External Function Interface

Joop Ringelberg 09-01-20 Version: 2

Introduction

A Perspective model consists in part of calculated roles and properties. It can also include rules for bots: the combination of a condition and a series of statements that have a side effect. All these calculations consist of compositions of built-in functions such as `binding` or `context`.

In this text we describe a mechanism to make use of functions that are not built-in but added through modules. A *built-in* function keyword is recognised by the parser. In contrast, the name of an *external* function is not recognised. We need to use an explicit calling mechanism to signal the system it is a function.

Furthermore, the types of the arguments of a built-in function are compared to the requirements of its parameters. This is not true for functions that come from modules. The only check the parser/compiler system performs is on the *number* of arguments.

We will call these non-built-in functions *external*.

Outline of the design

Core and foreign modules

External functions can *belong to the core*, meaning they are compiled with the source code into the core program. These function are written in Purescript and thus are type checked by the Purescript compiler. It may seem confusing to call a function that belongs to the core program 'external'. It may help to remember that such functions do not belong to the Perspectives Design Language; they are external in that sense.

An external function can also be *foreign*. A foreign function is *not* distributed with the core program but comes in a separate Javascript module. Such modules can be added to a particular PDR installation by the end user (through appropriate mechanisms).

A foreign module introduces a safety risk, as its code is executed in the same interpreting environment as the core code itself. Someone with enough determination and skills will be able to write code that can access all secrets and affect the working of the PDR. This means that we will have to provide a mechanism of trust to enable end users to take an informed decision with regard to such a module. However, that topic is not the subject of this text.

Needless to say that core modules do not introduce such a risk: the end user can put the same trust in them as in the PDR itself.

Functions that yield a result

A *Computed Role* depends on an external function, by definition. Actually, a Computed Role is just the call of such an external function. To underline that fact, the syntax for constructing a Computed Role uses the keyword `callExternal`:

```
    context: Modellen = callExternal cdb:Models() returns: Model$External
```

Here, `cdb:Models` is an ordinary prefixed Perspectives identifier that expands to `model:Couchdb$Models`.

The `returns` keyword is followed by a Perspectives Identifier that identifies the type of the return value of the external function. Obviously, the function must return role identifiers[1] (in the case of a Computed Role).

Similarly, a *Computed Property* is an external function that yields a sequence of Values.

Once defined, a Computed Role or Property can be used in other calculations, just like any other Role or Property.

`db:Models` is a good example of an external function. It fetches the list of documents stored in the Models database of the Couchdb installation of the PDR.

It is also an example of a parameterless function. Functions with parameters need their arguments between parentheses. For a Computed Role, the argument expressions are interpreted relative to the context instance; for a Computed Property, the argument expressions are interpreted relative to the role instance.

External functions may be part of a composition. They may occur in exactly the same syntactical places as built-in functions.

Code that exerts an effect

In contrast to Computed Roles and Computed Properties, an *Effectful Statement* is the name we give to code that is executed purely for its side effect. The functional result of such code (if any) is ignored by Perspectives. Effectful statements can only be part of a model as the right hand side of bot rules.

A *core* Effectful statement can change the state of Perspectives: it may add or delete roles, contexts, properties, etc. A *foreign* Effectful statement cannot do so. It is executed outside the Perspectives engine[2]. An example of a foreign Effectful statement might be code that sends an email. For example:

```
    callEffect mail:Send( Receiver >> EmailAddress, Message >> Text )
```

[1] Please do not be confused by the `$External` part of the result type given here; it has nothing to do with the fact that `Models` is an external function.
[2] ‘Cannot’ is strictly not true. Such code is executed in the same javascript environment as the core code itself, as we’ve seen. But foreign Effectful statements are not *meant* to change Perspectives state.

This (hypothetical) statement reads the property `EmailAddress` of the role `Receiver`, and the property `Text` of a role `Message` and actually sends that text to the receiver. In general, an Effectful statement is called with any number of argument values between parentheses, separated by commas. Arguments are ordinary Perspectives query expressions. Like all other expressions in a rule[3], they are interpreted relative to the current context (the context of the bot).

Technical details

Core modules with external functions

We have several modules that are part of the PDR, that expose external functions to the modeller (*core* external functions). These modules have namespaces that are recognised by default. One of them is `model:Couchdb`, with the prefix `cdb`. It contains functions to access various aspects of storage of types and instances in Couchdb.

A qualified Perspective identifier does not comply with lexical rules for Purescript identifiers. For that reason we map the Perspective identifier we use in the model text for an external core function, to a function name as used in the core external modules. For example, `model:Couchdb$Models` is mapped to `couchdb_Models`. The mapping must be provided in the module that defines the external functions.

The QueryCompiler builds a function from a QueryFunctionDescription. This includes descriptions of external functions. So how does the QueryCompiler actually access the core modules that hold the functions themselves?

Even though these external functions are defined in Purescript modules that are compiled with the core, we need to retrieve them from a store and apply them to arguments in the QueryCompiler. Because of their variable number of parameters, we store them as HiddenFunctions in a specific store for ExternalCoreFunctions (defined in the module `Perspectives.External.CoreFunctionsCache`).

The compiler must generate a function call with a fixed number of arguments. Hence we record the number of arguments for external functions. Because we check the number of arguments in the DescriptionCompiler, the QueryCompiler can use an unsafe function to retrieve arguments by index from an array of computed values.

Foreign modules with external functions

If the QueryFunctionDescriptionCompiler encounters an expression with callExternal[4] followed by a function name scoped to a *foreign* module, it constructs a description that is

[3] Apart from Property Assignment expressions: these are interpreted relative to the current object set.
[4] And the same holds for statements with `callEffect`.

different from that constructed in case of a *core* module. The QueryCompiler uses that description as follows:

1. It separates the module name from the function name
2. It then calls a function `callForeignModuleFunction` in Aff, that has three parameters:
    a. The first is bound to the module name
    b. The second is bound to the function name
    c. The third is bound to an array holding all arguments. Each argument is itself an Array of Strings.

The result of that function is ignored.

`callForeignModuleFunction` is a foreign import (a function imported by Purescript). It's implementation is in Javascript. It essentially requires[5] a module by the given name, obtains a function from it and applies it to the argument list.

It also throws an error in each of these situations:

1. The module is not found;
2. The function is not found;
3. The integer value of property nArgs of the function object is not equal to the length of the argument list.
4. An error is raised during execution of the function.

Retrieving a foreign module

We use exactly the same mechanism for foreign external modules as for the modules that hold screen definitions for models. Consequently, a foreign module is stored as an attachment to a model file (DomeinFile). These files are stored in Couchdb and are retrieved from Couchdb just like screen modules.

[5] 'require' is Javascript lingo for loading a module.