Idris Tutorial Series

Release 0.10.1

The Idris Community

February 27, 2016

Contents

Tutorials submitted by community members.

CHAPTER 1

Type Providers in Idris

Type providers in Idris are simple enough, but there are a few caveats to using them that it would be worthwhile to go through the basic steps. We also go over foreign functions, because these will often be used with type providers.

1.1 The use case

First, let's talk about *why* we might want type providers. There are a number of reasons to use them and there are other examples available around the net, but in this tutorial we'll be using them to port C's `struct stat` to Idris.

Why do we need type providers? Well, Idris's FFI needs to know the types of the things it passes to and from C, but the fields of a `struct stat` are implementation-dependent types that cannot be relied upon. We don't just want to hard-code these types into our program... so we'll use a type provider to find them at compile time!

1.2 A simple example

First, let's go over a basic usage of type providers, because foreign functions can be confusing but it's important to remember that providers themselves are simple.

A type provider is simply an IO action that returns a value of this type:

```
data Provider a = Provide a | Error String
```

Looks familiar? `Provider` is just `Either a String`, given a slightly more descriptive name.

Remember though, type providers we use in our program must be IO actions. Let's write a simple one now:

```
module Provider
-- Asks nicely for the user to supply the size of C's size_t type on this
-- machine
getSizeT : IO (Provider Int)
getSizeT = do
  putStrLn "I'm sorry, I don't know how big size_t is. Can you tell me, in bytes?"
  resp <- getLine
  case readInt resp of
     Just sizeTSize => return (Provide sizeTSize)
     Nothing => return (Error "I'm sorry, I don't understand.")
-- the readInt function is left as an exercise
```

We assume that whoever's compiling the library knows the size of `size_t`, so we'll just ask them! (Don't worry, we'll get it ourselves later.) Then, if their response can be converted to an integer, we present `Provide sizeTSize` as the result of our IO action; or if it can't, we signal a failure. (This will then become a compile-time error.)

Now we can use this IO action as a type provider:

```
module Main
-- to gain access to the IO action we're using as a provider
import Provider

-- TypeProviders is an extension, so we'll enable it
%language TypeProviders

-- And finally, use the provider!
-- Note that the parentheses are mandatory.
%provide (sizeTSize : Int) with getSizeT

-- From now on it's just a normal program where `sizeTSize` is available
-- as a top-level constant
main : IO ()
main = do
  putStr "Look! I figured out how big size_t is! It's "
  putStr (show sizeTSize)
  putStr " bytes!"
```

Yay! We... asked the user something at compile time? That's not very good, actually. Our library is going to be difficult to compile! This is hardly a step up from having them edit in the size of `size_t` themselves!

Don't worry, there's a better way.

1.3 Foreign Functions

It's actually pretty easy to write a C function that figures out the size of `size_t`:

```
int sizeof_size_t() { return sizeof(size_t); }
```

(Why an int and not a `size_t`? The FFI needs to know how to receive the return value of this function and translate it into an Idris value. If we knew how to do this for values of C type `size_t`, we wouldn't need to write this function at all! If we really wanted to be safe from overflow, we could use an array of multiple integers, but the SIZE of `size_t` is never going to be a 65535 byte integer.)

So now we can get the size of `size_t` as long as we're in C code. We'd like to be able to use this from Idris. Can we do this? It turns out we can.

1.3.1 `mkForeign`

With mkForeign, we can turn a C function into an IO action. It works like this:

```
getSizeT : IO Int
getSizeT = mkForeign (FFun "sizeof_size_t" [] FInt)
```

Pretty simple. `mkForeign` takes a specification of what function it needs to call, and we construct this specification with `FFun`. And `FFun` just takes a name, a list of argument types (we have none), and a return type.

One thing you might want to note: the return type we've specified is `FInt`, not `Int`. That's because `Int` is an idris type and C functions don't return idris types. `FInt` is not an idris type, but is just the representation of the type of a C int. It tells the compiler "Treat the return value of this C function like it's a C int, and when you pass it back into Idris, convert it to an Idris int."

1.3.2 Caveats of mkForeign

First and foremost: `mkForeign` is not actually a function. It is treated specially by the compiler, and there are certain rules you need to follow when using it.

- Rule 1: the name string must be a literal or constant

This does not work:

```
intIntToInt : String -> Int -> Int -> IO Int
intIntToInt name = mkForeign (FFun name [FInt, FInt] FInt)
```

You'll just have to bite the bullet and write out the whole `mkForeign` and `FFun` expression each time.

- Rule 2: the "call" to `mkForeign` must be fully applied

This just means that every argument appearing in the list of argument types must be applied wherever you write `mkForeign`. The arguments don't have to be literals or even known at compile time; they just have to be there. For example, if we have `strlen : String -> IO Int`, then this is fine:

```
strlen str = mkForeign (FFun "strlen" [FString] FInt) str
```

but this is not fine:

```
strlen = mkForeign (FFun "strlen" [FString] FInt)
```

Note that this only applies to places where you literally typed `mkForeign`. Once you've defined it, `strlen` is just a normal function returning an IO action, and it doesn't need to be fully applied. This is okay:

```
lengths : IO [Int]
lengths = mapM strlen listOfStrings
```

1.3.3 Running foreign functions

This is all well and good for writing code that will typecheck. To actually run the code, we'll need to do just a bit more work. Exactly what we need to do depends on whether we want to interpret or compile our code.

1.3.4 In the interpreter

If we want to call our foreign functions from interpreted code (such as the REPL or a type provider), we need to dynamically link a library containing the symbols we need. This is pretty easy to do with the `%dynamic` directive:

```
%dynamic "./filename.so"
```

Note that the leading "./" is important: currently, the string you provide is interpreted as by `dlopen()`, which on Unix does not search in the current directory by default. If you use the "./", the library will be searched for in the directory from which you run idris (*not* the location of the file you're running!). Of course, if you're using functions from an installed library rather than something you wrote yourself, the "./" is not necessary.

1.3.5 In an executable

If we want to run our code from an executable, we can statically link instead. We'll use the `%include` and `%link` directives:

```
%include C "filename.h"
%link C "filename.o"
```

Note the extra argument to the directive! We specify that we're linking a C header and library. Also, unlike `%dynamic`, these directives search in the current directory by default. (That is, the directory from which we run idris.)

1.4 Putting it all together

So, at the beginning of this article I said we'd use type providers to port `struct stat` to Idris. The relevant part is just translating all the mysterious typedef'd C types into Idris types, and that's what we'll do here.

First, let's write a C file containing functions that we'll bind to.

```
/* stattypes.c */
int sizeof_dev_t() { return sizeof(dev_t); }
int sizeof_ino_t() { return sizeof(ino_t); }
/* lots more functions like this */
```

Next, an Idris file to define our providers:

```
-- Providers.idr
module Providers

%dynamic "./stattypes.so"

sizeOfDevT : IO Int
sizeOfDevT = mkForeign (FFun "sizeof_dev_t" [] FInt)
{- lots of similar functions -}

-- now we have an integer, but we want a Provider FTy
-- since our sizeOf* functions are ordinary IO actions, we
-- can just map over them.
bytesToType : Int -> Provider FTy
bytesToType 1 = Provide (FIntT IT8) -- "8 bit foreign integer"
bytesToType 2 = Provide (FIntT IT16)
bytesToType 4 = Provide (FIntT IT32)
bytesToType 8 = Provide (FIntT IT64)
bytesToType _ = Error "Unrecognised integral type."

getDevT : IO (Provider FTy)
getDevT = map bytesToType sizeOfDevT
{- lots of similar functions -}
```

Finally, we'll write one more idris file where we use the type providers:

```
-- Main.idr
module Main
import Providers
%language TypeProviders
%provide (FDevT : FTy) with getDevT

-- interpFTy translates a foreign type to the corresponding idris type
DevT : Type
DevT = interpFTy FDevT -- on most systems, DevT = Bits64

-- We can now use DevT in our program and FDevT in our FFun expressions!
```

The Interactive Theorem Prover

This short guide contributed by a community member illustrates how to prove associativity of addition on `Nat` using the interactive theorem prover.

First we define a module `Foo.idr`

```
module Foo

plusAssoc : plus n (plus m o) = plus (plus n m) o
plusAssoc = ?rhs
```

We wish to perform induction on `n`. First we load the file into the Idris `REPL` as follows:

```
$ idris Foo.idr
```

We will be given the following prompt, in future releases the version string will differ:

```
     ____    __     _
    /  _/___/ /____(_)____
    / // __  / ___/ / ___/     Version 0.9.18.1
  _/ // /_/ / /  / (__  )      http://www.idris-lang.org/
 /___/\__,_/_/  /_/____/       Type :? for help

Idris is free software with ABSOLUTELY NO WARRANTY.
For details type :warranty.
Type checking ./Foo.idr
Metavariables: Foo.rhs
*Foo>
```

2.1 Explore the Context

We start the interactive session by asking Idris to prove the hole `rhs` using the command `:p rhs`. Idris by default will show us the initial context. This looks as follows:

```
*Foo> :p rhs
----------                 Goal:                  ----------
{ hole 0 }:
 (n : Nat) ->
 (m : Nat) ->
 (o : Nat) ->
 plus n (plus m o) = plus (plus n m) o
```

2.2 Application of Intros

We first apply the `intros` tactic:

```
-Foo.rhs> intros
----------              Other goals:              ----------
{ hole 2 }
{ hole 1 }
{ hole 0 }
----------              Assumptions:              ----------
 n : Nat
 m : Nat
 o : Nat
----------                 Goal:                  ----------
{ hole 3 }:
 plus n (plus m o) = plus (plus n m) o
```

2.3 Induction on `n`

Then apply `induction` on to `n`:

```
-Foo.rhs> induction n
----------              Other goals:              ----------
elim_S0
{ hole 2 }
{ hole 1 }
{ hole 0 }
----------              Assumptions:              ----------
 n : Nat
 m : Nat
 o : Nat
----------                 Goal:                  ----------
elim_Z0:
 plus Z (plus m o) = plus (plus Z m) o
```

2.4 Compute

```
-Foo.rhs> compute
----------              Other goals:              ----------
elim_S0
{ hole 2 }
{ hole 1 }
{ hole 0 }
----------              Assumptions:              ----------
 n : Nat
 m : Nat
 o : Nat
----------                 Goal:                  ----------
elim_Z0:
 plus m o = plus m o
```

2.5 Trivial

```
-Foo.rhs> trivial
----------              Other goals:              ----------
{ hole 2 }
{ hole 1 }
{ hole 0 }
----------              Assumptions:              ----------
```

```
 n : Nat
 m : Nat
 o : Nat
----------                 Goal:                  ----------
elim_S0:
 (n__0 : Nat) ->
 (plus n__0 (plus m o) = plus (plus n__0 m) o) ->
 plus (S n__0) (plus m o) = plus (plus (S n__0) m) o
```

2.6 Intros

```
-Foo.rhs> intros
----------              Other goals:              ----------
{ hole 4 }
elim_S0
{ hole 2 }
{ hole 1 }
{ hole 0 }
----------              Assumptions:              ----------
 n : Nat
 m : Nat
 o : Nat
 n__0 : Nat
 ihn__0 : plus n__0 (plus m o) = plus (plus n__0 m) o
----------                 Goal:                  ----------
{ hole 5 }:
 plus (S n__0) (plus m o) = plus (plus (S n__0) m) o
```

2.7 Compute

```
-Foo.rhs> compute
----------              Other goals:              ----------
{ hole 4 }
elim_S0
{ hole 2 }
{ hole 1 }
{ hole 0 }
----------              Assumptions:              ----------
 n : Nat
 m : Nat
 o : Nat
 n__0 : Nat
 ihn__0 : plus n__0 (plus m o) = plus (plus n__0 m) o
----------                 Goal:                  ----------
{ hole 5 }:
 S (plus n__0 (plus m o)) = S (plus (plus n__0 m) o)
```

2.8 Rewrite

```
-Foo.rhs> rewrite ihn__0
----------              Other goals:              ----------
{ hole 5 }
{ hole 4 }
elim_S0
{ hole 2 }
{ hole 1 }
{ hole 0 }
----------              Assumptions:              ----------
 n : Nat
 m : Nat
 o : Nat
 n__0 : Nat
 ihn__0 : plus n__0 (plus m o) = plus (plus n__0 m) o
----------                 Goal:                  ----------
{ hole 6 }:
 S (plus n__0 (plus m o)) = S (plus n__0 (plus m o))
```

2.9 Trivial

```
-Foo.rhs> trivial
rhs: No more goals.
-Foo.rhs> qed
Proof completed!
Foo.rhs = proof
  intros
  induction n
  compute
  trivial
  intros
  compute
  rewrite ihn__0
  trivial
```

Two goals were created: one for `Z` and one for `S`. Here we have proven associativity, and assembled a tactic based proof script. This proof script can be added to `Foo.idr`.