On the path of better interoperability with Rust!

https://github.com/yvan-sraka/hs-bindgen

https://functional.cafe/@yvan

Context

At IOG we maintain large Haskell codebases and we would like to interface them with some libraries written in Rust.

Why FFI (Foreign Function Interface)?

Solving the interoperability problem means:

1. designing a protocol that allows two codes written with different languages and using different runtime systems to communicate ;
2. designing tools and methods to build, to bundle, and to distribute such polyglot code bases (what developers fear most).

As our main criterion is performance, we want a solution with a minimal overhead. In particular, we want to avoid the use of any solution that relies on syscalls (like I/Os) and on costly data (de)serialization.

FFI looks like the right choice: no syscall, a foreign function call just behaves as a jump in memory.

Rust ecosystem of integration with other PL

So, what's lacking in Haskell ecosystem? Let's take a look at what kind of integration other languages offer with Rust:

- From **C** to Rust `rust-bindgen` ;
- From Rust to **C** `c-bindgen` and Rust to **ECMAScript**, `wasm-bindgen` ;
- Both from and to Rust with **C++** `cxx` and **Python** `PyO3`.

This list isn't exhaustive but give you a hint, all these projects are about generating bindings (bindgen)!

Why bindgen (bindings code generation)?

Let's sum it up by: *"A good FFI is an FFI that you don't write ..."*

FFI are like a blind spot in your type system. Writing them manually is both frankly painful and really dangerous, as your compiler will not warn you about non-matching interfaces.

Binding generation comes to the rescue by considerably reducing the room for human errors. As a bonus, it also makes maintainers' life easier thanks to a smaller and more readable code base.

A minimal example ...

```rust
//! src/main.rs in a `greetings` crate

use hs_bindgen::*;

#[hs_bindgen(hello :: CString -> IO ())]
fn hello(name: &str) {
    println!("Hello, {name}!");
}
```

... after Rust macro expansion:

```rust
use hs_bindgen::*;

fn hello(name: &str) {
    println!("Hello, {name}!");
}

#[no_mangle] // Mangling makes symbol names more difficult
             // to predict. We disable it to ensure that
             // the resulting symbol is really `__c_hello`
extern "C" fn __c_hello(__0: *const core::ffi::c_char) {
    // `traits` module is `hs-bindgen::hs-bindgen-traits`
    // n.b. do not forget to import it, e.g., with:
    //      use hs-bindgen::*
    traits::ReprC::from(
        hello(traits::ReprRust::from(__0))
    )
}
```

Why use C ABI (Application Binary Interface)?

First, GHC currently doesn't know anything about Rust calling convention, while it does about C's one: C's calling convention is the *lingua franca* of `rustc`/`ghc`.

Additionally, the Rust ABI (call-convention and types memory layout) isn't stable. That means that it's specified internally but could be broken by any `rustc` minor release, building a software on top of it is by definition a "hack" ... If we think it's worth it, we would have to perform our bindgen against a given `rustc` version (and that would be really laborious to maintain). So, do not fear the C ABI because, at least, it is stable!

Why implement bindgen as a Rust macro?

Binding code generation could have been achieved using an external tool, e.g., `cbindgen` parses Rust code (before macro expansion) and deduces C function signatures.

But instead we decided to define a custom macro (like `cxx`, `wasm-bindgen`, and `PyO3` do), and so we require the user to depend on a custom crate.

The reason is that we want generated bindings to always match the source code used for their generation. By using a macro we enforce binding generation during the build process and bindings can't get out-of-sync.

Haskell code generated:

```haskell
-- This file was generated by `hs-bindgen` crate and
-- contains C FFI bindings wrappers for every Rust
-- function annotated with `#[hs_bindgen]`

{-# LANGUAGE ForeignFunctionInterface #-}

module Greetings (hello) where

import Data.Int
import Data.Word
import Foreign.C.String
import Foreign.C.Types
import Foreign.Ptr

foreign import ccall safe "__c_hello"
    hello :: CString -> IO ()
```

... (re)look at Rust code expansed:

```rust
use hs_bindgen::*;

fn hello(name: &str) {
    println!("Hello, {name}!");
}

#[no_mangle] // Mangling makes symbol names more difficult
             // to predict. We disable it to ensure that
             // the resulting symbol is really `__c_hello`
extern "C" fn __c_hello(__0: *const core::ffi::c_char) {
    // `traits` module is `hs-bindgen::hs-bindgen-traits`
    // n.b. do not forget to import it, e.g., with:
    //      use hs-bindgen::*
    traits::ReprC::from(
        hello(traits::ReprRust::from(__0))
    )
}
```

Traits! (Rust typeclasses)

Wrapping user types by these traits have several benefits:

- Unsupported types are nicely reported as *"the trait `ReprRust<T>` is not implemented for `U`"* error (that suggest other types that the trait implement to the user);
- The user can extensively always implement these traits for arbitrary types ;
- Provided traits implementation for `std` types take care of memory management ;
- Traits improve a lot of ergonomics by implicitly and safely casting a given type to an FFI-safe one.

Digression on memory management and GC

The memory management strategy is freeing the value is the role of the receiver (which has "ownership" of it). This means that values returned by Rust functions aren't `dropped` by Rust but rather should be `freed` on the Haskell side!

`hs-bindgen` generates `safe` Haskell foreign imports by default! You can still generate `unsafe` bindings simply by prefixing a function name like `#[hs_bindgen(unsafe NAME :: TYPE)]` in Rust attribute macro. [1]

[1] To go further read "FFI safety and GC" by Fraser Tweedale or GHC's users guide to understand the differences between Haskell unsafe/safe keywords.

Developer experience

`cargo-cabal`[2] is a CLI tool that helps you, in one simple command, turn a Rust crate into a Haskell Cabal library!

What `cargo-cabal` actually does is:

- Ask the user to add `crate-type = ["staticlib"]` (or `"cdylib"`, dynamic libraries require an extra `build.rs` file that is generated by `cargo-cabal`) to their `Cargo.toml` file;
- Generate a custom `X.cabal` linking `rustc` output as `extra-librairies`, and either a (`naersk` and `haskell.nix` based) `flake.nix` or a `Setup.lhs`.

[2] https://github.com/yvan-sraka/cargo-cabal

What's next?

RFCs:

- Build Drivers for cabal[3] ;
- Experimental feature gate proposal `interoperable_abi`[4].

[3] https://github.com/haskell/cabal/issues/7906
[4] https://github.com/rust-lang/rust/pull/105586

Q/A