Adding Aspect Perspectives

Joop Ringelberg 21-12-21, 28-9-22 Version: 2

Introduction

The functional languages Haskell and Purescript have a feature that is called *type classes*. They may be compared with *interfaces* as known in other languages. In Haskell/Purescript, a type class can have *instances*, where an instance is a type that must supply an implementation for the types (usually functions) that make up the type class. In some cases, the compiler is able to derive instances automatically.

Type classes allow us to decorate a type *multiple times* with 'behaviour' in terms of functions. It is in that respect superior to (singular) OO inheritance.

Perspectives allows contexts to have another context as an aspect, effectively giving the former to the latter as an extra, or super type. It also allows us to add the roles of other contexts, effectively allowing *context composition*. The same holds for roles: a role can have an aspect role, adding the aspect roles properties to it and 'inheriting' the aspect role's restriction on its possible fillers.

Aspects are, in a way, to Perspectives what type classes are to Haskell and Purescript.

User roles have Perspectives. They, too, can be composed of Aspect (User) roles. What if the latter have Perspectives, too? In this text we explore the space of meanings we can give to such compositions. It turns out that Queries, Actions and Automatic Actions must be 'contextualised' when taken from an Aspect and incorporated into a decorated context.

Contextualising can be compared to supplying the implementation of a type class's types in an instance. It turns out that the Perspectives compiler can derive all these 'instances' automatically – that is, it can carry out contextualisation automatically without requiring any action on the part of the modeller.

Compile time or run time?

Queries, actions and perspectives are described as particular data structures that are part of a (compiled) model. We may contextualise them in compile time to new versions that we can store in the model. Alternatively, it turns out, we can adapt the runtime engine to carry out the contextualisation of an abstract structure in a concrete situation. As this text will show, we pursue a mixed strategy on further expanding Perspectives, contextualising some structures in compile time, others in run time.

Contextualising a Perspective

We work with an example, taken from `model:BodiesWithAccounts`:

```
case Body

  user Test
    property UserName (String)

  user Admin filledBy sys:PerspectivesSystem$User
    aspect bwa:WithCredentials

    perspective on Accounts
      only (Create, Fill, CreateAndFill, Remove)
      props (UserName, Voornaam, Achternaam) verbs (SetPropertyValue, Consult)

  user Accounts (unlinked, relational) filledBy sys:PerspectivesSystem$User
```

The user roles in this model fragment are used as Aspects in another model, `model:CouchdbManagement`:

```
case CouchdbServer
  aspect acc:Body

  user Admin filledBy CouchdbManagementApp$Manager
    aspect acc:Body$Admin

    perspective on CouchdbServer$Accounts
      props (ToBeRemoved) verbs (Consult, SetPropertyValue)

  user Accounts (unlinked, relational) filledBy sys:PerspectivesSystem$User
    aspect acc:Body$Accounts
```

Notice the relations between the latter model and the former:

- `CouchdbServer` has `Body` as Aspect;
- `CouchdbServer$Admin` has `Body$Admin as` Aspect;
- `CouchdbServer$Accounts` has `Body$Accounts as` Aspect;
- Both Admin roles have a perspective on the Accounts role in their context.

What we want

Clearly, we want that `CouchdbServer$Admin` can apply `Body$Admin's` facets of the perspective on `Body$Accounts` to `CouchdbServer$Accounts`. Specifically, we expect `CouchdbServer$Admin` to be able to create an instance of `CouchdbServer$Accounts` in virtue of the perspective of its aspect `Body$Accounts`.

But we also expect that the verbs applicable to the property `ToBeRemoved` are available to `CouchdbServer$Accounts`.

In other words, we want to 'add' both perspectives in such a way that the resulting perspective:

- grants access to the union of the properties of both;
- allows, per property, the union of the propertyverbs allowed by each perspective individually;
- allows the union of the roleverbs allowed to each perspective individually.

To add two perspectives is conditional on the relation between their objects: we may only add an Aspect perspective to another perspective if the latter's object is a specialization of the former's object.

Compile time contextualisation

For roles that are added *as is* to a context, we'd have to contextualise

- their calculation (if any): this would be query contextualisation and the paragraphs below will show we will apply run time contextualisation to queries. Consequently, no specific action is required for the compile time contextualisation of the calculation of roles.
- Their perspective object. As these are queries as well, this falls in the same case as calculations. No action required.
- Their Actions (actions with an object). As the relevant paragraph will show, Actions will be contextualised in run time.

In short: nothing needs to be done to contextualise a role that is added *as is* to a context.

For roles that are added as an aspect to a role, things are different. First of all, we have to distinguish two cases:

1. An aspect adds a perspective on an object that the specialised user role does not have a perspective on. In this case, the perspective is added *as is* to the specialised role (is simply copied).
2. The specialised user role already has a perspective on a local object role, that turns out to be a specialisation of the object of the aspect user role.

In the second case, we have to adapt the perspective of the specialised user role, but only with respect to its

- role verbs,
- property verbs.

Role and property verbs are additive over aspects: a verb applicable according to an aspect or the specialised role is applicable to the specialised role.

Design decision I: we rely on compile time perspective contextualization w.r.t. verbs for composed roles; on run time contextualization of calculations, perspectives objects and actions.

Where perspectives are used, does contextualization matter?

Synchronization. The query that is the object of the perspective, is inverted in order to find user roles that should be informed when a change affects the outcome of that object query. Obviously, the query must be contextualized if it is taken from an aspect to a special context. However, because queries are contextualized run time, no further action is required for synchronization.

Authorization. A Delta is only accepted if the author of the change it describes, does have a perspective that authorizes him to do so. Contextualisation of aspect perspectives is important for this process, as the required authority may derive from an aspect. Compile time contextualization as described above will do the trick neatly.

Screens generation. Obviously, the perspectives contributed by aspect user roles to a user for whom we request a screen giving access to a context, cannot be missed. The mixed strategy contextualization of perspectives described above will take care of this.

State complicates matters

In the example above, perspectives were valid in the root state of their objects or subjects. When a perspective is only valid in some state, things get complicated very quickly when we want to combine perspectives.

The relation between two (macro) states can be thought of as a Venn-diagram of two circles representing the micro-states of both (macro) states. The diagram allows for three separate regions, where the perspectives may only be added in the union (the overlap of the two circles).

This would require we compute, in type time, the logical conjunction of both states' condition. This we will not do. It requires logical reasoning beyond the complexity we are willing to tackle.

Of course, we can just combine both conditions and see what happens in runtime. However, we then might create conditions that will never evaluate to true but consume resources nevertheless.

Also, picture once again the Venn-diagram. It may be that one state is entirely inside the other. We would have no way of knowing it and would still have a separate perspective for the contained state. Again, a waste of resources.

What we will allow

Instead of reasoning about state conditions, we will allow

1. a perspective to be conditional on an Aspect state (this is a modelling facility: one can refer to Aspect state in an `in state` clause);
2. a non-ground state Aspect perspective to be added to a ground state specialised role's perspective;
3. an ground state Aspect perspective to be added to a non-ground state specialised role's perspective;
4. two perspectives to be added when conditional on the same state.

With regard to the formerly discussed Venn-diagram, this reduces to these cases:

1. the states of the specialised role perspective and the Aspect perspective are equal;
2. the specialised role perspective is valid in the (a) ground state, while the Aspect perspective is valid in a named state. In this case, the intersection coincides with the Aspect state;
3. The intersection coincides with the specialised role state perspective.

Aspect states that may be used

We only allow an Aspect state to be used in contexts or roles that have the type that has that state, as aspect.

Contextualisation of queries

A query consists of a series of steps. Some of these steps must be contextualised when a query moves into a context as an aspect (e.g. in the form of a Calculated role, or as the condition of a state):

- the role step: moving from a context instance to a role instance of a particular type R;
- the filled role step: moving from a role instance to another instance of a particular type R, that is filled by it.

Referring back to our example above: assume a query defined for case `Body`, that moves to instances of role `Accounts`. Clearly, when we start with an instance of `CouchdbServer` (having `Body` as aspect), we will not find role instances of `Body$Accounts` on it; instead, we should contextualise the query step from `Account` to `CouchdbServer$Accounts`.

It turns out that runtime contextualization is rather easy for the role step. For a given context type, we can compile an *alias administration* that maps, for each Enumerated role type available in the context, its supertypes onto that Enumerated role. Then, when a role step has to be carried out on a context instance by the query evaluator, we have it

1. look up the type of the context instance;
2. fetch the alias administration from it;
3. look up the role step type to find the appropriate Enumerated role type
4. use that to look up instances on the context instance.

This causes some overhead during query evaluation, but saves a great deal of space with inverted queries. Remember that a query is *inverted* for reasons of synchronization and state transition and that the inversions are stored with each role and context type that is part of the query. When we contextualize a query in compile time, we have to invert the contextualized version, too.

If we rely on runtime contextualization, when a role instance is added to a context instance, we have to look up inverted queries on the type of the role instance *and on all its aspect types*. Climbing the aspect hierarchy takes a little time, but in the end the same number of queries has to be evaluated.

The **filled role step** can be contextualized in runtime using a variant of the alias administration. We then need to build this administration in the *filler role* instances. An example will make this clearer. Suppose we have a pattern with a Driver and a Vehicle role and we use it to

specialise a context with a Pilot and a Plane, respectively. The first time we fill a `Pilot` role instance with some role instance R, we'd have to record on R that `Pilot` is an alias for `Driver`. Then, when the filled role step `Driver` is carried out on R, we look up `Driver` in the alias administration of R, find `Pilot`, then read from R what role instances are recorded as filled under the key `Pilot`.

Overall, we find that run time contextualisation of queries is conceptually simpler, imposes less storage overhead and introduces an acceptable runtime penalty.

Design decision II: we rely on run time query contextualization.

Contextualisation of actions

Actions consist of statements. Some statement types mention a role- or context type to create:

```
create role RoleType [in <contextExpression>]
bind <binding> to RoleType [in <contextExpression>]
unbind <binding> [from RoleType]
create context ContextType bound to RoleType in <contextExpression>
create_ context ContextType bound to <roleExpression>
```

In these statements, `RoleType` refers to some role that should be defined for the current context, or the type of context that results from `<contextExpression>`.

We have two kinds of Action: those defined in lexical positions with a current object (where the action will be applied to that object), and without a current object (these actions will be applied to the current context). We call the former kind *perspective actions* and latter kind *context actions*.

Both kinds must be contextualised, when the user role with access to the actions is used as an aspect for a specialised user role, or is included *as is* in a context. The point is that the `RoleType` (and `ContextType`) may be specialised in the contextualising context, too, in which case this specialisation should be substituted for the original in the assignment statements.

Returning to the Driver-Pilot example: let's assume for the sake of the explanation that the Driver can create the Vehicle role. Clearly, we want the Pilot to create a Plane, rather than a Vehicle. So we substitute Plane for Vehicle in the actions of the Pilot.

The consequences of compile time contextualisation

Skipping the details of *how* to contextualise an action[1], we ask ourselves: where can we store the contextualised actions?

A contextualised *context action* cannot be tied to its user role's representation, since we can incorporate an aspect user role *as is* in as many contexts as we like. Nevertheless, we'd have to contextualise the action for each such context. Obviously, a contextualised action is specific to the *combination* of a user role and a context. We can save Actions directly in the DomeinFile in a map with keys constructed from context- and user role types.

What about contextualised *perspective actions*? Currently (version v0.18.0) we store perspective actions in the perspectives. We could contextualise the perspective holding contextualised actions, but then store perspectives in the DomeinFile, again under a key constructed from context- and user role type. In other words, if we incorporate a(n aspect) user role into a context, we create a specialised version of the perspectives of that user role.

This requires a major change to the implementation, however. This is because the above implies we conceive of a perspective as a relation between *two role-in-a-context combinations*, rather than between two roles. On the perspective object side, we've already tackled that issue (because of query inversion). The abstract data types that we use to describe query domains and ranges are in terms of combinations of role and context. However, on the perspective subject side it means a complete overhaul of many modules.

Compile time contextualisation of actions requires a major refactoring.

How to contextualise an Action

Before deciding on compile time versus runtime contextualisation of actions, we explore how to contextualise individual statements.

[1] More about this later.

Create role

The `create` operator in conjunction with the `role` keyword:

```
create role RoleType [in <contextExpression>]
```

mentions a `RoleType` that must be contextualised. Let us work with a different example to create some intuition: we have an Aspect context `Meeting` with a role `Organizer` and a role `Participant`. Now, we create a `MedicalAppointment` with:

- a role `Physician` and a role `Patient`, both having aspect `Participant`
- a role `Assistant`, having aspect `Organizer`.

Organizer can create `Participant` role instances and has an action to create one. What happens when `Assistant` executes that action? As there are two specialisations of `Participant`, two instances will be created: a `Physician` and a `Patient`.

Contextualisation of this action in **compile time** would result in *two* statements in the action:

```
create role Physician [in <contextExpression>]
create role Patient [in <contextExpression>]
```

However, we have a variant of this assignment operator that lets us save the result in a `letA` variable. This introduces a dilemma: should we create an ad hoc new variable? Or can we assume that the variable can hold multiple values, and will semantics of the rest of the action be conserved, under this change?

Contextualisation of this action in **run time** would require an adaptation of the code that is compiled from the `CreateRole` datastructure. We would have to look up the specialisations of `Participant` in `MedicalAppointment`, keep only those that the user executing the action has a sufficient perspective on and then iterate over those specialisations.

Both approaches need the same lookup (of local specialisations of the `RoleType`). While compile time contextualisation is not quite clear, run time contextualisation is conceptually simple.

Bind

The `bind` operator:

```
bind <binding> to RoleType [in <contextExpression>]
```

needs to be handled just like the `create role` operator. It, too, creates a role instance.

Unbind

The `unbind` operator:

```
unbind <filler> [from RoleType]
```

works by clearing those roles filled by `filler` that have type `RoleType`. However, no roles will be cleared that have `RoleType` as a supertype (an aspect of their type). In terms of our example: if the instruction is to unbind from `Drivers`, no `Pilot` roles will lose their filler. In compile time, we should replace `RoleType` with the specialised role type to contextualise the action. Again, if multiple role types are specialised, we'd have to duplicate the statement for each of them.

However, we can handle it in runtime, too, relying on the *alias administration* in the filler role instances described above for the filled role step. Each time we fill a `Pilot` role instance with some role instance R, we'd have to record on R that `Pilot` is an alias for `Driver`. Then, when `unbind` is carried out, we look up `Driver` in the alias administration of R, find `Pilot`, then read from R what role instances are recorded as filled under the key `Pilot`. Finally, we'd clear the filler from those instances.

Create context

The `create` operator can also be used in conjunction with the `context` keyword:

```
create context ContextType bound to RoleType in <contextExpression>
```

The `RoleType` must be contextualised just like in the `create role` situation. The `ContextType` must be replaced, too: by the type of context that may be bound to the substitution of `RoleType`.

To extend our `Meeting` and `MedicalAppointment` example, let's assume there is a `Calendar` context with a role `Meetings` that holds contexts of type `Meeting`. And let's assume that `Calendar` is added as an aspect to `HospitalCalendar`, with a role `HospitalMeetings` that has aspect `Meetings`. It, however, restricts its fillers to `MedicalAppointment`. Now, some user in `HospitalCalendar` with an aspect role that is allowed to create `Meetings`, should create an `HospitalMeetings` role instance rather than a `Meetings` instance. And it should fill it with a `MedicalAppointment`, instead of a `Meeting`. To accomplish this, the runtime first finds that `HospitalMeetings` is the local substitution for `Meetings` and then discovers `MedicalAppointment` as its filler restriction, being a specialisation of `Meeting`.

Create_ context

The `create_` operator is a variant of the `create` operator, in that it omits the `RoleType` to create and instead retrieves an existing role instance that should be filled with a new context:

```
create_ context ContextType bound to <roleExpression>
```

Looking up the `ContextType` goes exactly in the same way as with `create`.

Wrapping up: contextualising Actions

Compile time contextualisation of actions requires a major refactoring. In contrast, run time contextualisation requires just part of the mechanisms that are required for compile time contextualisation (the lookup of type substitutions for statements with the operators `create`, `create_`, `bind` and `unbind`.

Design decision III: we rely on run time action contextualization.

Contextualization of automatic actions and notifications

Automatic actions are predicated on state change. A modeler may refer to a state change defined in an imported model. Such actions, however, are stored with the state definition itself. This poses a problem: a modeler may not change an 'upstream' model (he need not have authoring rights of that model).

This is a problem similar to that of calculated perspective objects, where the object role type is defined in another model. We have devised a mechanism that consists of a store of inverted queries in a model, that are distributed over imported models *in the installation*. In other words, we extend upstream models – but only locally, each time as a user installs a model.

We will apply a similar mechanism to automatic actions that predicate on an aspect state (a state of an aspect context or role that are defined in an upstream model).

Automatic actions itself need no contextualization, as we've shown above.

The same reasoning applies to notifications, that predicate on state changes, too.

Design decision IV: automatic actions and notifications are distributed over upstream models in runtime.

Contextualization of Properties

When we add a role type `A` to a role type `R`, all `A`'s properties become available on `R`. That is, we can add values for a property `A$P` to the representation of an instance of `R`. It is as if `A$P` was defined as one of `R`'s own properties.

In some circumstances, `R` might already have a property that can be seen as a local version of `A$P`. This is much like the situation where we have a role in a context that we want to add an aspect role to. So, in analogy, we would like to be able to specify that a property `R$P` should be considered as the specialization of `A$P`; that is, we want to add `A$P` as an 'aspect property' to `A$R`.

Notice that the defaults are different for roles than for properties. No aspect role is added implicitly to a specializing context, whereas we do want all aspect properties to be added implicitly to a specializing role. As a consequence, we need a different syntax to indicate what we want (explicitly adding all aspect properties would make our models very verbose).

Instead, we will write that *an aspect property is **replaced** by a local property*:

```
user Driver filledBy sys:PerspectivesSystem$User
  property License (String)
user Pilot
  aspect ta:Transport$Driver where
    License is replaced by Certification
  property Certification (String)
```

We expect the following effects from such a replacement:

1. That a specialized perspective does not hold the aspect property, but its specialization (e.g. a perspective on `Driver` with property `License`, when specialized to `Pilot` should show `Certification`).
2. That a calculation in the aspect that refers to `License` should, in effect, be computed as if `License` was replaced in the source text by `Certification`.
3. That an assignment in the aspect on replaced property should be carried out on the replacement (e.g. when an assignment line sets `License` for `Driver`, it should set `Certification` for `Pilot`).

Obviously, the expected behavior 2 follows from runtime query contextualization. Behavior 1, in contrast, should be carried out on perspectives that are added from the aspect user role to a specializing user role – in compile time. Behavior 3, finally, will be carried out in runtime much as we do for action contextualization of role assignments.

Summary

This text contains the following design decisions:

1. We rely on compile time perspective contextualization w.r.t. verbs for composed roles; on run time contextualization of calculations, perspectives objects and actions.
2. We rely on run time query contextualization.
3. We rely on run time action contextualization.
4. Automatic actions and notifications are distributed over upstream models in runtime.