HDF5 for Rust

Ivan Smirnov

Rust?..

```rust
fn main() {
    println!("Hello, world!");
}
```

"Rust is a multi-paradigm system programming language focused on safety, especially safe concurrency."

"Rust is syntactically similar to C++, but is designed to provide better memory safety while maintaining high performance."

Why Rust: memory management and memory safety

— No UB[1]: dangling/null pointers, data races, etc
— No GC: determinism without reference counting
— Ownership model and borrow checker
— Reference safety verified at compile time
— Lifetime management with syntax support

[1] In safe Rust.

Why Rust: ownership, references and borrowing

- Each value has a variable that's called *owner*
- There can only be *one* owner at a time
- The owner goes out of scope => the value is dropped
- Values may be *borrowed* via const/mut references
- References may not outlive the owner
- At any given time, you can have either:
  - *one* mutable reference
  - any number of *const* references
- Move semantics by default (unless type implements Copy)

Why Rust: modules, macros, tooling

- Proper module system with privacy layers
  (no more `#include`)
- Hygienic AST macros + procedural macros
  (no more `#define`)
- `cargo`: build system, package manager, tests, docs

Why Rust: type system

Algebraic data types with pattern matching:

```
pub enum Filter {
    Deflate(u8),
    Shuffle,
}

pub fn apply_filter(filter: Filter, id: hid_t) {
    match filter {
        Filter::Deflate(level) => apply_deflate(id, level),
        Filter::Shuffle => apply_shuffle(id),
    }
}
```

Why Rust: error handling

- No exceptions, no try/catch/finally
- No "error code is non-zero, check it yourself"
- Two types of errors:
  - Non-recoverable: panic - unwind the stack, quit
  - Recoverable: `Result<T, E>`

Result type:

```
enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

Example:

```
let f = File::open("hello.txt");  // Result<File, io::Error>

let f = match f {
    Ok(file) => file,
    Err(error) => {
        panic!("Problem opening the file: {:?}", error)
    },
};
```

Error propagation:

```rust
use std::{io, io::Read, fs::File};

fn read_file(filename: &str) -> Result<String, io::Error> {
    let f = File::open(filename);
    let mut f = match f {
        Ok(file) => file,
        Err(e) => return Err(e),
    };
    let mut s = String::new();
    match f.read_to_string(&mut s) {
        Ok(_) => Ok(s),
        Err(e) => Err(e),
    }
}
```

Simpler error propagation with ?:

```rust
use std::{io, io::Read, fs::File};

fn read_file(filename: &str) -> Result<String, io::Error> {
    let mut s = String::new();
    File::open(filename)?.read_to_string(&mut s)?;
    Ok(s)
}
```

Why Rust: traits

Traits are "pure interfaces". To implement a trait, either:

— You own the trait (it's in your crate)
— You own the type (it's in your crate)

```
trait Square {
    fn square(self) -> Self;
}

impl Square for u32 {
    fn square(self) -> Self {
        self * self
    }
}

println!("3^2 = {}", 3.square());
```

Blanket trait implementation:

```rust
use std::ops::Mul;

// Implement Square for all types that know how to
// multiply themselves by values of the same type
impl<T> Square for T
    where T: Mul<Self, Output=Self> + Copy
{
    fn square(self) -> Self {
        self * self
    }
}
```

Let's add a `.squared()` method to all iterators over types that implement `Square`, so that square operation is applied to the stream:

```rust
// `iter` is a wrapped iterator
struct SquaredIter<T> { iter: T }

// Squared "derives" from Iterator and has known size
trait Squared: Sized + Iterator {
    fn squared(self) -> SquaredIter<Self>;
}

// Implement Squared for all sized Iterators
impl<T> Squared for T
    where T: Iterator + Sized
{
    fn squared(self) -> SquaredIter<Self> {
        SquaredIter { iter: self }
    }
}
```

Finally:

```
// Let's make SquaredIter an Iterator as well
impl<T> Iterator for SquaredIter<T>
    where T: Iterator, T::Item: Square,
{
    type Item = T::Item;

    fn next(&mut self) -> Option<Self::Item> {
        match self.iter.next() {
            None => None,
            Some(item) => Some(item.square()),
        }
    }
}

...

// prints 1 4 9 16
for x in (1..).take(4).squared() {
    println!("{}", x);
}
```

Some of the built-in traits:

- Arithmetical/ops: `Mul`, `Add`, `BitwiseXor`, `Not`, etc
- Comparison: `Eq`, `PartialEq`, `Ord`, `PartialOrd`
- Printing/formatting: `Display`, `Debug`
- Copying/cloning: `Clone`, `Copy`
- Other: `Iterator`, `Deref`

Traits can be auto-derived:

```rust
#[derive(Clone, Copy, Debug, PartialEq)]
struct Foo {
    x: i32,
    y: bool,
}

// prints "Foo { x: 2, y: true }"
println!("{:?}", Foo { x: 2, y: true });
```

It's also possible to implement derive mechanism for user traits (via "procedural macros").

Why Rust: error messages

```rust
fn foo(x: &mut i32) -> i32 {
    *x * *x
}

fn main() {
    foo(4);
}
```

```
error[E0308]: mismatched types
 --> src/main.rs:6:12
  |
6 |     foo(4);
  |         ^
  |         |
  |         expected &mut i32, found integer
  |         help: consider mutably borrowing here: `&mut 4`
```

hdf5-rust

`hdf5-rust`

- GitHub repo: https://github.com/aldanor/hdf5-rust
- WIP, in development for the last few years
- Goals:
  - Build system for multiple platforms / versions (✔)
  - Rust bindings to cover all of HDF5 C API (✔)
  - Rust bitwise equivalents for HDF5-specific types (✔)
  - Automatic datatype generation for structs/enums (✔)
  - The high-level memory-safe/thread-safe interface (⌛)

Build system

- Builds on Linux / macOS / Windows: `cargo build`
- Tries it best to locate the HDF5 library (pkgconfig, brew, registry, venv/conda, system locations, etc)
- Library location can be provided manually
- Parses `H5pubconf.h` to extract library settings
- Library settings/version - available at compile-time

Crate layout

- `hdf5-sys` - C API bindings
- `hdf5-types` - type descriptors and special types
- `hdf5-derive` - type descriptor auto-deriving
- `hdf5` - the main high-level crate

C API: enums and structs

Enums and structs in `hdf5-sys` are memory-equivalent to their C counterparts:

```
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct H5F_sect_info_t {
    pub addr: haddr_t,
    pub size: hsize_t,
}

#[repr(C)]
#[derive(Copy, Clone, PartialEq, PartialOrd, Debug)]
pub enum H5FD_mpio_chunk_opt_t {
    H5FD_MPIO_CHUNK_DEFAULT = 0,
    H5FD_MPIO_CHUNK_ONE_IO = 1,
    H5FD_MPIO_CHUNK_MULTI_IO = 2,
}
```

C API: multiple versions

Single source for multiple versions (1.8.4 - 1.10.5):

```
// hdf5_sys::h5d

pub fn H5Dopen2(
    file_id: hid_t, name: *const c_char, dapl_id: hid_t,
) -> hid_t;

#[cfg(hdf5_1_10_5)]
pub fn H5Dget_num_chunks(
    dset_id: hid_t, fspace_id: hid_t, nchunks: *mut hsize_t,
) -> herr_t;
```

C API: opt-in features

```rust
// hdf5_sys::h5p

#[cfg(feature = "mpio")]
pub fn H5Pset_fapl_mpio(
    fapl_id: hid_t, comm: mpi_sys::MPI_Comm, info: mpi_sys::MPI_Info,
) -> herr_t;
```

Example of other features: "lzf", "blosc" (WIP).

C API: globals, library initialization

Some HIDs are not static and only become available after `H5open()` has been called.

```
// H5Ppublic.h

#define H5OPEN H5open(),
#define H5P_ROOT (H5OPEN H5P_CLS_ROOT_ID_g)

/* (Internal to library, do not use!  Use macros above) */
H5_DLLVAR hid_t H5P_CLS_ROOT_ID_g;
```

— In Rust, dereferencing `*hdf5_sys::h5p::H5P_ROOT` will trigger `H5open()` (behind a mutex) and then store and cache the returned HID.

— Linking pain points: `__imp_H5P_CLS_ROOT_g` on MSVC vs `H5P_CLS_ROOT_ID_g` everywhere else.

Higher-level API

- Thread-safe
- Memory-safe
- Error handling
- Reasonably easy to use
- Object hierarchy
- Immutability by default

Thread safety

- Similar to h5py: provide thread-safety without `--enable-threadsafe`
- Critical operations locked behind a reentrant mutex (e.g., anything that can modify the error stack)
- Mutexes used: `parking_lot`
- Thread-safe global registry of object IDs

Error handling

Most HDF5 calls return `hdf5::Result` which captures stack on errors:

```
/// The error type for HDF5-related functions.
#[derive(Clone)]
pub enum Error {
    /// An error occurred in the C API of the HDF5 library. Full error stack is captured.
    HDF5(ErrorStack),
    /// A user error occurred in the high-level Rust API (e.g., invalid user input).
    Internal(String),
}

pub type Result<T> = ::std::result::Result<T, Error>;
```

E.g.:

```
impl File {
    pub fn open<P: AsRef<Path>>(filename: P) -> Result<Self> { ... }
}
```

Object hierarchy via `Deref`

```
pub trait Deref {
    type Target: ?Sized;
    fn deref(&self) -> &Self::Target;
}
```

If T implements `Deref<Target = U>`, (1) values of type &T are coerced to &U, (2) T implicitly implements immutable methods from U.

```
impl Deref for File {
    type Target = Group;
    fn deref(&self) -> &Group { ... }
}
```

(1) `&File` is accepted where `&Group` is required, (2) all group methods are available in `File`, e.g. `file.link_exists("foo")`.

Type descriptor interface (H5Type)

```rust
#[derive(Clone, Debug, PartialEq, Eq)]
pub enum TypeDescriptor {
    Integer(IntSize),
    Unsigned(IntSize),
    Float(FloatSize),
    Boolean,
    Enum(EnumType),
    Compound(CompoundType),
    FixedArray(Box<TypeDescriptor>, usize),
    FixedAscii(usize),
    FixedUnicode(usize),
    VarLenArray(Box<TypeDescriptor>),
    VarLenAscii,
    VarLenUnicode,
}

pub unsafe trait H5Type: 'static {
    fn type_descriptor() -> TypeDescriptor;
}
```

Special data types (`hdf5-types`)

- Memory-equivalent Rust types compatible with HDF5 C API:
  - `FixedAscii`
  - `FixedUnicode`
  - `VarLenAscii`
  - `VarLenUnicode`
  - `VarLenArray`
  - (`[T; N]` is native Rust type)
- String types deref into `&str`
- Array types deref into `&[T]`

Deriving H5Type for user structs/enums

```rust
#[derive(hdf5::H5Type, Clone, PartialEq, Debug)]
#[repr(u8)]
pub enum Color {
    RED = 1,
    GREEN = 2,
    BLUE = 3,
}

#[derive(hdf5::H5Type, Clone, PartialEq, Debug)]
#[repr(C)]
pub struct Pixel {
    xy: (i64, i64),
    color: Color,
}
```

Example - writing to file

```rust
use ndarray::{arr1, arr2};
use self::Pixel::*;

fn main() -> hdf5::Result<()> {
    let file = hdf5::File::create("pixels.h5")?;

    let colors = file.new_dataset::<Color>().create("colors", 2)?;
    colors.write(&[RED, BLUE])?;

    let group = file.create_group("dir")?;
    let pixels = group.new_dataset::<Pixel>().create("pixels", (2, 2))?;
    pixels.write(&arr2(&[
        [
            Pixel { xy: (1, 2), color: RED },
            Pixel { xy: (3, 4), color: BLUE },
        ],
        [
            Pixel { xy: (5, 6), color: GREEN },
            Pixel { xy: (7, 8), color: RED },
        ],
    ]))?;
    Ok(())
}
```

Example - reading from file

```
fn main() -> hdf5::Result<()> {
    let file = hdf5::File::open("pixels.h5")?;

    let colors = file.dataset("colors")?;
    assert_eq!(colors.read_1d::<Color>()?, arr1(&[RED, BLUE]));

    let pixels = file.dataset("dir/pixels")?;
    assert_eq!(
        pixels.read_raw::<Pixel>()?,
        vec![
            Pixel { xy: (1, 2), color: RED },
            Pixel { xy: (3, 4), color: BLUE },
            Pixel { xy: (5, 6), color: GREEN },
            Pixel { xy: (7, 8), color: RED },
        ]
    );
    Ok(())
}
```

What's next

Already done but not merged in yet:

— LZF integration (builds with system compiler)
— Blosc interation (builds with CMake)
— LZF & Blosc filters rewritten in pure Rust
— Filter pipeline rewrite with lzf/blosc support
— Full DCPL / DAPL support
— Selections rewrite, support pointwise / regular HS (WIP)
— Unlimited selections support for VDS (WIP)

Next:

— Finish selections
— Full attributes HL support
— Support all remaining plist types
— Const generics when they land
— ...