SHARED LIBRARY OPERATORS IN DORA

Creating, linking, calling into shared libraries

Philipp Oppermann
os2edu
2023-11-17

Agenda

- Background: Executables and libraries
- Motivation: Operators in Dora
- Shared Libraries
  - C and Rust examples
  - Shared library search paths
  - Dynamic loading
- Improving Safety

Executables and Libraries

- Executables are programs with a main function
- Libraries provide callable functions to executable and other libraries
  - e.g. Rust crates can use libraries through a `[dependencies]` section in their `Cargo.toml` file
  - typical file endings of libraries are `.a`, `.lib`, `.so`, `.dll`

Executables and Libraries

- Executables are programs with a main function
- Libraries provide callable functions to executable and other libraries
  - e.g. Rust crates can use libraries through a `[dependencies]` section in their `Cargo.toml` file
  - typical file endings of libraries are `.a`, `.lib`, `.so`, `.dll`

Example: Use `readelf` to read type of ELF files:

```
❯ readelf --file-header /usr/bin/ls | grep "Type"
  Type:                              DYN (Position-Independent Executable file)
❯ readelf --file-header /usr/lib32/libc.so.6 | grep "Type"
  Type:                              DYN (Shared object file)
```

C Example

- `executable.c`:

```c
#include <stdio.h>
void main() {
    printf("hello world");
}
```

  - create executable: **`gcc -o executable executable.c`**
  - run executable: `./executable` → prints "hello world"

C Example

- `executable.c`:

```
#include <stdio.h>
void main() {
    printf("hello world");
}
```

  - create executable: **`gcc -o executable executable.c`**
  - run executable: `./executable` → prints "hello world"

- `sum.c`:

```
int sum(const int a, const int b) {
    return a + b;
}
```

  - try to build as executable using `gcc -o sum sum.c` → error: undefined reference to `main'
  - create object file: **`gcc -c sum.c`**
  - then create static library: **`ar r libsum.a sum.o`**

C Example: Using static library

- Use `libsum.a` in `executable.c`:
  - define header file with function signature in `sum.h`:

    ```c
    int sum(const int a, const int b);
    ```

  - include `sum.h` in executable:

    ```c
    #include <stdio.h>
    #include "sum.h"

    void main() {
        printf("hello world %i", sum(40, 2));
    }
    ```

  - create executable, linking `libsum.a`: **`gcc -o executable executable.c -L. -lsum`**
  - run executable: `./executable` → prints "hello world 42"

C Example: Using static library

- Use `libsum.a` in `executable.c`:
  - define header file with function signature in `sum.h`:

    ```
    int sum(const int a, const int b);
    ```

  - include `sum.h` in executable:

    ```
    #include <stdio.h>
    #include "sum.h"

    void main() {
        printf("hello world %i", sum(40, 2));
    }
    ```

  - create executable, linking `libsum.a`: **`gcc -o executable executable.c -L. -lsum`**
  - run executable: `./executable` → prints "hello world 42"

- **Note:** The argument order is important: `-l` needs to come after `executable.c`
  - else **undefined reference to `sum'** error, as linker only looks for symbols that are referenced

Rust Example

- Executable using cargo:

```
> cargo new --bin executable
 Created binary (application) 'executable' package
> cd executable
> cargo build
   Compiling executable v0.1.0 (/../executable)
    Finished dev [unoptimized + debuginfo] target(s) in 0.13s
> target/debug/executable
Hello, world!
```

Rust Example

- Executable using cargo:

```
> cargo new --bin executable
 Created binary (application) 'executable' package
> cd executable
> cargo build
   Compiling executable v0.1.0 (/../executable)
    Finished dev [unoptimized + debuginfo] target(s) in 0.13s
> target/debug/executable
Hello, world!
```

- Library using cargo:
  - create: `cargo new --lib sum`
  - modify `src/lib.rs`:

```
pub fn sum(left: isize, right: isize) -> isize {
    left + right
}
```

  - build using `cargo build` → creates `target/debug/libsum.rlib` library

Rust Example: Using Rust library

- Using `libsum.rlib` in `executable` crate:
  - add dependency in `executable/Cargo.toml`:

```
[dependencies]
sum = { path = "../sum" }
```

  - use `sum` function in `executable/src/main.rs`:

```
use sum::sum;

fn main() {
    println!("Hello, world! {}", sum(40, 2));
}
```

  - rebuild using `cargo build`
  - run `target/debug/executable` → outputs *"Hello, world! 42"*

Note: `rlib` libraries are Rust-specific and might change over time

Operators in Dora

- Dora operators are components that can be executed by the Dora runtime
  - act as nodes in the dataflow graph → receive inputs and send outputs
  - a single Dora runtime process can run multiple operators concurrently
- Operators are **libraries** that implement a specific template
  - e.g. they need to provide an **`on_event`** function

Operators in Dora

- Dora operators are components that can be executed by the Dora runtime
  - act as nodes in the dataflow graph → receive inputs and send outputs
  - a single Dora runtime process can run multiple operators concurrently
- Operators are **libraries** that implement a specific template
  - e.g. they need to provide an **`on_event`** function

Challenge:

- We don't want to recompile the Dora runtime when loading operators
- `.a` and `.rlib` libraries need to be included at compile time

→ we need to use a different library format that can be loaded without recompiling

Python Libraries

- Python is an interpreted language → no precompiled code
- Rust-based Dora runtime can import Python operators using `pyo3` crate, e.g.:

```
Python::with_gil(|py| {
    let sum = PyModule::from_code(py, "
        def sum(x, y):
            return x + y
    ", "sum.py", "sum")?;

    let result: i32 = sum.getattr("sum")?.call2((40, 2))?.extract()?;
    assert_eq!(result, 42);

    Ok(())
})
```

- `pyo3` returns errors when operator has wrong format, e.g. is missing a required function

→ we want something similar for compiled languages: **shared libraries**

Shared libraries

- Approach
  - Precompile library as before, but include additional metadata
  - Compile executable against stub library that describes the template
  - When executable is loaded, combine compiled executable and library using the metadata
- Supported and used on all major OS platforms, but different format:
  - Linux: `.so`
  - Windows: `.dll`
  - MacOS: `.dylib`

Shared libraries

- Approach
  - Precompile library as before, but include additional metadata
  - Compile executable against stub library that describes the template
  - When executable is loaded, combine compiled executable and library using the metadata
- Supported and used on all major OS platforms, but different format:
  - Linux: `.so`
  - Windows: `.dll`
  - MacOS: `.dylib`
- Our example Rust `executable` already has some shared library dependencies:

```
❯ readelf --dynamic target/debug/executable | grep "Shared"
0x0000000000000001 (NEEDED)             Shared library: [libgcc_s.so.1]
0x0000000000000001 (NEEDED)             Shared library: [libc.so.6]
0x0000000000000001 (NEEDED)             Shared library: [ld-linux-x86-64.so.2]
```

Example: C Shared library

- We reuse the same `executable.c`, `sum.h`, and `sum.c` as before

```c
int sum(const int a, const int b) {
    return a + b;
}
```

- Build steps:
  - create position-independent object file for `sum`: **`gcc -c -fPIC sum.c`**
  - then create shared library: **`gcc -shared sum.o -o shared/libsum.so`**
  - link against the shared library: **`gcc -o executable executable.c -Lshared -lsum`**

Example: C Shared library

- We reuse the same `executable.c`, `sum.h`, and `sum.c` as before

```c
int sum(const int a, const int b) {
    return a + b;
}
```

- Build steps:
  - create position-independent object file for `sum`: **`gcc -c -fPIC sum.c`**
  - then create shared library: **`gcc -shared sum.o -o shared/libsum.so`**
  - link against the shared library: **`gcc -o executable executable.c -Lshared -lsum`**

- Try running it:

```
❯ ./executable
./executable: error while loading shared libraries: libsum.so: cannot open shared object file: No such file or directory
```

→ shared library is required for running the executable, but not found

Shared Library Search Paths

Lookup of shared libraries depends on operating system:

- On Linux, the linker tries the following directories:
  1. directories listed in the `LD_LIBRARY_PATH` environment variable
  2. paths specified in the executable itself (through an `rpath` attribute)
  3. system search paths
- MacOS is similar, but it uses an env variable called `DYLD_LIBRARY_PATH`
- On Windows, the behavior is more complex. Some differences are:
  - no separate environment variable, instead the normal `PATH` is searched
  - the folder containing the executable is searched
  - the *current working directory* is searched too by default (note: this can be dangerous)

Fixing the C Shared Library Example

- Problem: The `libsum.so` library lives in the `shared` subdirectory
    - not in the default search path

Fixing the C Shared Library Example

- Problem: The `libsum.so` library lives in the `shared` subdirectory
  - not in the default search path
- Solution 1: Set `LD_LIBRARY_PATH` environment variable:

```
> LD_LIBRARY_PATH=shared ./executable
hello world 42
```

Fixing the C Shared Library Example

- Problem: The `libsum.so` library lives in the `shared` subdirectory
  - not in the default search path
- Solution 1: Set `LD_LIBRARY_PATH` environment variable:

```
› LD_LIBRARY_PATH=shared ./executable
hello world 42
```

- Solution 2: Move library to system search path → not recommended

Fixing the C Shared Library Example

- Problem: The `libsum.so` library lives in the `shared` subdirectory
  - not in the default search path
- Solution 1: Set `LD_LIBRARY_PATH` environment variable:

```
> LD_LIBRARY_PATH=shared ./executable
hello world 42
```

- Solution 2: Move library to system search path → not recommended
- Solution 3: Set `rpath` attribute when building `executable`:

```
> gcc -o executable executable.c -Lshared -lsum -Wl,-rpath shared
> ./executable
hello world 42⏎
```

Example: Rust Shared Library

- Steps:
  - `cargo new --lib sum2`
  - modify `sum2/src/lib.rs`:

```rust
pub extern "C" fn sum(left: isize, right: isize) -> isize {
    left + right
}
```

  - set the `crate-type` in `Cargo.toml`:

```toml
    [lib]
    crate-type = ["cdylib"]
```

  - run `cargo build` to create `libsum2.so` in `target/debug`

Example: Rust Shared Library

- Steps:
    - `cargo new --lib sum2`
    - modify `sum2/src/lib.rs`:

```rust
pub extern "C" fn sum(left: isize, right: isize) -> isize {
    left + right
}
```

    - set the `crate-type` in `Cargo.toml`:

```toml
    [lib]
    crate-type = ["cdylib"]
```

    - run `cargo build` to create `libsum2.so` in `target/debug`
- Created shared library *should be* C-compatible, but linking with `executable` somehow fails:

```
> gcc -o executable executable.c -Lsum2/target/debug -lsum2
/usr/bin/ld: /tmp/ccf95YLL.o: in function `main':
executable.c:(.text+0x13): undefined reference to 'sum'
```

Debug Rust Shared Library Example

- Print available symbols using `nm`:

```
> nm -gD sum/target/debug/libsum.so
           w __cxa_finalize@GLIBC_2.2.5
           w __gmon_start__
           w _ITM_deregisterTMCloneTable
           w _ITM_registerTMCloneTable
```

→ no `sum` function

Debug Rust Shared Library Example

- Print available symbols using `nm`:

```
❯ nm -gD sum/target/debug/libsum.so
                 w __cxa_finalize@GLIBC_2.2.5
                 w __gmon_start__
                 w _ITM_deregisterTMCloneTable
                 w _ITM_registerTMCloneTable
```

→ no `sum` function

- Reasons:
  - Rust functions are not exported as symbols by default
  - Rust function names are *mangled* by default to prevent name conflicts

Debug Rust Shared Library Example

- Print available symbols using `nm`:

```
> nm -gD sum/target/debug/libsum.so
                 w __cxa_finalize@GLIBC_2.2.5
                 w __gmon_start__
                 w _ITM_deregisterTMCloneTable
                 w _ITM_registerTMCloneTable
```

→ no `sum` function

- Reasons:
  - Rust functions are not exported as symbols by default
  - Rust function names are *mangled* by default to prevent name conflicts
- Solution: Set `#[no_mangle]` attribute:

```
#[no_mangle]
pub extern "C" fn sum(left: isize, right: isize) -> isize { ... }
```

→ after a `cargo build`, the `gcc` link error is now fixed

Link Shared Library from Rust

- Rust does not support header files
  - instead, use `extern` block to specify dependency on external `sum` function:

    ```rust
    // in executable/src/main.rs
    extern "C" {
        fn sum(a: isize, b: isize) -> isize;
    }
    ```

  - the `bindgen` crate allows auto-generating this `extern` block based on header files

Link Shared Library from Rust

- Rust does not support header files
  - instead, use `extern` block to specify dependency on external `sum` function:

    ```rust
    // in executable/src/main.rs
    extern "C" {
        fn sum(a: isize, b: isize) -> isize;
    }
    ```

  - the `bindgen` crate allows auto-generating this `extern` block based on header files
- Calling external functions is unsafe, as the Rust compiler cannot guarantee their safety:

  ```rust
  println!("Hello, world! {}", unsafe { sum(40, 2) });
  ```

Link Shared Library from Rust

- Rust does not support header files
    - instead, use `extern` block to specify dependency on external `sum` function:

    ```
    // in executable/src/main.rs
    extern "C" {
        fn sum(a: isize, b: isize) -> isize;
    }
    ```

    - the `bindgen` crate allows auto-generating this `extern` block based on header files
- Calling external functions is unsafe, as the Rust compiler cannot guarantee their safety:

```
println!("Hello, world! {}", unsafe { sum(40, 2) });
```

- Pass linker flags via new `build.rs` build script:

```
fn main() {
    println!("cargo:rustc-link-search=native=../sum2/target/debug");
    println!("cargo:rustc-link-lib=dylib=sum2");
}
```

→ `cargo build` now links to the shared library

Shared libraries: Pros and Cons

Advantages:

- Avoid duplication of common libraries
  - Both on disk and after loading in RAM
  - Example: `libLLVM-15.so` is over 100MiB
- Security: Updating a shared library fixes all executables
  - Example: SSL libraries
- Platform-specific libraries
  - Example: libc

Drawbacks:

- Program might not work when when library is missing or at wrong version
- Distribution is more difficult

Shared Library Operators in Dora

Challenges:

- The dora runtime is compiled ahead of time
  - we don't have access to the operator at this point → workaround possible using stub library
- We want to load multiple operators → name conflicts
  - e.g., there can only be one `on_event` function
- The dora runtime should be able to recover from load errors
  - operators in an incompatible format should not bring the runtime down
  - load errors should result in useful error messages

→ use dynamic loading to link library manually at runtime

Dynamic Loading

- Don't link to the shared library directly
- Instead, load library at runtime using system functions
    - on Linux and MacOS: `dlopen`, `dlsym`, `dlclose`
    - on Windows: `LoadLibraryExW`, `GetProcAddress`, `FreeLibrary`

Dynamic Loading

- Don't link to the shared library directly
- Instead, load library at runtime using system functions
  - on Linux and MacOS: `dlopen`, `dlsym`, `dlclose`
  - on Windows: `LoadLibraryExW`, `GetProcAddress`, `FreeLibrary`
- Advantages:
  - more resilient → better error messages
  - no name conflicts → multiple operators can be loaded simultaneously
  - no stubs needed for building Dora runtime
- Drawbacks: less convenient, more manual work necessary

Example: Dynamic Loading using `dlopen`

```c
#include <stdio.h>
#include <dlfcn.h>

int main(void) {
    void *handle = dlopen("libsum.so", RTLD_LAZY);
    if (!handle) { return -1; }

    int (*sum)(int, int) = (int (*)(int, int)) dlsym(handle, "sum");
    if (dlerror() != NULL) { return -1; }

    printf("%d\n", (*sum)(40, 2));

    dlclose(handle);
    return 0;
}
```

Dynamic Loading in Rust

Use the cross-platform `libloading` crate:

```rust
let result = unsafe {
    let lib = libloading::Library::new("shared/libsum.so")?;
    let func: libloading::Symbol<unsafe extern fn(isize, isize) -> isize> = lib.get(b"sum")?;
    func(40, 2)
};
assert_eq!(result, 42);
```

Dynamic Loading in Rust

Use the cross-platform `libloading` crate:

```rust
let result = unsafe {
    let lib = libloading::Library::new("shared/libsum.so")?;
    let func: libloading::Symbol<unsafe extern fn(isize, isize) -> isize> = lib.get(b"sum")?;
    func(40, 2)
};
assert_eq!(result, 42);
```

Safety:

- We get nice error messages when the library or the requested symbol don't exist
  - e.g. "Error: shared/libsum.so: cannot open shared object file: No such file or directory"
  - e.g. "Error: shared/libsum.so: undefined symbol: sum"
- But there is no type checking → the function signature is not verified
  - that's the reason why all of the above code is considered `unsafe`

Improving Safety

- Provide header files to check operator signature
    - Still very easy to cause undefined behavior accidentally

Improving Safety

- Provide header files to check operator signature
  - Still very easy to cause undefined behavior accidentally
- Use the `abi_stable` crate
  - designed for Rust-to-Rust FFI
  - generates `static` variables with type and layout information → verify on load
  - provides FFI-compatible wrappers for standard library types
  - custom types can derive the `StableAbi` trait
  - even provides trait object support through `DynTrait` (for a selection of traits)

Improving Safety

- Provide header files to check operator signature
  - Still very easy to cause undefined behavior accidentally
- Use the `abi_stable` crate
  - designed for Rust-to-Rust FFI
  - generates `static` variables with type and layout information → verify on load
  - provides FFI-compatible wrappers for standard library types
  - custom types can derive the `StableAbi` trait
  - even provides trait object support through `DynTrait` (for a selection of traits)
- `abi_stable` guards against accidental incompatibilities, but not against malicious inputs
  - → isolate untrusted shared library operators into separate runtime process

Future Work: WebAssembly Operators

- WebAssembly is a sandboxed executable format designed for the web
  - even safer than Python (no dangerous FFI calls possible)
- Many languages can be compiled to WebAssembly, including Rust, C, etc.
- There are multiple mature Rust-based WebAssembly runtimes available
  - Wasmtime based on Cranelift
  - WasmEdge based on LLVM

→ we plan to add support for WebAssembly operators to Dora in the future

Summary

- Creating shared libraries in C and Rust
    - Different shared library search algorithms on Linux and Windows
- Advantages and drawbacks of shared libraries
    - avoid duplication, faster security fixes, use platform-specific code
    - missing or incompatible libraries prevent running of executable
- Dynamic loading
    - using `dlopen`/`LoadLibraryExW` or the `libloading` crate
    - enables independent linking, better error messages, and avoids name conflicts
    - drawback: manual type specifications needed → mistakes can cause undefined behavior
- Improving safety using `abi_stable` crate → future work: sandboxed WebAssembly operators