays

In datashape dimension kinds [6] are part of array type declarations. Datashape supports the following dimension kinds:

2.3.1 Fixed Dimension

A fixed dimension denotes an array type with a fixed number of elements of a specific type. The type can be written in two ways:

```
>>> ndt.type("fixed[10] * uint64")
ndt.type('10 * uint64')

>>> ndt.type("10 * uint64")
ndt.type('10 * uint64')
```

Formally, `fixed[10]` is a dimension constructor, not a type constructor. The `*` is the array type constructor in infix notation, taking as arguments a dimension and an element type.

The second form is equivalent to the first one. For users of other languages, it may be helpful to view this type as `array[10] of uint64`.

Multidimensional arrays are constructed in the same manner, the `*` is right associative:

```
>>> ndt.type("10 * 25 * float64")
ndt.type('10 * 25 * float64')
```

[6] In the whole text *dimension kind* and *dimension* are synonymous.

Again, it may help to view this type as `array[10] of (array[25] of float64)`.

In this case, `float64` is the *dtype* of the multidimensional array.

Dtypes can be arbitrarily complex. Here is an array with a dtype of a record that contains another array:

```
>>> ndt.type("120 * {size: int32, items: 10 * int8}")
ndt.type('120 * {size: int32, items: 10 * int8}')
```

2.3.2 Variable Dimension

The variable dimension kind describes an array type with a variable number of elements of a specific type:

```
>>> ndt.type("var * float32")
ndt.type('var * float32')
```

In this case, `var` is the dimension constructor and the `*` fulfils the same role as above. Many managed languages have variable sized arrays, so this type could be viewed as `array of float32`. In a sense, fixed size arrays are just a special case of variable sized arrays.

Dimension kinds can be mixed freely:

```
>>> ndt.type("10 * var * char")
ndt.type('10 * var * char')
```

Which corresponds to `array[10] of (array of char)`.

2.3.3 Symbolic Dimension

Datashape supports symbolic dimensions, which are used in pattern matching. A symbolic dimension is an uppercase variable that stands for a fixed dimension [7].

In this manner entire sets of array types can be specified. The following type describes the set of all `M * N` matrices with a `float32` dtype:

```
>>> ndt.type("M * N * float32")
ndt.type('M * N * float32')
```

The next type describes a function that performs matrix multiplication on any permissible pair of input matrices with dtype `T`:

```
>>> ndt.type("(M * N * T, N * P * T) -> M * P * T")
ndt.type('(M * N * T, N * P * T) -> M * P * T')
```

In this case, we have used both symbolic dimensions and the type variable `T`.

Symbolic dimensions can be mixed with other dimension kinds:

```
>>> ndt.type("10 * N * var * real")
ndt.type('10 * N * var * float64')
```

2.3.4 Ellipsis Dimension

The ellipsis, used in pattern matching, stands for any number of dimensions. Datashape supports both named and unnamed ellipses:

[7] It is currently under debate if symbolic dimensions should be restricted to match fixed dimensions only.

```
>>> ndt.type("... * float32")
ndt.type('M * N * float32')
```

Named form:

```
>>> ndt.type("Dim... * float32")
ndt.type('Dim... * float32')
```

Ellipsis dimensions play an important role in broadcasting, more on the topic in the section on pattern matching.

2.3.5 Power Dimension

Power dimensions are syntactic sugar for repeated dimension kinds:

```
>>> ndt.type("128**2 * float32")
ndt.type('128 * 128 * float32')

>>> ndt.type("var**3 * (complex, complex)")
ndt.type('var * var * var * (complex[float64], complex[float64])')

>>> ndt.type("N**3 * {a: int32, b: int64}")
ndt.type('N * N * N * {a: int32, b: int64}')
```

Ellipsis dimensions cannot be repeated.

CHAPTER 3

Pattern matching

The libndtypes implementation of datashape is dynamically typed with strict type checking. Static type checking of datashape would be far more complex, since datashape allows dependent types [1], i.e. types depending on values.

Dynamic pattern matching is used for checking function arguments, return values, broadcasting and general array functions.

Again, we will be using the `ndtypes` module included in ndtypes-python to demonstrate datashape pattern matching. The rest of this document assumes that the `ndt` module has been imported:

```
from ndtypes import ndt
```

3.1 General notes

`ndt.type` instances have a `match()` method for determining whether the argument type is compatible with the instance type. The match succeeds if and only if the set of types described by the right hand side is a subset of the set of types described by the left hand side.

3.1.1 Simple example

```
>>> ndt.type("Any").match("int32")
True
```

3.1.2 Non-commutativity

From the above definition it follows that pattern matching is not commutative:

```
>>> ndt.type("int32").match("Any")
False
```

3.1.3 Concrete matching

Much like members of the alphabet in regular expressions, concrete types match themselves:

[1] An argument is often made that the term *dependent types* should be reserved for static type systems. We use it here while explicitly acknowledging that the datashape implementation is dynamically typed.

```
>>> ndt.type("int32").match("int32")
True

>>> ndt.type("10 * var * float32").match("10 * var * float32")
True

>>> ndt.type("10 * var * float64").match("10 * var * float32")
False
```

3.2 Type kinds

Type kinds are named subsets of *types*.

Unlike *dtype variables*, matching type kinds does not require that a well defined substitution exists. Two instances of a type kind can match different types:

```
>>> ndt.type("(Any) -> Any").match("(float64) -> int32")
True
```

3.2.1 Any

The *Any* type kind is the most general and describes the set of all *types*.

Here's how to match a dtype against the set of all types:

```
>>> ndt.type("Any").match("int32")
True
```

This matches an array type against the set of all types:

```
>>> ndt.type("Any").match("10 * 5 * { v: float64, t: float64 }")
True
```

3.2.2 Scalar

The *Scalar* type kind stands for the set of all *scalars*.

`int32` is a member of the set of all scalars:

```
>>> ndt.type("Scalar").match("int32")
True
```

A pattern for a function that takes any type and returns a scalar:

```
>>> ndt.type("(Any) -> Scalar").match("(10 * complex128) -> float64")
True

>>> ndt.type("(Any) -> Scalar").match("(?{a: 10 * uint8}) -> uint8")
True

>>> ndt.type("(Any) -> Scalar").match("(?{a: 10 * uint8}) -> 10 * uint8")
False
```

Only the type kind is important, not the concrete type:

```
>>> ndt.type("(Scalar, Scalar)").match("(uint8, float64)")
True
```

3.2.3 Categorical

The set of all categorical types. Categorical types are currently not implemented.

3.2.4 FixedString

The set of all *fixed string* types.

```
>>> ndt.type("FixedString").match("fixed_string[100]")
True

>>> ndt.type("FixedString").match("fixed_string[100, 'utf16']")
True

>>> ndt.type("FixedString").match("string")
False
```

3.2.5 FixedBytes

The set of all *fixed bytes* types.

```
>>> ndt.type("FixedBytes").match("fixed_bytes[100]")
True

>>> ndt.type("FixedBytes").match("fixed_bytes[100, align=2]")
True

>>> ndt.type("FixedBytes").match("bytes[align=2]")
False
```

3.3 Dimension kinds

Dimension kinds stand for the set of all instances of the respective kind.

3.3.1 Fixed

The set of all instances of the *fixed dimension* kind.

```
>>> ndt.type("Fixed * var * bool").match("10 * var * bool")
True

>>> ndt.type("Fixed * var * bool").match("var * var * bool")
False

>>> ndt.type("Fixed * var * bool").match("N * var * bool")
False
```

3.4 Dtype variables

dtype variables are placeholders for dtypes. It is important to note that they are *not* general type variables. For example, they do not match *array types*, a concept which is used in general array functions [3], whose base cases may operate on a dtype.

This matches a record against a single *dtype* variable:

```
>>> ndt.type("T").match("{v: float64, t: float64}")
True
```

An *array* is not a *dtype*, so this match fails:

```
>>> ndt.type("T").match("10 * 5 * {v: float64, t: float64}")
False
```

Match against several dtype variables in a tuple type:

```
>>> ndt.type("(T, T, S)").match("(int32, int32, bool)")
True

>>> ndt.type("(T, T, S)").match("(int32, int64, bool)")
False
```

3.5 Symbolic dimensions

Recall that *array* types include the dimension kind, which can be symbolic.

3.5.1 Simple symbolic match

This matches a concrete fixed size array against the set of all one-dimensional fixed size [2] arrays:

```
>>> ndt.type("N * float64").match(ndt.type("100 * float64"))
True
```

3.5.2 Symbolic-symbolic match

Symbolic dimensions also match against other symbolic dimensions:

```
>>> ndt.type("N * float64").match(ndt.type("M * float64"))
True
```

3.5.3 Symbolic+Dtypevar

Symbolic dimensions can be used in conjunction with dtype variables:

```
>>> ndt.type("N * T").match(ndt.type("10 * float32"))
True
```

[3] Additional section needed.

[2] It is currently under debate whether symbolic dimensions should only match fixed size dimensions or also include variable size dimensions.

3.5.4 Ellipsis match

Finally, all dimension kinds (including multiple dimensions) match against ellipsis dimensions (named or unnamed):

```
>>> ndt.type("... * float64").match(ndt.type("N * float64"))
True

>>> ndt.type("... * float64").match(ndt.type("10 * N * float64"))
True

>>> ndt.type("Dim... * float64").match(ndt.type("10 * 20 * float64"))
True
```

This is used in broadcasting [3].

CHAPTER 4

Constants and functions

```
from ndtypes import ndt
```

4.1 Constants

The *ndt* namespace includes the following dtype constants. Currently they are an alternative for using the *ndt.type* constructor.

```
>>> ndt.bool
ndt.type('bool')

# It follows that the constants do not define separate classes.
>>> type(ndt.bool)
<class 'ndtypes.ndt.type.type'>
```

Constants
void
bool
int8
int32
int64
int128
uint8
uint32
uint64
uint128
float16
float64
float128
complex64
complex128
complex_float32
complex_float64

4.2 Functions

Some dtypes and arrays can also be constructed using functions instead of the string notation.

4.2.1 Create dtypes

ndt.**make_string**()
Create a variable length string dtype with encoding UTF8.

```
>>> ndt.make_string()
ndt.type('string')
```

ndt.**make_fixed_string**(*length*, *encoding='utf8'*)
Create a fixed string dtype.

```
>>> ndt.make_fixed_string(100, encoding='utf32')
ndt.type("fixed_string[100, 'utf32']")
```

ndt.**make_fixed_bytes**(*length*, *align*)
Create a fixed bytes dtype.

```
>>> ndt.make_fixed_bytes(100, 2)
ndt.type('fixed_bytes[100, align=2]')
```

ndt.**make_tuple**(*field_types*)
Create a tuple dtype.

```
>>> ndt.make_tuple([ndt.int64, ndt.complex128])
ndt.type('(int64, complex[float64])')
```

ndt.**make_struct**(*field_types*, *field_names*)
Create a struct dtype.

```
>>> ndt.make_struct([ndt.int64], ["x"])
ndt.type('{x: int64}')
```

4.2.2 Create arrays

ndt.**make_fixed_dim**(*size*, *dtype*)
Create an array with a fixed dimension.

```
>>> ndt.make_fixed_dim(10, ndt.int32)
ndt.type('10 * int32')
```

```
>>> ndt.make_fixed_dim(20, ndt.make_fixed_dim(10, ndt.int32))
ndt.type('20 * 10 * int32')
```

ndt.**make_var_dim**(*dtype*)
Create an array with a variable dimension.

```
>>> ndt.make_var_dim(ndt.int32)
ndt.type('var * int32')
```

```
>>> ndt.make_var_dim(ndt.make_var_dim(ndt.int32))
ndt.type('var * var * int32')
```

ndt.**make_fixed_dim_kind**(*dtype*)
Create an array with a fixed dimension kind.

```
>>> ndt.make_fixed_dim_kind(ndt.int32)
ndt.type('Fixed * int32')
```

CHAPTER 5

Numpy support

ndtypes-python currently has limited NumPy interoperability.

```
from ndtypes import ndt
import numpy as np
```

5.1 Dtype conversion

5.1.1 Convert a NumPy dtype to a Datashape dtype

```
>>> x = np.zeros(dtype="U64", shape=[10,2])
>>> ndt.type(x.dtype)
ndt.type("fixed_string[64, 'utf32']")

# Aligned structs are supported
>>> dt = np.dtype([('foo', 'i4'),('bar', 'f4'), ('baz', 'S10')], align=True)
>>> x = np.array([(1,2.,'Hello'), (2,3.,"World")], dtype=dt)
>>> ndt.type(x.dtype)
ndt.type("{foo: int32, bar: float32, baz: fixed_string[10, 'ascii']}")
```

5.1.2 Convert a Datashape dtype to a NumPy dtype

```
>>> t = ndt.type("int64")
>>> t.as_numpy()
dtype('int64')

# Structs are currently not supported in this direction
>>> t = ndt.type("{foo: int32, bar: float32}")
>>> t.as_numpy()
TypeError: cannot convert dynd type {foo: int32, bar: float32} into a Numpy dtype
```

5.1.3 Extract the shape from a Datashape array

```
>>> t = ndt.type("10 * 20 * int64")
>>> t.shape
(10, 20)
```

CHAPTER 6

Various functionality

This section contains functionality of ndtypes-python that may or may not be worth preserving.

```
from ndtypes import ndt
```

6.1 Syntax sugar for creating array types

```
>>> ndt.fixed[10] * ndt.bool
ndt.type('10 * bool')

>>> ndt.var * ndt.fixed[10] * '{x : int32, y : float32}'
ndt.type('var * 10 * {x: int32, y: float32}')
```

CHAPTER 7

Grammar

7.1 Lexing

The following table contains lexemes mapped to their corresponding tokens. Keywords currently serve a double purpose in record field names.

7.2 Lexeme/Token Table

```
(*** Scalars ***)

(** fixed-size **)
"void" -> VOID
"bool" -> BOOL

"int8" -> INT8
"int16" -> INT16
"int32" -> INT32
"int64" -> INT64
"int128" -> INT128

"uint8" -> UINT8
"uint16" -> UINT16
"uint32" -> UINT32
"uint64" -> UINT64
"uint128" -> UINT128

(* binary floating point (IEEE 754-2008) *)
"float16" -> FLOAT16
"float32" -> FLOAT32
"float64" -> FLOAT64
"float128" -> FLOAT128

(* complex numbers (IEEE 754-2008) *)
"complex64" ->  COMPLEX64     (* complex<float32> *)
"complex128" -> COMPLEX128    (* complex<float64> *)

(** aliases **)
(* Machine dependent *)
"size" -> SIZE          (* size_t *)
"intptr" -> INTPTR      (* intptr_t *)
```

```
"uintptr" -> UINTPTR    (* uintptr_t *)

(* machine independent *)
"int" -> INT            (* int32_t *)
"real" -> REAL          (* float64 *)
"complex" -> COMPLEX    (* complex<float64> *)


(*** Chars, strings, bytes ***)
"char" -> CHAR                  (* utf32 character *)
"string" -> STRING              (* utf8 string *)
"fixed_string" -> FIXED_STRING (* fixed_string[size] or fixed_string[size, encoding] *)
"bytes" -> BYTES                (* bytes[align=2] *)
"fixed_bytes" -> FIXED_BYTES    (* fixed_bytes[size] or fixed_bytes[size, align=2] *)
"align" -> ALIGN                (* keyword for the "bytes" constructor *)


(*** Pointers, option type ***)
"pointer" -> POINTER  (* pointer[datashape] *)
"option" -> OPTION    (* option[datashape] *)


(*** Dimension kinds ***)
"fixed" -> FIXED (* fixed[size] *)
"var" -> VAR     (* var *)


(*** Type kinds ***)
"Any" -> ANY_KIND
"Scalar" -> SCALAR_KIND
"Categorical" -> CATEGORICAL_KIND
"FixedBytes" -> FIXED_BYTES_KIND
"FixedString" -> FIXED_STRING_KIND
"Fixed" -> FIXED_DIM_KIND


(*** Punctuation ***)
"," -> COMMA
":" -> COLON
"(" -> LPAREN
")" -> RPAREN
"{" -> LBRACE
"}" -> RBRACE
"[" -> LBRACK
"]" -> RBRACK
"*" -> STAR
"**" -> DOUBLESTAR
"..." -> ELLIPSIS
"->" -> RARROW
"=" -> EQUAL
"?" -> QUESTIONMARK


(*** Value-carrying tokens ***)
INTEGER     (* natural number *)
NAME_LOWER (* ['a'-'z']['a'-'z' 'A'-'Z' '0'-'9' '_']* *)
NAME_UPPER (* ['A'-'Z']['a'-'z' 'A'-'Z' '0'-'9' '_']* *)
NAME_OTHER (* '_' ['A'-'Z']['a'-'z' 'A'-'Z' '0'-'9' '_']* *)
```

```
STRINGLIT  (* only used for encoding arguments: 'utf8' etc.  *)
```

7.3 Encodings

`fixed_string` and `char` take encoding arguments, which must be given as string literals. The preferred spelling uses single quotes (here double quotes are used for better syntax highlighting):

```
"A"     | "ascii" | "us-ascii" -> Ascii
"U8"    | "utf8"  | "utf-8"    -> Utf8
"U16"   | "utf16" | "utf-16"   -> Utf16
"U32"   | "utf32" | "utf-32"   -> Utf32
"ucs2"  | "ucs-2" | "ucs_2"    -> Ucs2
```

7.4 Grammar

This is the actual grammar in BNF form:

```
input:
  datashape EOF

datashape:
  datashape_nooption
| QUESTIONMARK datashape_nooption
| OPTION LBRACK datashape_nooption RBRACK

datashape_nooption:
  (* dimension types *)
  INTEGER STAR datashape
| FIXED LBRACK INTEGER RBRACK STAR datashape
| FIXED_DIM_KIND STAR datashape
| VAR STAR datashape
| NAME_UPPER STAR datashape
| ELLIPSIS STAR datashape
| NAME_UPPER ELLIPSIS STAR datashape

  (* power dimension syntax sugar *)
| INTEGER DOUBLESTAR INTEGER STAR datashape
| VAR DOUBLESTAR INTEGER STAR datashape
| NAME_UPPER DOUBLESTAR INTEGER STAR datashape

| ANY_KIND

| dtype

dtype:
  (***** Scalars *****)

  (*** fixed-size ***)
  VOID
| BOOL

| INT8
| INT16
```

```
| INT32
| INT64
| INT128

| UINT8
| UINT16
| UINT32
| UINT64
| UINT128

  (* binary floating point (IEEE 754-2008) *)
| FLOAT16
| FLOAT32
| FLOAT64
| FLOAT128

  (* complex numbers (IEEE 754-2008) *)
| COMPLEX64
| COMPLEX128

  (*** aliases ***)
  (* machine independent *)
| INT
| REAL
| COMPLEX

  (* machine dependent *)
| INTPTR
| UINTPTR
| SIZE

  (*** complex constructor (scalars internally) ***)
  (* complex[float32] *)
| COMPLEX LBRACK FLOAT32 RBRACK
  (* complex[float64] *)
| COMPLEX LBRACK FLOAT64 RBRACK
  (* complex[real] *)
| COMPLEX LBRACK REAL RBRACK

  (***** Chars, strings, bytes *****)
  (* char[encoding] *)
| CHAR LBRACK STRINGLIT RBRACK

  (* alias: unicode character (utf32) *)
| CHAR

  (* unicode string (utf8) *)
| STRING

  (* fixed_string[size] *)
| FIXED_STRING LBRACK INTEGER RBRACK
  (* fixed_string[size, encoding] *)
| FIXED_STRING LBRACK INTEGER COMMA STRINGLIT RBRACK

  (* bytes[align] (target alignment) *)
| BYTES LBRACK ALIGN EQUAL INTEGER RBRACK

  (* fixed_bytes[size, align] (data alignment) *)
```

```
| FIXED_BYTES LBRACK INTEGER COMMA ALIGN EQUAL INTEGER RBRACK

  (* pointer[datashape] *)
| POINTER LBRACK datashape RBRACK

  (* dtype variable *)
| NAME_UPPER

  (* dtype kinds *)
| SCALAR_KIND
| CATEGORICAL_KIND
| FIXED_BYTES_KIND
| FIXED_STRING_KIND

| NAME_UPPER LBRACK datashape RBRACK

| tuple_type
| struct_type
| function_type

variadic_flag:
  (* empty *)
| ELLIPSIS

comma_variadic_flag:
  (* empty *)
| COMMA
| COMMA ELLIPSIS

tuple_type:
  LPAREN variadic_flag RPAREN
| LPAREN tuple_item_list comma_variadic_flag RPAREN

tuple_item_list:
  datashape
| tuple_item_list COMMA datashape

struct_type:
  LBRACE variadic_flag RBRACE
| LBRACE struct_field_list comma_variadic_flag RBRACE

struct_field_list:
  struct_field
| struct_field_list COMMA struct_field

struct_field:
  struct_field_name COLON datashape

struct_field_name:
  NAME_LOWER
| NAME_UPPER
| NAME_OTHER
| keyword

function_type:
  tuple_type RARROW datashape
| LPAREN struct_field_list comma_variadic_flag RPAREN RARROW datashape
| LPAREN tuple_item_list COMMA struct_field_list comma_variadic_flag RPAREN
```

```
  (* line continued *) RARROW datashape
| LPAREN tuple_item_list COMMA ELLIPSIS COMMA struct_field_list comma_variadic_flag RPAREN
  (* line continued *) RARROW datashape

(* record fields may have keyword names *)
keyword:
  VOID
| BOOL
| INT8
| INT16
| INT32
| INT64
| INT128
| UINT8
| UINT16
| UINT32
| UINT64
| UINT128
| FLOAT16
| FLOAT32
| FLOAT64
| FLOAT128
| COMPLEX64
| COMPLEX128
| INTPTR
| UINTPTR
| SIZE
| REAL
| COMPLEX
| INT
| CHAR
| STRING
| FIXED_STRING
| BYTES
| FIXED_BYTES
| POINTER
| OPTION
| FIXED
| VAR
| ALIGN
| ANY_KIND
| SCALAR_KIND
| CATEGORICAL_KIND
| FIXED_BYTES_KIND
| FIXED_STRING_KIND
| FIXED_DIM_KIND
```

CHAPTER 8

Abstract Syntax Tree

An abstract syntax tree provides an excellent insight into the datashape tree structure. Internally, libndtypes uses an object tree that closely mirrors this presentation.

8.1 AST

Following CPython's tradition of specifying the AST in ASDL form, here we use a very similar OCaml data type:

```
type size = int
type alignment = int

type variadic_flag
  = Nonvariadic
  | Variadic

type encoding
  = Ascii
  | Utf8
  | Utf16
  | Utf32
  | Ucs2

type datashape =
  (* dtypes *)
  | Void
  | Bool

  | Int8
  | Int16
  | Int32
  | Int64
  | Int128

  | Uint8
  | Uint16
  | Uint32
  | Uint64
  | Uint128

  | Float16
  | Float32
  | Float64
```

```
  | Float128

  | Complex of datashape (* argument restricted to IEEE 754-2008 floats *)

  | Char of encoding

  | String (* utf8 string *)
  | FixedString of size * encoding option

  | Bytes of alignment
  | FixedBytes of size * alignment

  | Pointer of datashape
  | Option of datashape

  | CudaHost of datashape
  | CudaDevice of datashape

  | Constr of string * datashape (* general type constructor *)

  (* symbolic dtypes *)
  | Dtypevar of string

  (* dtype kinds (denoting specific subsets of dtypes) *)
  | ScalarKind
  | CategoricalKind
  | FixedBytesKind
  | FixedStringKind

  (* compound types *)
  | Tuple of variadic_flag * datashape list
  | Record of variadic_flag * field list
  | Function of parameters

  (* array types *)
  | FixedDim of size * datashape      (* equivalent to "array[size] of type" *)
  | VarDim of datashape               (* equivalent to "array of type" *)
  | SymbolicDim of string * datashape (* equivalent to "array[N] of type" *)
  | EllipsisDim of string * datashape (* any number of dimensions (... or Dim...) *)

  (* dimension kinds *)
  | FixedDimKind of datashape         (* set of all array[N] of type *)

  (* type kinds *)
  | AnyKind                           (* set of all types *)

and field = (string * datashape)

and parameters =
  { fun_ret: datashape;    (* any type *)
    fun_pos: datashape;    (* always a tuple *)
    fun_kwds: datashape }  (* always a record *)
```

M