The practical guide to writing a module with core external functions

Joop Ringelberg 20-04-20 Version: 2

Introduction

External functions are introduced in the text *External Function Interface*. A quick recap: queries consist of compositions of built-in functions and external functions. External functions come in two flavours: core and foreign, where core functions are written in Purescript and are compiled and bundled with the PDR.

This text describes the intricacies of creating a new core external module and adding functions to one.

Create a Core External Module

A core external module is just a Purescript module. However, in order for it to be usable in the Perspectives Language, it should be described by a PL model. This model can be empty, as illustrated by the model for the external module `model:Couchdb`:

```
-- Copyright Joop Ringelberg and Cor Baars 2019
domain: Couchdb
```

However, a model description instance is mandatory:

```
UserData
import model:System as sys
sys:Model usr:CouchdbModel
  extern $Name = "Couchdb External Functions"
  extern $Description = "A collection of purescript functions made
available as external core functions."
  extern $Url = http://127.0.0.1:5984/repository/model%3ACouchdb
```

The PDR parser/compiler system uses this model and its description to recognize names scoped to the model. Each core external module must be accounted for in the module `Perspectives.External.CoreModules`.

Declare function names in the Core External Module

Name your external function as you wish. However, you should tie it (in your module) to its qualified name as follows:

```
externalFunctions :: Array (Tuple String HiddenFunctionDescription)
externalFunctions =
  [ Tuple "model:Couchdb$Models" {func: unsafeCoerce models, nArgs: 0} ]
```

Points to notice:

1. There should be an identifier `externalFunctions` in your module, with the given type;
2. Each function should be declared by a Tuple having the *expanded* qualified name as its first member;
3. The second member of the Tuple contains a `HiddenFunctionDescription` and the number of parameters the function needs, minus one.

A `HiddenFunctionDescription` is just a type that tells the Purescript compiler to look away. The QueryCompiler will force the compiler to accept our functions as types as we have meant them.

The default argument

A query is a composition of functions. On executing a query, the entire composition is applied to either a Context instance (in the case of a Calculated Role) or a Role instance (in the case of a Calculated Property). Furthermore, each function in the composition receives as input value the output of the previously applied function.

All built-in query functions have just a single parameter. The modeller cannot apply a query explicitly, in the Perspectives Language; the query expression is just used as part of a Role or Property definition:

```
thing SomeRole = AnotherRole >> binding
```

So we do not need syntax for applying a built-in functions to arguments. This is different for external functions, as we are free to define them with any number of parameters!

That is why we have the callExternal syntax:

```
callExternal cdb:Models () returns: Model$External
```

Notice the parentheses. *If your function needs more arguments than the default, list them here, comma-separated!*

To come back to declaring the number of arguments in your module: you should declare the number of *extra* arguments. By default, a single argument will be offered. Your function is expected to bind that to its *last* parameter.

The type of that argument depends on the syntactical location you deploy your function

- If it is used as a query step, the type is the result of the previous step;
- If it happens to be the first function in a Calculated Role definition, it will be a Context instance;
- The first step of a Calculated Property definition gets a Role instance;
- If it is the (first step of the) right hand side of an assignment, it is the resource the assignment is applied to.

The other arguments

It is not possible to type the parameters of your function, because every parameter will be bound to an array of strings by the QueryCompiler. Obviously, you are free to coerce these strings to whatever value you like, as long as you do so responsibly. Remember that Context- and Role instances are referred to by their string identifier and that all Values are shuffled around as their string representation.

CallEffect

Instead of calling an external function for its functional result, you are free to create and deploy a function for side effects. As an example:

```
callEffect cdb:AddModelToLocalStore( object >> binding >> Url )
```

Here we call a function that downloads a model and adds it to the users' local model database.

Like all external functions, functions called for their effect take the default argument. You will have to provide a parameter for it to be bound to; and like with the other functions, you should ignore it when declaring the number of arguments your effectful function takes.

`callEffect` constitutes in itself a complete assignment expression. That means that it should stand by itself on a line wherever assignment expressions are allowed:

- following a `do for` clause;
- following an `action` clause;
- In the body of a `letA` expression.

When callEffect is in an automatic Action, it will be executed on a state transition. If the state is defined on a role, the default parameter is a role instance. If the state is defined on a context, it will be a context instance.

When callEffect is in a user Action, it will be in a perspective on a role. The instance of that role is the value of the default parameter.

The type of the external functions

As shown above, we have two ways of calling functions that are defined in an External Core Module:

- callExternal
- callEffect

The former is a query expression, while the latter is a statement evaluated for its side effect. Consequently, the type of the purescript functions that are actually evaluated - i.e. the functions you write in your external puresecript module - depends this distinction.

A query expression, called with `callExternal`, should be a purescript function whose return value is in `MonadPerspectivesQuery`.

A statement, called with `callEffect`, should be a purescript function whose return value is in `MonadPerspectivesTransaction`.