An Idris Foreign Function Interface to OCaml

Robert Atkey ⟨robert.atkey@strath.ac.uk⟩
Ioan Luca ⟨ioan.luca7@gmail.com⟩

1 Introduction

Idris is a strict functional language with dependent types intended for practical use [1]. Unfortunately, due to the relatively small size of the Idris user community, the ecosystem of available libraries for practical programming is limited. To remedy this, we have constructed a Foreign Function Interface (FFI) for Idris to call OCaml libraries. We have extended Dolan's backend [2] for the Idris compiler that generates Malfunction (a concrete syntax for the OCaml compiler's `Lambda` intermediate language) with the ability to call arbitrary OCaml libraries. Using this facility[1], we have been able to call higher order functions in OCaml's `List` library, use the `Cohttp` library to make a web server with the callback function written in Idris using the `Lwt` library, and to construct a simple Mirage unikernel written in Idris. In the latter case, we needed to export an implementation of an OCaml functor from Idris to fit with Mirage's functorised design.

In the talk, we will report on and demo the modifications we have made to Dolan's Malfunction backend, and the progress we have made in expressing the complex OCaml type system in Idris.

Idris's dependent types offer an expressive language for describing types, which we have used to expose a simple fragement of the module system in Idris. However, ergonomic expression of module types, that allow the use of names for looking up structure entries is not yet possible (only type safe lookup by index is possible). The nominal type systems of both languages also present problems when attempting to use structural descriptions of data types.

2 A Mirage Unikernel in Idris

We demonstrate our FFI by writing a simple Mirage Unikernel in Idris. We will fulfil the following functor signature that takes an implementation of the time service (which exposes a single function : `sleep : int64 -> unit Lwt.t`) and produces a single function `start` that forms the main function of the unikernel.

```
module Hello
    (Time : Mirage_time_lwt.S) : sig
  val start : 'a -> unit Lwt.t
end
```

We represent this functor type in Idris as a value of type `Type`: a pure function that takes an `OCamlModule` to an `OCamlModule`, parameterised by the types of the entries:

```
HelloSig : Type
HelloSig =
 OCamlModule
   [OCamlFn (Int64 -> OCaml_IO (Lwt ()))]
 ->
 OCamlModule
   [OCamlFn (() -> OCaml_IO (Lwt ()))]
```

Note that we have to write the types of the entries in the signatures in order, and below we will reference them by index. Future work is to be able to refer to structure entries by name, and to generate these types directly from `cmi` files. We have also artificially monomorphised type of `start` in the Idris signature: representing polymorphic structure items is currently future work, and will rely on the precise details of Idris's erasure features.

Finally, we implement the unikernel in Idris. It prints to the console four times with a second's delay between each. The `sleep` function from the time service is used to implement the delay.

```
Hello : HelloSig
Hello time =
    mkModule (Step (mkOCamlFn start) Stop)
  where
    sleep : Int64 -> OCaml_IO (Lwt ())
    sleep = unOCamlFn $ modGet 0 time

    loop : Int -> OCaml_IO (Lwt ())
    loop 0 = lwtUnit
    loop n = do
      putStrLn' "Idris Unikernel Hello!"
      lwtThread <- sleep (of_sec 1)
      lwtThread `lwtBind`
         (\ _ => loop (n - 1))

    start : () -> OCaml_IO (Lwt ())
    start _ = loop 4
```

We have implemented functions `mkModule` and `modGet` that build and project from modules respectively. Idris type checks both functions to make sure that they match the declared signatures. Idris is a pure language, so effectful functions are marked with the `OCaml_IO` monad, which makes writing functions in the `Lwt` monad layered on top somewhat awkward. The functions `mkOcamlFn` and `unOCamlFn` translate between effectful Idris functions and OCaml functions, inserting the dummy "world" arguments that Idris uses to implement I/O in a pure language. As with the module primitives, outside the internals of the FFI implementation, Idris's typing ensures that we insert these coercions in the right places.

[1] https://github.com/ioanluca/real-world-idris

3 Future Work

Currently, the FFI is a proof of concept. We are able to wrap simple OCaml functions, including higher order ones, and write simple modules, including functors. However, polymorphism, both at the core and module level are challenging. We can "fake" it in some cases by inserting "`believe_me`" casts at the right places, but this is fragile. Representing OCaml's rich module language in Idris is also challenging. We conjecture that the techniques used in the F-ing modules encoding may help [3]. Another requirement is the wrapping of algebraic datatypes. We currently have hard coded wrappings of lists and the `Maybe`/`option` type. We have experimented with codes for algebraic datatypes in Idris, but this interacts poorly with Idris's nominal datatypes. Idris's support for type providers [4] may be useful here.

References

[1] Edwin Brady. *Type-driven Development with Idris*. Manning Publications Company, 2017.

[2] Stephen Dolan. Malfunctional programming. https://www.cl.cam.ac.uk/~sd601/papers/malfunction.pdf, 2016.

[3] Andreas Rossberg, Claudio V. Russo, and Derek Dreyer. F-ing modules. *J. Funct. Program.*, 24(5):529–607, 2014.

[4] The Idris Community. Type Providers in Idris. http://docs.idris-lang.org/en/latest/guides/type-providers-ffi.html, 2017.