ndtypes

Release v0.2.0dev3

Dec 11, 2018

Contents

ndtypes is a package for typing raw memory blocks using a close variant of the datashape type language.

CHAPTER 1

Libndtypes

C library.

1.1 libndtypes

libndtypes implements the type part of a compiler frontend. It can describe C types needed for array computing and additionally includes symbolic types for dynamic type checking.

libndtypes has the concept of abstract and concrete types. Concrete types contain the exact data layout and all sizes that are required to access subtypes or individual elements in memory.

Abstract types are for type checking and include functions, symbolic dimensions and type variables. Module support is planned at a later stage.

Concrete types with rich layout information make it possible to write relatively small container libraries that can traverse memory without type erasure.

1.1.1 Initialization and tables

libndtypes has global tables that need to be initialized and finalized.

```
int ndt_init(ndt_context_t *ctx);
```

Initialize the global tables. This function must be called once at program start before using any other libndtypes functions.

Return *0* on success and *-1* otherwise.

```
void ndt_finalize(void);
```

Deallocate the global tables. This function may be called once at program end for the benefit of memory debuggers.

```
int ndt_typedef_add(const char *name, const ndt_t *type, ndt_context_t *ctx);
```

Add a type alias for *type* to the typedef table. *name* must be globally unique. The function steals the *type* argument.

On error, deallocate *type* and return *-1*. Return *0* otherwise.

```
const ndt_t *ndt_typedef_find(const char *name, ndt_context_t *ctx);
```

Try to find the type associated with *name* in the typedef table. On success, return a const pointer to the type, `NULL` otherwise.

1.1.2 Context

The context is used to facilitate error handling. The context struct itself should not be considered public and is subject to change.

Constants

The *err* field of the context is set to one of the following enum values:

```
#include <ndtypes.h>

enum ndt_error {
  NDT_Success,
  NDT_ValueError,
  NDT_TypeError,
  NDT_InvalidArgumentError,
  NDT_NotImplementedError,
  NDT_LexError,
  NDT_ParseError,
  NDT_OSError,
  NDT_RuntimeError,
  NDT_MemoryError
};
```

Static contexts

```
NDT_STATIC_CONTEXT(ctx);
```

This creates a static context, usually a local variable in a function. Error messages may be dynamically allocated, so `ndt_context_del` must be called on static contexts, too.

Functions

```
ndt_context_t *ndt_context_new(void);
void ndt_context_del(ndt_context_t *ctx);
```

Create an initialized context or delete a context. It is safe to call `ndt_context_del` on both dynamic and static contexts.

```
void ndt_err_format(ndt_context_t *ctx, enum ndt_error err, const char *fmt, ...);
```

Set a context's error constant and error message. *fmt* may contain the same format specifiers as `printf`.

```
int ndt_err_occurred(const ndt_context_t *ctx);
```

Check if an error has occurred.

```
void ndt_err_clear(ndt_context_t *ctx);
```

Clear an error.

```
void *ndt_memory_error(ndt_context_t *ctx);
```

Convenience function. Set `NDT_MemoryError` and return `NULL`;

```
const char *ndt_err_as_string(enum ndt_error err);
```

Get the string representation of an error constant.

```
const char *ndt_context_msg(ndt_context_t *ctx);
```

Get the current error string. It is safe to call this function if no error has occurred, in which case the string is `Success`.

```
ndt_err_fprint(FILE *fp, ndt_context_t *ctx);
```

Print an error to *fp*. Mostly useful for debugging.

1.1.3 Types

Types are implemented as a tagged union. For the defined type enum values it is best to refer to `ndtypes.h` directly or to search the constructor functions below.

Abstract and concrete types

```
/* Protect access to concrete type fields. */
enum ndt_access {
  Abstract,
  Concrete
};
```

An important concept in libndtypes are abstract and concrete types.

Abstract types can have symbolic values like dimension or type variables and are used for type checking.

Concrete types additionally have full memory layout information like alignment and data size.

In order to protect against accidental access to undefined concrete fields, types have the *ndt_access* field that is set to *Abstract* or *Concrete*.

Flags

```
/* flags */
#define NDT_LITTLE_ENDIAN  0x00000001U
#define NDT_BIG_ENDIAN     0x00000002U
#define NDT_OPTION         0x00000004U
#define NDT_SUBTREE_OPTION 0x00000008U
#define NDT_ELLIPSIS       0x00000010U
```

The endian flags are set if a type has explicit endianness. If native order is used, they are unset.

`NDT_OPTION` is set if a type itself is optional.

`NDT_SUBTREE_OPTION` is set if any subtree of a type is optional.

`NDT_ELLIPSIS` is set if the tail of a dimension sequence contains an ellipsis dimension. The flag is not propagated to an outer array with a dtype that contains an inner array with an ellipsis.

Common fields

```
struct _ndt {
    /* Always defined */
    enum ndt tag;
    enum ndt_access access;
    uint32_t flags;
    int ndim;
    /* Undefined if the type is abstract */
    int64_t datasize;
    uint16_t align;
    ...
};
```

tag, *access* and *flags* are explained above. Every type has an *ndim* field even when it is not an array, in which case *ndim* is zero.

The *datasize* and *align* fields are defined for concrete types.

Abstract fields

```
union {
    ...

    struct {
        int64_t shape;
        ndt_t *type;
    } FixedDim;

    ...

};
```

These fields are always defined for both abstract and concrete types. `FixedDim` is just an example field. Refer to `ndtypes.h` directly for the complete set of fields.

Concrete fields

```
struct {
    union {
        struct {
            int64_t itemsize;
            int64_t step;
        } FixedDim;

    ...

    };
} Concrete;
```

These fields are only defined for concrete types. For internal reasons (facilitating copying etc.) they are initialized to zero for abstract types.

Type constructor functions

All functions in this section steal their arguments. On success, heap allocated memory like *type* and *name* arguments belong to the return value.

On error, all arguments are deallocated within the respective functions.

Special types

The types in this section all have some property that makes them different from the regular types.

```
ndt_t *ndt_option(ndt_t *type);
```

This constructor is unique in that it does *not* create a new type with an `Option` tag, but sets the `NDT_OPTION` flag of its argument.

The reason is that having a separate `Option` tag complicates the type traversal when using libndtypes.

The function returns its argument and cannot fail.

```
ndt_t *ndt_module(char *name, ndt_t *type, ndt_context_t *ctx);
```

The module type is for implementing type name spaces and is always abstract. Used in type checking.

```
ndt_t *ndt_function(ndt_t *ret, ndt_t *pos, ndt_t *kwds, ndt_context_t *ctx);
```

The function type is used for declaring function signatures. Used in type checking.

```
ndt_t *ndt_void(ndt_context_t *ctx)
```

Currently only used as the empty return value in function signatures.

Any type

```
ndt_t *ndt_any_kind(ndt_context_t *ctx);
```

Constructs the abstract *Any* type. Used in type checking.

Dimension types

```
ndt_t *ndt_fixed_dim(ndt_t *type, int64_t shape, int64_t step, ndt_context_t *ctx);
```

type is either a dtype or the tail of the dimension list.

shape is the dimension size and must be a natural number.

step is the amount to add to the linear index in order to move to the next dimension element. *step* may be negative.

If *step* is `INT64_MAX`, the steps are computed from the dimensions shapes and the resulting array is C-contiguous. This is the regular case.

If *step* is given, it is used without further checks. This is mostly useful for slicing. The computed datasize is the minimum datasize such that all index combinations are within the bounds of the allocated memory.

```
ndt_t *ndt_to_fortran(const ndt_t *type, ndt_context_t *ctx);
```

Convert a C-contiguous chain of fixed dimensions to Fortran order.

```
ndt_t *ndt_abstract_var_dim(ndt_t *type, ndt_context_t *ctx);
```

Create an abstract *var* dimension for pattern matching.

```
/* Ownership flag for var dim offsets */
enum ndt_offsets {
  InternalOffsets,
  ExternalOffsets,
};

ndt_t *ndt_var_dim(ndt_t *type,
                   enum ndt_offsets flag, int32_t noffsets, const int32_t *offsets,
                   int32_t nslices, ndt_slice_t *slices,
                   ndt_context_t *ctx);
```

Create a concrete *var* dimension. Variable dimensions are offset-based and use the same addressing scheme as the Arrow data format.

Offset arrays can be very large, so copying must be avoided. For ease of use, libndtypes supports creating offset arrays from a datashape string. In that case, *flag* must be set to `InternalOffsets` and the offsets are managed by the type.

However, in the most common case offsets are generated and managed elsewhere. In that case, *flag* must be set to `ExternalOffsets`.

The offset-based scheme makes it hard to store a sliced var dimension or repeatedly slice a var dimension. This would require additional shape arrays that are as large as the offset arrays.

Instead, var dimensions have the concept of a slice stack that stores all slices that need to be applied to a var dimension.

Accessing elements recomputes the (start, stop, step) triples that result from applying the entire slice stack.

The *nslices* and *slices* arguments are used to provide this stack. For an unsliced var dimension these arguments must be *0* and *NULL*.

```
ndt_t *ndt_symbolic_dim(char *name, ndt_t *type, ndt_context_t *ctx);
```

Create a dimension variable for pattern matching. The variable stands for a fixed dimension.

```
ndt_ellipsis_dim(char *name, ndt_t *type, ndt_context_t *ctx);
```

Create an ellipsis dimension for pattern matching. If *name* is non-NULL, a named ellipsis variable is created.

In pattern matching, multiple named ellipsis variables always stand for the exact same sequence of dimensions.

By contrast, multiple unnamed ellipses stand for any sequence of dimensions that can be broadcast together.

Container types

```
ndt_t *ndt_tuple(enum ndt_variadic flag, ndt_field_t *fields, int64_t shape,
                 uint16_opt_t align, uint16_opt_t pack, ndt_context_t *ctx);
```

Construct a tuple type. *fields* is the field sequence, *shape* the length of the tuple.

align and *pack* are mutually exclusive and have the exact same meaning as gcc's *aligned* and *packed* attributes applied to an entire struct.

Either of these may only be given if no field has an *align* or *pack* attribute.

```
ndt_t *ndt_record(enum ndt_variadic flag, ndt_field_t *fields, int64_t shape,
                  uint16_opt_t align, uint16_opt_t pack, ndt_context_t *ctx);
```

Construct a record (struct) type. *fields* is the field sequence, *shape* the length of the record.

align and *pack* are mutually exclusive and have the exact same meaning as gcc's *aligned* and *packed* attributes applied to an entire struct.

Either of these may only be given if no field has an *align* or *pack* attribute.

```
ndt_t *ndt_ref(ndt_t *type, ndt_context_t *ctx);
```

Construct a reference type. References are pointers whose contents (the values pointed to) are addressed transparently.

```
ndt_t *ndt_constr(char *name, ndt_t *type, ndt_context_t *ctx);
```

Create a constructor type. Constructor types are equal if their names and types are equal.

```
ndt_t *ndt_nominal(char *name, ndt_t *type, ndt_context_t *ctx);
```

Same as constructor, but the type is stored in a lookup table. Comparisons and pattern matching are only by name. The name is globally unique.

Scalars

```
ndt_t *ndt_scalar_kind(ndt_context_t *ctx);
```

Create a scalar kind type for pattern matching.

Categorical

```
ndt_t *ndt_categorical(ndt_value_t *types, int64_t ntypes, ndt_context_t *ctx);
```

Create a categorical type. The categories are given as an array of typed values.

Fixed string and fixed bytes

```
ndt_t *ndt_fixed_string_kind(ndt_context_t *ctx);
```

Create a fixed string kind symbolic type for pattern matching.

```
ndt_t *ndt_fixed_string(int64_t len, enum ndt_encoding encoding, ndt_context_t *ctx);
```

Create a fixed string type. *len is the length in code points, for *encoding* refer to the encodings section.

```
ndt_t *ndt_fixed_bytes(int64_t size, uint16_opt_t align, ndt_context_t *ctx);
```

Create a fixed bytes kind symbolic type for pattern matching.

```
ndt_t *ndt_fixed_bytes(int64_t size, uint16_opt_t align, ndt_context_t *ctx);
```

Create a fixed bytes type with size *size* and alignment *align*.

String, bytes, char

```
ndt_t *ndt_string(ndt_context_t *ctx);
```

Create a string type. The value representation in memory is a pointer to a `NUL`-terminated UTF-8 string.

```
ndt_t *ndt_bytes(uint16_opt_t target_align, ndt_context_t *ctx);
```

Create a bytes type. The value representation in memory is a struct containing an `int64_t` *size* field and a pointer to `uint8_t`.

The alignment of the pointer value is *target_align*.

```
ndt_t *ndt_char(enum ndt_encoding encoding, ndt_context_t *ctx);
```

Create a char type with a specific *encoding*. Encodings apart from UTF-32 may be removed in the future, since single UTF-8 chars etc. have no real meaning and arrays of UTF-8 chars can be represented by the fixed string type.

Integer kinds

```
ndt_t *ndt_signed_kind(ndt_context_t *ctx);
```

Create a symbolic signed kind type for pattern matching.

```
ndt_t *ndt_unsigned_kind(ndt_context_t *ctx);
```

Create a symbolic unsigned kind type for pattern matching.

```
ndt_t *ndt_float_kind(ndt_context_t *ctx);
```

Create a symbolic float kind type for pattern matching.

```
ndt_t *ndt_complex_kind(ndt_context_t *ctx);
```

Create a symbolic complex kind type for pattern matching.

Numbers

```
ndt_t *ndt_primitive(enum ndt tag, uint32_t flags, ndt_context_t *ctx);
```

Create a number type according to the given enum value. *flags* can be `NDT_LITTLE_ENDIAN` or `NDT_BIG_ENDIAN`.

If no endian flag is given, native order is assumed.

```
ndt_t *ndt_signed(int size, uint32_t flags, ndt_context_t *ctx);
```

Create a signed fixed width integer according to *size*. *flags* as above.

```
ndt_t *ndt_unsigned(int size, uint32_t flags, ndt_context_t *ctx);
```

Create an unsigned fixed width integer according to *size*. *flags* as above.

```
enum ndt_alias {
  Size,
  Intptr,
  Uintptr
};

ndt_t *ndt_from_alias(enum ndt_alias tag, uint32_t flags, ndt_context_t *ctx);
```

Create a fixed width integer type from an alias. Sizes are platform dependent.

Type variables

```
ndt_t *ndt_typevar(char *name, ndt_context_t *ctx);
```

Create a type variable for pattern matching.

1.1.4 Predicates

libndtypes has a number of type predicates.

```
int ndt_is_abstract(const ndt_t *t);
int ndt_is_concrete(const ndt_t *t);
```

Determine whether a type is abstract or concrete. These functions need to be called to check whether the concrete type fields are defined.

```
int ndt_is_optional(const ndt_t *t);
```

Check if a type is optional.

```
int ndt_subtree_is_optional(const ndt_t *t);
```

Check if a subtree of a type is optional. This is useful for deciding if bitmaps need to be allocated for subtrees.

```
int ndt_is_ndarray(const ndt_t *t);
```

Check if a type describes an n-dimensional (n > 0) array of fixed dimensions.

```
int ndt_is_c_contiguous(const ndt_t *t);
int ndt_is_f_contiguous(const ndt_t *t);
```

Check if a type is an n-dimensional (n > 0) contiguous C or Fortran array. Currently this returns 0 for scalars.

```
int ndt_is_scalar(const ndt_t *t);
```

Check if a type is a scalar.

```
int ndt_is_signed(const ndt_t *t);
int ndt_is_unsigned(const ndt_t *t);
int ndt_is_float(const ndt_t *t);
int ndt_is_complex(const ndt_t *t);
```

Check if a type is signed, unsigned, float or complex.

```
int ndt_endian_is_set(const ndt_t *t);
```

Check whether the endianness of a type is explicitly set.

```
int ndt_is_little_endian(const ndt_t *t);
int ndt_is_big_endian(const ndt_t *t);
```

Check whether a type is big or little endian. Use the native order if no endian flag is set.

1.1.5 Functions

Most library functions are for creating types. The functions in this section operate *on* types.

Copying

```
ndt_t *ndt_copy(const ndt_t *t, ndt_context_t *ctx);
```

Create a copy of the argument. This is an important function, since types should be immutable.

Equality

```
int ndt_equal(const ndt_t *t, const ndt_t *u);
```

Return 1 if *t* and *u* are structurally equal, *0* otherwise.

Pattern matching

```
int ndt_match(const ndt_t *p, const ndt_t *c, ndt_context_t *ctx);
```

Match concrete candidate *c* against the (possibly abstract) pattern *p*.

This is the main function used in type checking.

Type checking

```
ndt_t *ndt_typecheck(const ndt_t *f, const ndt_t *args, int *outer_dims, ndt_context_
↪t *ctx);
```

Take a function type *f*, check if it can accept the concrete type *args*. *args* must be a tuple type that contains the individual arguments.

The return value is the inferred return type.

Store the number of outer dimensions that need to be traversed before applying the function kernel.

1.1.6 Typedef

libndtypes has a global lookup table for type aliases. These aliases are treated as nominal types in pattern matching.

```
int ndt_init(ndt_context_t *ctx);
```

This function must be called at program start to initialize the typedef table.

```
int ndt_typedef(const char *name, ndt_t *type, ndt_context_t *ctx);
```

Create a nominal type alias for *type*. The function steals the *type* argument.

1.1.7 Input/output

Functions for creating and displaying types.

Input

```
ndt_t *ndt_from_file(const char *name, ndt_context_t *ctx);
```

Create a type from a file that contains the datashape representation.

```
ndt_t *ndt_from_string(const char *input, ndt_context_t *ctx);
```

Create a type from a string in datashape syntax. This is the primary function for creating types.

```
typedef struct {
  int num_offset_arrays;              /* number of offset arrays */
  int32_t num_offsets[NDT_MAX_DIM];     /* lengths of the offset arrays */
  int32_t *offset_arrays[NDT_MAX_DIM]; /* offset arrays */
} ndt_meta_t;

ndt_t *ndt_from_metadata_and_dtype(const ndt_meta_t *m, const char *dtype, ndt_
↪context_t *ctx);
```

Create a concrete var dimension using the external offset arrays given in the `ndt_meta_t` struct.

The application is responsible for keeping the offset arrays alive while the type *and all copies of the type* exist.

This is not as difficult as it sounds. One approach that utilizes a resource manager object is implemented in the Python ndtypes module.

```
ndt_t *ndt_from_bpformat(const char *input, ndt_context_t *ctx);
```

Create a type from a buffer protocol format string (PEP-3118 syntax). This is useful for translating dtypes in a *Py_buffer* struct.

The outer dimensions specified by the *Py_buffer* shape member need to be created separately.

Output

```
char *ndt_as_string(const ndt_t *t, ndt_context_t *ctx);
```

Convert *t* to its string representation. This currently omits some layout details like alignment, packing or Fortran layout.

```
char *ndt_indent(const ndt_t *t, ndt_context_t *ctx);
```

Same as `ndt_as_string`, but indent the result.

```
char *ndt_ast_repr(const ndt_t *t, ndt_context_t *ctx);
```

Return the representation of the abstract syntax tree of the input type. This representation includes all low level details.

1.1.8 Encodings

Some types support encoding parameters.

```
#include <ndtypes.h>

/* Encoding for characters and strings */
enum ndt_encoding {
  Ascii,
  Utf8,
  Utf16,
  Utf32,
  Ucs2,
};
```

Functions

```
enum ndt_encoding ndt_encoding_from_string(const char *s, ndt_context_t *ctx);
```

Convert a string to the corresponding enum value. The caller must use `ndt_err_occurred` to check for errors.

```
const char *ndt_encoding_as_string(enum ndt_encoding encoding);
```

Convert an encoding to its string representation.

```
size_t ndt_sizeof_encoding(enum ndt_encoding encoding);
```

Return the memory size of a single code point.

```
uint16_t ndt_alignof_encoding(enum ndt_encoding encoding);
```

Return the alignment of a single code point.

1.1.9 Fields and values

Some API functions expect fields for creating tuple or record types or values for creating categorical types.

Fields

```
enum ndt_option {
  None,
  Some
};

typedef struct {
  enum ndt_option tag;
  uint16_t Some;
} uint16_opt_t;
```

Due to the multitude of options in creating fields a number of functions take a *uint16_opt_t* struct. If *tag* is *None*, no value has been specified and the *Some* field is undefined.

If *tag* is *Some*, the value in the *Some* field has been explicitly given.

Functions

```
ndt_field_t *ndt_field(char *name, ndt_t *type, uint16_opt_t align,
                       uint16_opt_t pack, uint16_opt_t pad, ndt_context_t *ctx);
```

Create a new field. For tuples, *name* is `NULL`. The *align* and *pack* options are mutually exclusive and have exactly the same function as *gcc's aligned* and *packed* attributes when applied to individual fields.

The *pad* field has no influence on the field layout. It is present to enable sanity checks when an explicit number of padding bytes has been specified (Example: PEP-3118).

```
void ndt_field_del(ndt_field_t *field);
```

Deallocate a field.

```
void ndt_field_array_del(ndt_field_t *fields, int64_t shape);
```

Deallocate an array of fields.

Values

```
/* Selected values for the categorical type. */
enum ndt_value {
  ValBool,
  ValInt64,
  ValFloat64,
  ValString,
  ValNA,
};

typedef struct {
  enum ndt_value tag;
    union {
```

(continues on next page)

(continued from previous page)

```
      bool ValBool;
      int64_t ValInt64;
      double ValFloat64;
      char *ValString;
  };
} ndt_value_t;
```

The categorical type contains values. Currently a small number of primitive types are supported. It would be possible to use memory typed by *ndt_t* itself either by introducing a circular relationship between libndtypes and container libraries or by duplicating parts of a container library.

It remains to be seen if such an added complexity is useful.

```
ndt_value_t *ndt_value_from_number(enum ndt_value tag, char *v, ndt_context_t *ctx);
```

Construct a number or boolean value from a string. *tag* must be one of `ValBool`, `ValInt64`, or `ValFloat64`.

```
ndt_value_t *ndt_value_from_string(char *v, ndt_context_t *ctx);
```

Construct a `ValString` value from a string.

```
ndt_value_t *ndt_value_na(ndt_context_t *ctx);
```

Construct the `NA` value.

```
int ndt_value_equal(const ndt_value_t *x, const ndt_value_t *y);
```

Determine if two values are equal. `NA` compares not equal to itself.

```
ndt_value_mem_equal(const ndt_value_t *x, const ndt_value_t *y);
```

Determine if two values are structurally equal. `NA` compares equal to itself.

```
int ndt_value_compare(const ndt_value_t *x, const ndt_value_t *y);
```

Compare values according to a sorting order. `NA` compares equal to itself.

1.1.10 Memory handling

Type allocation and deallocation

```
ndt_t *ndt_new(enum ndt tag, ndt_context_t *ctx);
```

Allocate a new type according to *tag* with the common fields initialized to the default values.

Most types need additional initialization, so this function is rarely used on its own.

```
ndt_t *ndt_tuple_new(enum ndt_variadic flag, int64_t shape, ndt_context_t *ctx);
ndt_t *ndt_record_new(enum ndt_variadic flag, int64_t shape, ndt_context_t *ctx);
```

Allocate a new tuple or record type. Because of their internal complexity these types have dedicated allocation functions.

As above, the functions are never used outside of wrapper functions.

```
void ndt_del(ndt_t *t);
```

Deallocate a type. *t* may be `NULL`. This function is meant to be used by applications directly.

Custom allocators

```
extern void *(* ndt_mallocfunc)(size_t size);
extern void *(* ndt_callocfunc)(size_t nmemb, size_t size);
extern void *(* ndt_reallocfunc)(void *ptr, size_t size);
extern void (* ndt_freefunc)(void *ptr);
```

libndtypes allows applications to set custom allocators at program start. By default these global variables are set to the usual libc allocators.

Allocation/deallocation

```
void *ndt_alloc(int64_t nmemb, int64_t size);
```

Allocate *nmemb* * *size* bytes, using the function set in the custom allocator.

Overflow in the multiplication is checked. Return `NULL` on overflow or if the allocation fails.

```
void *ndt_alloc_size(size_t size);
```

Allocate *size* bytes, using the function set in the custom allocator.

Return `NULL` on overflow or if the allocation fails.

```
void *ndt_calloc(int64_t nmemb, int64_t size);
```

Allocate *nmemb* * *size* zero-initialized bytes, using the function set in the custom allocator.

Return `NULL` if the allocation fails.

```
void *ndt_realloc(void *ptr, int64_t nmemb, int64_t size);
```

Reallocate *ptr* to use *nmemb* * *size* bytes.

Return `NULL` on overflow or if the allocation fails. As usual, *ptr* is still valid after failure.

```
void ndt_free(void *ptr);
```

Free a pointer allocated by one of the above functions. *ptr* may be `NULL` if the custom allocator allows this – the C Standard requires `free` to accept `NULL`.

Aligned allocation/deallocation

```
void *ndt_aligned_calloc(uint16_t alignment, int64_t size);
```

Allocate *size* bytes with a guaranteed *alignment*.

```
void ndt_aligned_free(void *ptr);
```

Free a pointer that was allocated by `ndt_aligned_calloc`. *ptr* may be `NULL`.

1.1.11 Utilities

This section contains utility functions that are meant to be used by other applications. Some of these functions are not yet in the stable API and are subject to change.

Stable API

```
char *ndt_strdup(const char *s, ndt_context_t *ctx);
```

Same as `strdup`, but uses libndtypes's custom allocators. On failure, set an error in the context and return `NULL`. The result must be deallocated using `ndt_free`.

```
char *ndt_asprintf(ndt_context_t *ctx, const char *fmt, ...);
```

Print to a string allocated by libndtypes's custom allocators. On failure, set an error in the context and return `NULL`. The result must be deallocated using `ndt_free`.

```
bool ndt_strtobool(const char *v, ndt_context_t *ctx);
```

Convert string *v* to a bool. *v* must be "true" or "false". Return *0* and set `NDT_InvalidArgumentError` if the conversion fails.

```
char ndt_strtochar(const char *v, ndt_context_t *ctx);
```

Convert string *v* to a char. *v* must have length *1*. Return *0* and set `NDT_InvalidArgumentError` if the conversion fails.

```
char ndt_strtol(const char *v, ndt_context_t *ctx);
```

Convert string *v* to a long. In case of an error, use the return value from `strtol`.

If *v* is not an integer, set `NDT_InvalidArgumentError`.

If *v* is out of range, set `NDT_ValueError`.

```
long long ndt_strtoll(const char *v, long long min, long long max, ndt_context_t␣
↪*ctx);
```

Convert string *v* to a long long.

If *v* is not an integer, set `NDT_InvalidArgumentError`.

If *v* is not in the range [*min*, *max*] , set `NDT_ValueError`.

```
unsigned long long ndt_strtoll(const char *v, long long min, long long max, ndt_
↪context_t *ctx);
```

Convert string *v* to an unsigned long long.

If *v* is not an integer, set `NDT_InvalidArgumentError`.

If *v* is not in the range [*min*, *max*] , set `NDT_ValueError`.

```
float ndt_strtof(const char *v, ndt_context_t *ctx);
```

Convert string *v* to a float.

If *v* is not an integer, set `NDT_InvalidArgumentError`.

If *v* is out of range, set `NDT_ValueError`.

```
double ndt_strtod(const char *v, ndt_context_t *ctx);
```

Convert string *v* to a double.

If *v* is not an integer, set `NDT_InvalidArgumentError`.

If *v* is out of range, set `NDT_ValueError`.

Unstable API

```
const ndt_t *ndt_dtype(const ndt_t *t);
```

Return the dtype (element type) of an array. If the argument is not an array, return *t* itself. The function cannot fail.

```
int ndt_dims_dtype(const ndt_t *dims[NDT_MAX_DIM], const ndt_t **dtype, const ndt_t␣
↪*t);
```

Extract constant pointers to the dimensions and the dtype of an array and return the number of dimensions. The function cannot fail.

```
int ndt_as_ndarray(ndt_ndarray_t *a, const ndt_t *t, ndt_context_t *ctx);
```

Convert *t* to its ndarray representation *a*. On success, return 0. If *t* is abstract or not representable as an ndarray, set an error in the context and return -1.

```
ndt_ssize_t ndt_hash(ndt_t *t, ndt_context_t *ctx);
```

Hash a type. This is currently implemented by converting the type to its string representation and hashing the string.

CHAPTER 2

Ndtypes

Python bindings for libndtypes.

2.1 ndtypes

ndtypes is a Python module based on libndtypes.

2.1.1 Quick Start

Install

Prerequisites

Python2 is not supported. If not already present, install the Python3 development packages:

```
# Debian, Ubuntu:
sudo apt-get install gcc make
sudo apt-get install python3-dev

# Fedora, RedHat:
sudo yum install gcc make
sudo yum install python3-devel

# openSUSE:
sudo zypper install gcc make
sudo zypper install python3-devel

# BSD:
# You know what to do.
```

(continues on next page)

(continued from previous page)

```
# Mac OS X:
# Install Xcode and Python 3 headers.
```

Install

If pip is present on the system, installation should be as easy as:

```
pip install ndtypes
```

Otherwise:

```
tar xvzf ndtypes-0.2.0dev3.tar.gz
cd ndtypes-0.2.0dev3
python3 setup.py install
```

Windows

Refer to the instructions in the *vcbuild* directory in the source distribution.

Examples

The libndtypes Python bindings are mostly useful in conjunction with other modules like the xnd module. While the underlying libndtypes does most of the heavy-lifting for libraries like libxnd, virtually all of this happens on the C level.

Nevertheless, some selected examples should give a good understanding of what libndtypes and ndtypes actually do:

Create types

The most fundamental operation is to create a type:

```
>>> from ndtypes import *
>>> t = ndt("2 * 3 * int64")
>>> t
ndt("2 * 3 * int64")
```

This type describes a 2 by 3 array with an int64 data type. Types have common and individual properties.

Type properties

All types have the following properties (continuing the example above):

```
>>> t.ndim
2
>>> t.datasize
48
>>> t.itemsize
8
>>> t.align
8
```

Array types have these individual properties:

```
>>> t.shape
(2, 3)

>>> t.strides
(24, 8)
```

For NumPy compatibility ndtypes displays *strides* (amount of bytes to skip). Internally, libndtypes uses steps (amount of indices to skip).

Internals

This is how to display the internal type AST:

```
>>> print(t.ast_repr())
FixedDim(
  FixedDim(
    Int64(access=Concrete, ndim=0, datasize=8, align=8, flags=[]),
    tag=None, shape=3, itemsize=8, step=1,
    access=Concrete, ndim=1, datasize=24, align=8, flags=[]
  ),
  tag=None, shape=2, itemsize=8, step=3,
  access=Concrete, ndim=2, datasize=48, align=8, flags=[]
)
```

2.1.2 Types

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

The rest of this document assumes that the `ndtypes` module has been imported:

```
from ndtypes import ndt
```

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.

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('int64')
ndt("int64")
```

All fixed-size scalars:

void	boolean	signed int	unsigned int	float[2]	complex
`void`	`bool`[1]	`int8`	`uint8`	`float16`	`complex32`
		`int16`	`uint16`	`float32`	`complex64`[3]
		`int32`	`uint32`	`float64`	`complex128`[4]
		`int64`	`uint64`	`bfloat16`	`bcomplex32`

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 an `intptr_t`:

```
>>> ndt('intptr')
ndt("int64")
```

Machine dependent aliases:

`intptr`	`intptr_t`
`uintptr`	`uintptr_t`

Chars, strings, bytes

Encodings

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

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("char('utf16')")
ndt("char('utf16')")
```

Chars

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

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

```
>>> ndt("char('ascii')")
ndt("char('ascii')")

>>> ndt("char('utf32')")
ndt("char('utf32')")

>>> ndt("char")
ndt("char('utf32')")
```

UTF-8 strings

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

```
>>> ndt("string")
ndt("string")
```

Fixed size strings

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

```
>>> ndt("fixed_string(1729)")
ndt("fixed_string(1729)")

>>> ndt("fixed_string(1729, 'utf16')")
ndt("fixed_string(1729, '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]`.

```
>>> ndt("bytes")
ndt("bytes")

>>> ndt("bytes(align=2)")
ndt("bytes(align=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:

```
>>> ndt("fixed_bytes(size=32)")
ndt("fixed_bytes(size=32)")

>>> ndt("fixed_bytes(size=128, align=8)")
ndt("fixed_bytes(size=128, align=8)")
```

References

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

```
>>> ndt("ref(int64)")
ndt("ref(int64)")

>>> ndt("ref(10 * {a: int64, b: 10 * float64})")
ndt("ref(10 * {a : int64, b : 10 * float64})")
```

Categorical type

The categorical type allows to specify subsets of types. This is implemented as a set of typed values. Types are inferred and interpreted as int64, float64 or strings. The *NA* keyword creates a category for missing values.

```
>>> ndt("categorical(1, 10)")
ndt("categorical(1, 10)")

>>> ndt("categorical(1.2, 100.0)")
ndt("categorical(1.2, 100)")

>>> ndt("categorical('January', 'August')")
ndt("categorical('January', 'August')")

>>> ndt("categorical('January', 'August', NA)")
ndt("categorical('January', 'August', NA)")
```

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.

```
>>> ndt("?complex64")
ndt("?complex64")
```

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("T")
ndt("T")

>>> ndt("10 * 16 * T")
ndt("10 * 16 * T")
```

Symbolic constructors

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

```
>>> ndt("Coulomb(float64)")
ndt("Coulomb(float64)")
```

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`
`FixedString`	`dtypes`	`fixed_strings`
`FixedBytes`	`dtypes`	`fixed_bytes`
`Fixed`	`dimension kind instances`	`fixed dimensions`

Type kinds are used in *pattern matching*.

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("(int64, float32, string)")
ndt("(int64, float32, string)")
```

Tuples can be nested:

```
>>> ndt("(bytes, (int8, fixed_string(10)))")
ndt("(bytes, (int8, fixed_string(10)))")
```

Records

Records are equivalent to tuples with named fields:

```
>>> ndt("{a: float32, b: float64}")
ndt("{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("(int32) -> int32")
ndt("(int32) -> int32")
```

This is a function type with three positional arguments:

```
>>> ndt("(int32, complex128, string) -> float64")
ndt("(int32, complex128, string) -> float64")
```

Positional-variadic

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

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

Arrays

In datashape dimension kinds[5] are part of array type declarations. Datashape supports the following dimension kinds:

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("fixed(shape=10) * uint64")
ndt("10 * uint64")

>>> ndt("10 * uint64")
ndt("10 * uint64")
```

Formally, `fixed(shape=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("10 * 25 * float64")
ndt("10 * 25 * float64")
```

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:

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

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

Variable Dimension

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

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

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.

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("M * N * float32")
ndt("M * N * float32")
```

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

```
>>> ndt("(M * N * T, N * P * T) -> M * P * T")
ndt("(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 fixed dimensions:

```
>>> ndt("10 * N * float64")
ndt("10 * N * float64")
```

Ellipsis Dimension

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

```
>>> ndt("... * float32")
ndt("... * float32")
```

Named form:

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

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

2.1.3 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 to demonstrate datashape pattern matching. The rest of this document assumes that the `ndtypes` module has been imported:

```
from ndtypes import ndt
```

General notes

`ndt` 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.

Simple example

```
>>> p = ndt("Any")
>>> c = ndt("int32")
>>> p.match(c)
True
```

Non-commutativity

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

```
>>> p = ndt("int32")
>>> c = ndt("Any")
>>> p.match(c)
False
```

Concrete matching

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

```
>>> p = ndt("int32")
>>> c = ndt("int32")
>>> p.match(c)
True
```

(continues on next page)

[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.

(continued from previous page)

```
>>> p = ndt("10 * float64")
>>> c = ndt("10 * float32")
>>> p.match(c)
False
```

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:

```
>>> p = ndt("(Any, Any)")
>>> c = ndt("(float64, int32)")
>>> p.match(c)
True
```

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:

```
>>> p = ndt("Any")
>>> c = ndt("int32")
>>> p.match(c)
True
```

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

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

Scalar

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

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

```
>>> p = ndt("Scalar")
>>> c = ndt("int32")
>>> p.match(c)
True
```

Unlike with type variables, different types match a type kind:

```
>>> p = ndt("(Scalar, Scalar)")
>>> c = ndt("(uint8, float64)")
>>> p.match(c)
True
```

FixedString

The set of all *fixed string* types.

```
>>> p = ndt("FixedString")
>>> c = ndt("fixed_string(100)")
>>> p.match(c)
True

>>> p = ndt("FixedString")
>>> c = ndt("fixed_string(100, 'utf16')")
>>> p.match(c)
True

>>> p = ndt("FixedString")
>>> c = ndt("string")
>>> p.match(c)
False
```

FixedBytes

The set of all *fixed bytes* types.

```
>>> p = ndt("FixedBytes")
>>> c = ndt("fixed_bytes(size=100)")
>>> p.match(c)
True

>>> p = ndt("FixedBytes")
>>> c = ndt("fixed_bytes(size=100, align=2)")
>>> p.match(c)
True

>>> p = ndt("FixedBytes")
>>> c = ndt("bytes(align=2)")
>>> p.match(c)
False
```

Dimension kinds

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

Fixed

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

```
>>> p = ndt("Fixed * 20 * bool")
>>> c = ndt("10 * 20 * bool")
>>> p.match(c)
True

>>> p = ndt("Fixed * Fixed * bool")
>>> c = ndt("var * var * bool")
```

(continues on next page)

(continued from previous page)

```
>>> p.match(c)
False
```

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[2], whose base cases may operate on a dtype.

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

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

Match against several dtype variables in a tuple type:

```
>>> p = ndt("T")
>>> c = ndt("(int32, int32, bool)")
>>> p.match(c)
True

>>> p = ndt("(T, T, S)")
>>> c = ndt("(int32, int64, bool)")
>>> p.match(c)
False
```

Symbolic dimensions

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

Simple symbolic match

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

```
>>> p = ndt("N * float64")
>>> c = ndt("100 * float64")
>>> p.match(c)
True
```

Symbolic+Dtypevar

Symbolic dimensions can be used in conjunction with dtype variables:

```
>>> p = ndt("N * T")
>>> c = ndt("10 * float32")
>>> p.match(c)
True
```

[2] Additional section needed.

Ellipsis match

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

```
>>> p = ndt("... * float64")
>>> c = ndt("10 * 2 * float64")
>>> p.match(c)
True

>>> p = ndt("Dim... * float64")
>>> c = ndt("10 * 20 * float64")
>>> p.match(c)
True
```

This is used in broadcasting[2].

2.1.4 Buffer protocol

ndtypes supports conversion from PEP-3118 format strings to datashape:

```
>>> from ndtypes import ndt
>>> ndt.from_format("T{<b:a:Q:b:}")
ndt("{a : <int8, b : uint64}")
```

Note that there are a couple of open issues around the buffer protocol, e.g. https://bugs.python.org/issue26746 .

CHAPTER 3

Grammar

Type grammar.

3.1 Grammar

3.1.1 Lexing

The latest version of the lexer can be found here:

lexer.l

3.1.2 Parsing

The latest version of the grammar can be found here:

grammar.y

CHAPTER 4

Releases

4.1 Releases

4.1.1 v0.2.0b2 (February 5th 2018)

Second release (beta2). This release addresses several build and packaging issues:

- The generated parsers are now checked into the source tree to avoid bison/flex dependencies and unnecessary rebuilds after cloning.
- Non-API global symbols are hidden on Linux (as long as the compiler supports gcc pragmas).
- The conda build supports separate library and Python module installs.
- Configure now has a **–without-docs** option for skipping the doc install.

4.1.2 v0.2.0b1 (January 20th 2018)

First release (beta1).

Index