

State and dependency tracking

Joop Ringelberg

22-11-19

Versie: 1

Introduction

This document is part of the description of the internal design of the Perspectives Distributed Runtime.

The runtime keeps state in an `AVar` in a `Reader` monad:

```
type MonadPerspectives = ReaderT (AVar PerspectivesState) Aff
```

Part of that state is for dependency tracking. Dependency tracking means that we recompute certain values when their input values, taken from representational state, change. This is the *functional reactive pattern*. With representational state we mean the information that is stored, in terms of contexts and roles, that represents part of the environment for end users.

We realise this by recording the relation between parts of that state, and functions that need to be recomputed.

It turns out, however, that we cannot store functions of type `MonadPerspectives a in` that state. Nevertheless we need to store such functions. They are kept in two caches that are stored in global, mutable variables.

The actual dependency administration can be kept in `PerspectivesState`.

Functions cached in global variables

We have three use cases:

1. A client of the PDR requests the value of a query. Whenever the underlying representation of contexts and roles changes, the PDR needs to update the query result and send it to the client (*functional reactive pattern*).
2. A bot has a rule with a condition that is true in certain states of a context instance. When the rule is triggered because of a state change, the right hand side of the rule must be executed. This right hand side consists of *assignments* and (context- and role-) *creation statements*. These change the representational state.
3. External functions. An external function can be used by the modeller by using the `callExternal` keyword. Such functions are defined in Purescript modules that are compiled with the PDR, but are not part of the Perspectives Language (and are, in that sense, 'external' to the language).

In the first case, the PDR constructs functions of the following type:

```
type ApiEffectRunner = Unit -> MP Unit
```

In the second case, the PDR constructs functions of this type:

```
type Updater s = s -> MonadPerspectivesTransaction Unit
```

Here, `s` is either a `ContextInstance` or a `RoleInstance`.

The third case stores structures of this form:

```
type ExternalFunction = {func :: HiddenFunction, nArgs :: Int}
```

Here, a `HiddenFunction` is a type not penetrable to the Purescript Compiler.

Updaters are stored in a cache defined in the module

`Perspectives.Assignment.ActionCache`. They are indexed by context instance identifier and action type (a so-called `ActionInstance`)

`ApiEffectRunners` are stored in a cache defined in the module

`Perspectives.DependencyTracking.Dependency`. This cache is indexed by correlation identifiers that are communicated over the external API with clients of the PDR.

`ExternalFunctions` are stored in a cache defined in the module

`Perspectives.External.CoreFunctionsCache`.

Dependency tracking administration

Queries

The administration for query-dependency tracking is defined in the module

`Perspectives.CoreTypes`:

```
type AssumptionRegister = F.Object (F.Object (Array CorrelationIdentifier))
```

The Foreign Objects are indexed by strings that represent the two elements of an `Assumption`. An `Assumption` is a combination of

- a resource (`ContextInstance` or `RoleInstance`)
- and a type (`EnumeratedRoleType`, `CalculatedRoleType`, `EnumeratedPropertyType` or `CalculatedPropertyType`), or the special identifier `"model:System$Role$binding"`

An assumption points to the value of a role in a context, a role binding, or of a property in a role¹. When changes to such values are made, the code reviews the assumption register to find correlation identifiers that identify functions that compute queries dependent on those values.

An instance of `AssumptionRegister` is stored in `PerspectivesState` under the key `queryAssumptionRegister`.

¹ See *Implementing the Functional Reactive Pattern* for a better explanation of assumptions (called 'steps' in that text).

Object versus GLStrMap

The `AssumptionRegister` is created as a `Foreign.Object`, while the other two caches are created as a `GLStrMap`. The underlying representation is the same. However, the former is modified purely functional, while the latter is modified destructively. We believe that there is no functional difference between the two in the way it is used in `Perspectives`. The destructive operations are faster. The implementation difference reflects the current state of the program and is likely to change in the future.