Contextualizing

Joop Ringelberg 14-05-20 Version: 1

Introduction

Perspectives relies on a powerful abstraction mechanism called *aspects*. An aspect is a Context[1] that may be added to another Context, which we call a *specialiser* of the aspect. It thereby brings its roles into that specialiser. Furthermore, a role in the specialiser may be augmented by a role of the aspect. The role thus becomes a specialiser, too. The effect of Role specialisation is that

1. The aspect Role properties are added to the specialiser's role;
2. The binding of the specialiser *must be equal to or more specific than* that of the aspect role;
3. If the aspect Role is a User role, its perspective (a collection of Actions) is added to that of the specialiser role.

Example

`model:SimpleChat` contains the definition of `Chat`, whose definition follows (partly) below:

```
  case: Chat
    aspect: sys:Invitation
    user: Partner (not mandatory, functional) filledBy:
sys:PerspectivesSystem$User
      aspect: sys:Invitation$Invitee
```

Chat has aspect `sys:Invitation`:

```
  case: Invitation
    user: Invitee (mandatory, functional) filledBy: Guest
    bot: for Invitee
      perspective on: Invitee
        if exists Invitee then
          bind object to ConnectedPartner in PrivateChannel >> binding >>
context
```

The role `Partner` in `Chat` is a specialisation of the role `Invitee` of `Invitation`. Notice that `Invitee` has a bot and its condition is that an `Invitee` must exist.

[1] We talk *types* unless specified otherwise.

Problem statement

When we add an instance of `Partner` to an instance of `Chat`[2], we expect the bot to do its thing. After all, `Partner` is just an `Invitee` in disguise. But in Perspectives v0.4.0 that will not happen. The rule condition, being the query `exists Invitee`, applies the step `Invitee` to the instance of `Chat`, looking for a *literal* occurrence of "Invitee"[3]. And it will not find it: it has an instance of `Partner` instead. As a consequence, the rule will not fire.

We could, of course, rewrite the rule:

```
if exists Partner then
  bind object to ConnectedPartner in PrivateChannel >> binding >> context
```

This would work[4]. We would have *contextualised* the rule in the type `Chat`. Obviously, we would have to add the rule (and thus the bot) to `Chat` (the model checker would refuse it as part of `Invitation`, because `Partner` is not defined in `model:System`). So in effect we would have to overwrite this rule in the specialising context.

That is not what we want. We want a mechanism that contextualises automatically.

Solution

There are several ways to solve this problem. In principle, we would like to solve it entirely in compile time (let the modeller wait so the end user has better performance). This would involve automatically rewriting queries from aspects to fit their specialisers. It would also entail inverting those specialised queries. And these inverted specialised queries would, by definition, cross more model boundaries than the originals. It is not impossible, but cumbersome.

Instead, we have chosen to do a little more work in runtime. The key observation is that *we need aliases when looking up roles*. When looking up `Invitee`, we should know that `Partner` is an alias - and lookup with that key, too.

We can find aliases by reflecting on the model. After all, the definition of `Partner` references `Invitee`. How and when do we use this information?

Double indexes in context instances

In v0.4.0, a `PerspectContext` instance contains an `Object (Array RoleInstance)` where the keys are the (string values) of Role(types). Retrieving the instances of a Role requires just one lookup.

2 In the rest of the text we will write 'when we add a Partner to a Chat', leaving out all references to instances, meaning the same thing.
3 Property qualified, so really: "model:System$Invitation$Invitee".
4 One might wonder: will the effect not fail on the step `PrivateChannel`? After all, that is a Role of `Invitation`, too, just like `Invitee`. But we need not change it, because there is no specialization of `PrivateChannel` in `Chat`. It is added *as is*.

We will change that by

- Adding a new member to `PerspectContext` (the representation of context instances): `aliases`. This will be an `Object String`. The keys are, as before, Role(types); the values are indices in an array `roleInstances`. The type of `roleInstances` is `Array (Array RoleInstance)`.
- Looking up a particular Role in two steps: first we find, in `aliases`, an index that we then use to look up the actual instances in `roleInstances`.

We build this structure gradually per `PerspectContext` instance, when we add a Role instance:

- If there is not yet an entry for the type of the instance to add, we reflect on the model by looking up the type. We then add an entry to `aliases` for the type *and for all its aspects*. We also add a new empty array to `roleInstances`. Obviously, we store the index of this new array with all keys we've just added to `aliases`.
- If there is an entry, we take the index associated with it and add the role instance in the corresponding array in `roleInstances`.

Performance wise, there is an small extra penalty for each role instance lookup and some more work on changing instances, mainly when adding the first instance of a particular type.

Reverse lookup: the binder step

There are two ways for a query to visit a role instance. The first, which we've discussed above, is when we move from a context instance to a role instance. The second is when we move from a role instance to a particular (set of) binders: roles that bind the instance we depart from.

The `binder` step requires a Role(type) because any Role may be bound by many others[5]. Obviously, this step is affected by contextualisation, too. As an example, consider this small query:

```
User >> binder Invitee >> context
```

As `Invitee` is bound to `model:PerspectivesSystem$User`, we can navigate back from an instance of `model:PerspectivesSystem` to an `Invitation`. But now consider a `Chat` with a `Partner`. `Partner` will be bound to `User` as well. However, in version v0.4.0, this query will not find any instance of `Chat`.

Again, we have to rely on aliases to make it work. But this time, as the direction is reversed, the alias table is structured differently.

[5] Be careful to distinguish between multiple instances of the same Role type, and multiple types! Here we mean the latter.

We add a new member to `PerspectRol`: `inverseAliases`. This is an `Object (Array RoleType)`. Its keys are the string values of Role(types). Its values are the Roles that specialise the key, including itself[6].

The representation of the inverse bindings does not change. But lookup does. In our example, when binder first looks up `Invitee` in `inverseAliases`. It finds `[Invitee, Partner]` and then looks up both in the inverse bindings, combining the results. Thus it finds the `Partner` instance that binds the user instance.

Again, we build this structure gradually. If there is not yet an entry for a Role(type), we reflect on the model, find all aspects of the type and add the type under the entry for each aspect (adding an entry when necessary).

No consequences for serialization

The association between a Role(type) in the object `aliases`, and a particular index in `roleInstances`, depends on the historic order in which role instances were added to that particular context instance. These histories could be different for users sharing that context: one may have a perspective on a Role that another has not.

This learns us that we cannot communicate these indices in Deltas. But this need not worry us, because a Delta is like a *remote procedure call* rather than a data item. The receiver of a Delta executes the call and that will lead to appropriate and possibly unique association between role types an indices.

Similarly, a context Serialization is translated into calls to functions that reconstruct contexts and roles.

[6] Notice that for a Role that has no aspects, an entry would be made whose value would be an array that contains just the Role type itself.