Introduction to the Perspectives language

Joop Ringelberg 25-10-19 Version: 2

Introduction

This text gives an introduction to the Perspectives language. Appendix I contains the complete specifications of its syntax. Appendix II is an informal semantics for PL.

A domain with some roles

A model is collection of types. It always starts with the declaration of a Domain:

```
domain Taxi
```

A model name always has the prefix `model`, followed by a name starting with an uppercase character. All names the modeller creates for his types must start with a capital[1].

Role: user and thing

Roles play a significant part in Perspectives models. A role is minimally declared as follows:

```
domain Taxi
  user Driver
```

The keyword `user` signifies a role that is played by a person. We have keywords for other kinds of roles:

```
domain Taxi
  user Driver
  thing Taxi
```

A Taxi is a thing that plays a role – but it is not a person.

Property

An important property of a taxi is how many passengers it can carry:

```
domain Taxi
  user Driver
  user Passenger
  thing Taxi
```

[1] See Appendix One for a detailed description of allowed names.

```
    property NrOfSeats
```

Attributes of properties; range

We should have declared NrOfSeats in this way:

```
  property NrOfSeats (mandatory, relational, Number)
```

With the keywords `mandatory`, `relational` and `Number` in parentheses behind the name of the property we further characterise that property. The first two we call *attributes* of the property. A mandatory property must have at least one value for each role instance. A relational property can have more than one value. By default, properties and roles are *functional*.

With `Number` we specify a *range* for the property. In this case we have a numerical property. Besides `Number`, other options are `Boolean` (true or false), `String` and `Date`. If we leave out the attributes and range, the following default values are assumed: `not mandatory`, `functional`, `String`.

Attributes of roles

Roles have the same two attributes:

```
domain Taxi
  user Driver
  user Passenger
  thing Taxi (mandatory)
    property NrOfSeats (mandatory, Number)
```

A mandatory role must have an instance; a functional role can have only one instance. Instead, one might stipulate a role to be `relational`: this means that it can have many instances. For example:

```
domain Taxi
  user Driver (mandatory)
  user Passenger (not mandatory, relational)
  thing Taxi (mandatory)
```

Passengers are not mandatory (however much the taxi driver might like that!) and there can be several passengers inside a taxi.

If we leave out the attributes of a role, by default it is constructed as not mandatory and functional.

Perspective

User roles have a *Perspective* on other roles.

```
domain Taxi
```

```
  user Driver
    perspective on Passenger
  user Passenger
  thing Taxi
```

By giving the `Driver` a perspective on the `Passenger`, we arrange that the application we model will provide a (graphical user interface) screen for the `Driver`. What will the `Driver` be able to see? If we don't specify otherwise, all properties that `Passenger` has.

Notice that our application, as it stands now, can only be used by the `Driver`. Even though `Passenger` is a user Role, it has no Perspective on anything, so no screens are provided to persons playing the `Passenger` Role..

Let's be more precise about what the `Driver` will see.

View

We can detail a Perspective with a *View*

```
domain Taxi
  user Driver
    perspective on Passenger (PassengerDetails)
  user Passenger
  thing Taxi
```

Here, `PassengerDetails` is a View. A view is merely a list of properties of the role it is defined on:

```
domain Taxi
  user Driver
  user Passenger
    view PassengerDetails (FirstName, FamilyName)
  thing Taxi
    property NrOfSeats (mandatory, Number)
```

Views are foremost important for their use in a perspective.

Roles can be filled

You may have noticed that we referred to two properties (`FirstName`, `FamilyName`). But where do they come from? They are not declared with `Passenger`.

If we reflect for a moment, we see that these properties are not really properties of the role `Passenger`, but of persons in general. It is not as if the driver says, when his passenger enters his taxi: I will call you Betty. The passenger had a name beforehand!

So how do we model this situation? Below we specify that `Passenger` roles are played by (filled by) a `Person`:

```
  user Passenger filledBy Person
```

Assuming that `Person` indeed has the properties `FirstName` and `FamilyName`, we would now be allowed to refer to them from a View on `Passenger`.

So where does this person come from? We take it from some other model, let us call it the `PersonalDomain`. In that model, `Person` is defined with the two properties:

```
Domain PersonalDomain
  user Person (mandatory)
    property FirstName (mandatory, String)
    property FamilyName (mandatory, String)
```

By filling the `Passenger` Role with `Person`, we make `Person`'s properties available in `Passenger`. Hence we can define a View with those properties, as we've done with `PassengerDetails`.

Don't confuse View with Perspective!

A View is a list of properties of a Role. A Perspective gives a *user role* the ability to see the properties of another role, as specified by some view (or all properties if no view is given in the Perspective). But a Perspective is not only about *seeing*: it is also about creating and changing, to name a few. We'll come to that shortly.

Qualified names

We have one thing to fix, however, with our `Passenger` definition. Up till now we have used simple, meaningful names for our roles. However, such names might well crop up in other models, too: 'Passenger' would be a good name for an aviation domain, for example. To prevent confusion, names are *qualified with their domain*. So really, `Passenger` is named `model:Taxi$Passenger`. Notice the $ sign between the model name and the role name: it separates the various *segments* of the *qualified name*.

Now this happens implicitly, most of the time. So we are allowed to write just `Passenger` if no risk of confusion arises. However, on using a name from another domain, we **have** to qualify it:

```
  user Passenger filledBy model:PersonalDomain$Person
```

This is quite a mouthful. Luckily, we have the means to abbreviate qualified names with *prefixes*. So our full model could be written like this:

```
domain Taxi
  use per for model:PersonalDomain
  user Driver
    perspective on Passenger (PassengerDetails)
  user Passenger filledBy per:Person
    view PassengerDetails (FirstName, FamilyName)
```

```
thing Taxi
  property NrOfSeats (mandatory, Number)
```

Verbs

A useful computer program usually allows its users (or at least *some* of its users) to change things. And to create things and delete them. In Perspectives, we link these capabilities to the *Verbs of the perspectives*. In our example, some user would have to enter the number of seats of the taxi. Let's assume it is the Driver. He must have a perspective that we could describe as: *the driver changes the number of seats of the taxi*.

Now here is a very important principle: **the Perspective of a user Role on another Role is given by the Verbs the user apply to that other Role.** So, really, a Perspective is a list of Verbs! Consulting is just one of these verbs. You may object that you've not seen any verbs in the perspective we've seen:

```
user Driver
  perspective on Passenger (PassengerDetails)
```

We've specified *what we would like to see* of the Passenger (by giving the View `PassengerDetails`), but not *what we would like to do* with it. However, this is a full Perspective, because, by leaving out any further information, by default we include all verbs.

Verbs come in two flavours: role verbs and property verbs. Role verbs allow the user having the perspective a.o. to create a role instance or delete it and, importantly, to fill it with another role.

Property verbs allow a user to give a property a value.

So our perspective above is, with respect to role verbs, equal to the one below:

```
user Driver
  perspective on Passenger (PassengerDetails)
    all roleverbs
```

If we want to deny some of these verbs to a user, we can to list them:

```
user Driver
  perspective on Passenger (PassengerDetails)
    excluding (Delete, Create, Remove)
```

this would be a better model, by the way, because the `Driver` should not be able to remove the `Passenger`![2]

We will come back to Verbs later, because there is a lot more detail we can specify about them.

[2] Client is King, after all.

Context

A role is tied to some context. A nurse is a nurse in the hospital; not while she is shopping. Let's introduce a context for the roles we've seen so far: the taxi ride.

```
domain Taxi
  use per for model:PersonalDomain
  case TaxiRide
    user Driver
    …
```

We simply indent all the lines we had so far to the right, under the new heading for the `TaxiRide`. Nothing much changes, except for the qualified names: `Passenger` now is `model:Taxi$TaxiRide$Passenger`. This stands to reason as the `Passenger` is now inside `TaxiRide`.

A single context in a model is not very useful. Let's introduce another:

```
domain Taxi
  use per for model:PersonalDomain
  case TaxiCompany
    user Personnel filledBy per:Person
      property EmployeeNumber (mandatory, Number)
    case TaxiRide
        user Driver filledBy Personnel
    …
```

Notice how we've pushed `TaxiRide` one stop further to the right, making it a subcontext of `TaxiCompany`. We've also added who can be a `Driver`: the personnel of the `TaxiCompany`. And, because we've stipulated that Personnel need be a `per:Person` and we know that the latter has a `FamilyName` property, we can now also add a view on the Driver, e.g. in order to be able to show his name to the `Passenger` (obviously we would need to give `Passenger` a Perspective on `Driver` for this to work).

Furthermore notice that we've added an `EmployeeNumber` to `Personnel`. With all those roles that fill each other, we have fine-grained control over where to register properties. An employee number is not a personal property; neither is it particular to some taxi ride. It is relevant in the context of the taxi company. However, if we wish to, we can make it available (through some view) in the taxi ride context.

Properties of a context

From the perspective of the scheduling operator, it is important to be able to have an overview of `TaxiRides` that have not yet finished. This we can model with a property of *the external role of the context*. Think of a context as having an *inside*, and an *outside*.

It's outside represents the context in other situations. To carry information, it can have properties on its outside (i.e. on its external role):

```
case TaxiRide
     external
       property Finished (mandatory, Boolean)
     user Driver filledBy Personnel
```

So now we have defined `TaxiRide` with an external property, for some operator of the TaxiCompany to see. How do we go about providing this operator with a Perspective on `TaxiRide`s?

To prevent misunderstanding…

… we have to carefully distinguish between **context types** *defined locally* to some other **context type**, and **context instances** *actually appearing* in another **context instance**. Let's explain with an example. Below is our full model, augmented with another line:

```
domain Taxi
  use per for model:PersonalDomain
  case TaxiCompany
    user Personnel filledBy per:Person
      property EmployeeNumber (mandatory, Number)
    context Rides filledBy TaxiRide
    case TaxiRide
      external
        property Finished (mandatory, Boolean)
      user Driver filledBy Personnel
        perspective on Passenger (FullName)
      user Passenger filledBy per:Person
        view PassengerDetails (FirstName, FamilyName)
      thing Taxi
        property NrOfSeats (mandatory, Number)
```

We've added a role `Rides` to `TaxiCompany`. `Rides` is a role, just like `Personnel` is a role. The `context` keyword introduces a role, just like `user` and `thing` introduce roles. Only, for `context` this role is *filled by another context*.

So why is this? We already had the case `TaxiRide` inside `TaxiCompany`. Why do we need the role `Rides`?

We have come to this point without discussing the difference between types and instances. Now we need the distinction. A model gives a lot of types. But when we 'run' a model, we deal with **instances** of these types (running a model means: using the software that we've specified with the model). When an actual person wants to order a taxi ride,

he will create a new instance of `TaxiRide` (by using the software, pressing a button for example, and by entering some details).

This means that *an instance* of a `TaxiCompany` (let's call them Unter) will accumulate *instances* of `TaxiRide`s. An operator working for Unter will inspect those rides and monitor those that are still in progress.

However, on the type level, the *type* `TaxiRide` is contained within the *type* `TaxiCompany`.

The takeaway is:

1. `case` gives the *declaration of a type of context, inside another context type*
2. `context` gives the *declaration of a role in a context, filled with a context*.

Properties of a context, revisited

We've given `TaxiRide` an external property `Finished`. One way to think of this is that a particular `TaxiRide` is *represented* by a role that has this property. We call that role the *External role* of `TaxiRide` (of contexts in general). It is that External role that actually fills the `Rides` role of `TaxiCompany`. Let's define a View on it:

```
domain Taxi
  use per for model:PersonalDomain
  case TaxiCompany
    user Personnel filledBy per:Person
      property EmployeeNumber (mandatory, Number)
    user Operator filledBy Personnel
      perspective on Rides (ViewOnRides)
    context Rides filledBy TaxiRide
      view ViewOnRides (Finished)
```

We have provided a perspective for a new type of user, `Operator`, on `Rides`. The operator cannot see inside the `TaxiRide` that fills an instance of `Rides`, but he can see its external property `Finished`.

External Roles support us when creating programs, by providing a way of *hiding information*. The operator doesn't need know who is inside that taxi or where it started. He wants to know whether it is finished (so he can schedule another ride for the driver).

Let's calculate some things!

Continuing with the `Operator`, who is interested in the status of the `TaxiRides`, we realise that seeing a long list of rides and a checkmark behind each of them saying whether it is finished or not, does not make for a good user interface. At the very least, we would like to be able present a list of just the rides that have not yet been finished. So how do we create such a *filtered list* in Perspectives?

Calculated Roles

Let's think of a name of such a list: `UnfinishedRides`. This is how we create a new type of Role in `TaxiCompany` that is just what we need:

```
case TaxiCompany
   thing UnfinishedRides = filter Rides with not Finished
```

We call such a Role a *Calculated Role*. In contrast, `Rides` is an *Enumerated Role*. We have at our disposition a number of functions and operators to calculate roles, `filter` being a very important example. Filter takes a source – `Rides`, in our example – and a criterium. The criterium is an *Expression* that has a Boolean value.

Some things are rather implicit in this filter expression:

1. what `Rides` do we mean? Sit back and reflect for a moment: there may be many taxi companies and each would have instances of `Rides`. Surely we do not mean all those instances! Instead, we want an operator to view just the unfinished Rides *in his own company*.
   So here is the rule: in a Calculated Role, the calculation *starts at its context instance*. We can think of the source as an expression that provides a *path* starting at a particular context instance, leading to some role instances. In this case, the path consists of just a single step.
2. Similarly, we want to judge each role on it's `Finished` property. Implicitly, the criterium expression *starts at each Role that is judged*. Again, it is a path leading from that Role to a Boolean value.

Let's scale up the example. Here is how we create a list of all `Drivers` in `TaxiRide`s that are still under way:

```
   user OccupiedDrivers = UnfinishedRides >> Driver
```

We have re-used our Calculated Role and created a new Calculated Role with it. The `>>` operator separates steps of the path we want to travel. Its name is *compose*; you might want to pronounce the entire Expression as: *take UnfinishedRides **and then** the Drivers of those rides*.

Actually, this Expression is not valid. As explained above, `UnfinishedRides` is a Context Role. Each of its instances have bound a `TaxiRide` instance – or actually, the External Role of a `TaxiRide` instance. So we have omitted a number of steps. This is the accurate path:

```
   user OccupiedDrivers = UnfinishedRides >> binding >> context >> Driver
```

Explanation:

1. the CalculatedRole starts at the `TaxiCompany` instance.
2. With `UnfinishedRides` we select some instances of the `Rides` Role;
3. *and then,* with `binding` we move to the External Role instances that they bind;

4. *and then,* with `context` we move from those External Roles to the `TaxiRide` instances;
5. *and then,* with `Driver` we move to our destination, the instances of `Driver` in all those `TaxiRide` instances.

`binding` and `context` are operators that move from a role to a role and from a role to a context, respectively. We have more operators. See the Perspectives Language Reference for a full list.

Calculated Properties

Just as we have Calculated Roles, we can have Calculated Properties, too. As Properties are about simple values, we can showcase a number of other functions that we can deploy in our Expressions.

But first we will extend `TaxiRide` with two Roles: `Origin` and `Destination`. Both should have properties that somehow say something about a location. We will not go into details about `Location`, but just assume it exists. But we also need some date- and time information. For both `Destination` and `Departure` we want to keep record of the planned moment and the actual moment. This is what we define:

```
case TaxiRide
  thing Origin (mandatory) filledBy Location
    property Planned (mandatory, DateTime)
    property Actual (mandatory, DateTime)
  thing Destination (mandatory) filledBy Location
    property Planned (mandatory, DateTime)
    property Actual (mandatory, DateTime)
```

We now are in a position to do some calculations. Let's define PlannedDuration:

```
  PlannedDuration = Destination >> Planned - Origin >> Planned
```

What, as a matter of fact, is the type of this definition? Inspecting the Expression, we see we subtract the values of two Properties. Hence, it must be a property definition. So what is it a Property of? Of what Role? As it is a property of the entire `TaxiRide`, a natural place would be its External Role. However, the starting point of a Property Calculation is the role on which it is defined. We can't move from the External role to another role of the context in one step, so we have to change the definition slightly:

```
case TaxiRide
  external
    property PlannedDuration = context >> (Destination >> Planned - Origin
>> Planned)
```

The first step in the Calculation moves from the External Role to its context (`TaxiRide`). Then we can retrieve the `Planned` property of both `Destination` and `Origin` and subtract them from each other.

Besides subtraction, we also have addition (+), division (/) and product (*). For Boolean values, we have ‘and’, ‘or’ and ‘not’. Strings we can concatenate with +. And DateTimes can be added and subtracted from each other. Furthermore, we can compare values with ‘==’, ‘<=’, ‘>=’, ‘<’ and ‘>’.

Functions on sequences

Let’s return for the moment to the perspective of the `Operator` on `Rides`, both roles of `TaxiCompany`. The operator might be interested in the average planned duration of all `Rides`. This is how we could try to model that:

```
case TaxiCompany
  external
    property AverageRideDuration = context >> (Rides >> PlannedDuration / Rides)
```

We mean to express that we want to sum all values for `PlannedDuration` and divide that by the number of `Rides`. Obviously, something is lacking here! What we need is a way to express that we want to apply a ‘sum’ function to a whole sequence of numbers, not just to two of them. Also, we want to count the number of `Rides`. This is how we do it:

```
case TaxiCompany
  external
    property AverageRideDuration = context >> (Rides >> PlannedDuration >>= sum / Rides >>= count)
```

The >>= operator applies the function on its right side to the sequence of values obtained by the expression on its left side. The function must reduce the sequence to a single value.

We have a number of such sequence functions: ‘sum’, ‘count’, ‘product’, ‘minimum’, and ‘maximum’. Notice how all work on numbers. However, ‘count’ will operate on any type of value.

Perspectives in more detail

We’ve seen how Verbs can be included implicitly, by just creating a Perspective on a Role. We’ve also seen how we can limit a Perspective by summing up the Verbs we allow. There are a few more details we can set for Perspectives.

Role verbs

We've seen how we can exclude some role verbs. We can also exclude them all except for a few:

```
user Driver
  perspective on Passenger
    including (Fill)
```

Now Driver can use no role verbs, except for `Fill`. The entire list of role verbs is: `Remove, Delete, Create, CreateAndFill, Fill, Unbind, RemoveFiller, Move.`

Property verbs

We can actually specify verbs for each property separately:

```
user Driver
  perspective on Passenger
    props (FirstName) verbs (DeleteProperty, SetPropertyValue)
```

Driver can apply to the (comma-separated) list of properties of Passenger the verbs listed after `verbs`. Alternatively, we can attach the property verbs to a view:

```
user Driver
  perspective on Passenger
    view (PassengerDetails) verbs (DeleteProperty, SetPropertyValue)
```

Quite often a user is allowed to see a lot more than that he is allowed to change. For the sake of demonstrating the principle, let us assume that the `Driver` can see, but not change, the `Planned` property of the `Origin` Role. Furthermore assume he has to set the value of the `Actual` property. This would amount to the following Perspective:

```
perspective on Origin
  view AllProperties (Consult)
  props (Actual) SetPropertyValue
```

(we assume the View AllProperties on Origin that we do not further define).

The property verbs are: `RemovePropertyValue, DeleteProperty, AddPropertyValue, SetPropertyValue, Consult.`

State

Sometimes, an Verb should only be available when certain conditions have been met. For example, the `Driver` can only charge the `Passenger` after arriving at the `Destination`. Let's take a value for the `Actual` property of `Destination` as a proxy for arriving at it.

Using that, we define a *state* for the external role of `TaxiRide`:

```
case TaxiRide
  thing Destination (mandatory) filledBy Location
    property Planned (mandatory, DateTime)
    property Actual (mandatory, DateTime)
    property Fare (Number)
    state Arrived = exists Actual
      perspective of Driver
        props (Fare) verbs (SetPropertyValue)
```

We've given a perspective to `Driver` that is only valid in state `Arrived`. As a consequence, only when the `Actual` property is set, can `Driver` fill in the definite `Fare`.

Automatic actions

A Perspectives model describes some part of the world in terms of several types, as we've outlined above. A user running a model will create instances of those types, change and delete them, thereby moving the state of the application forwards. Sometimes, such changes must *always* happen in certain circumstances. Those situations can be automated: users may delegate some of the work to the system.

It is important to realise automatic actions are always performed on behalf of a specific user.

Building on our previous example with state Arrived, we can define an automatic action to be performed when the Destination role gets into that state:

```
case TaxiRide
  thing Destination (mandatory) filledBy Location
    property Planned (mandatory, DateTime)
    property Actual (mandatory, DateTime)
    property Delayal (mandatory, DateTime)
    state Arrived = exists Actual
      on entry
        Finished = true for extern
```

As soon as the taxi arrives, the system sets the `Finished` property of the external role of the `TaxiRide`.

Notifications

As users change properties and roles and contexts, their peers can notice these changes. But will they? A property may change that is not even on screen, for some peer. This is where *notifications* come in. Notifications are assistive technology for end users: they help to draw his attention to some changes. Above we've modelled the Operator of the TaxiCompany with a perspective on Rides:

```
domain Taxi
  use per for model:PersonalDomain
  case TaxiCompany
    user Personnel filledBy per:Person
      property EmployeeNumber (mandatory, Number)
    user Operator filledBy Personnel
      perspective on Rides (ViewOnRides)
    context Rides filledBy TaxiRide
      view ViewOnRides (Finished)
```

Let's extend `Rides` with a state that is based on the property `Finished` of its filler, `TaxiRide`:

```
    context Rides filledBy TaxiRide
      state Completed = Finished
        on entry
          notify Operator "The ride starting at {binding >> context >>
Departure >> Location >> Address} has finished"
```

In the previous paragraph we've seen how the `Finished` property of the `TaxiRide` itself is set automatically. Now we build on that, making the system notify the `Operator` of that state. It does so with a message that is partly built from static text, partly from a computation, to wit the address of the point of departure of the `TaxiRide`.

System

`sys` is the standard prefix for `model:System`. `System` models the Perspectives Distributed Runtime itself. Each runtime has exactly one instance of the Context `sys:System`. In this context instance, we find one instance of `sys:System$User`. This instance represents the `user` of a particular runtime – 'the' user operating the computer running it. All user Roles are ultimately filled by instances of `sys:System$User`[3].

Assignment

An assignment is a statement to the extent that *something changes*. We have assignment statements for properties and assignment statements for roles. Automatic actions are expressed as assignments.

Properties

Let's begin with assignments for properties. They have the form:

```
  PropertyType <operator> <expression> [for <roleExpression>]
```

[3] But notice that, in our example, the Driver and the Passenger roles will be filled by *different* users – on different computers.

or:

```
delete property PropertyType
```

We have three operators: one to add a value to an existing set, one to take away a value and one to replace all values with a new set. For example:

```
MyProperty =+ 10
```

will add the value 10 to the existing values of `MyProperty` (if any). By now you may wonder: of what role do we change property values? By default, this is the role that is specified as the Object of the Action. However, in an assignment statement one can change that by supplying the optional `for <roleExpression>` part.

Create a Role

It is straightforward to create a new instance of a Role in an assignment statement:

```
createRole Rides
```

will create a new instance of `Rides` in `TaxiCompany` (the role name is plural, because it represents a collection of instances. A newly created instance is just a single Role that is added to the existing collection). The `createRole Rides` syntax actually is an assignment statement.

Create a Context

With `createContext`, we create a context of the given type and bind it to a new instance of the given Enumerated Role type in the current context:

```
createContext ContextType bound to RoleType
```

In order to bind it in another context, we add a clause:

```
createContext ContextType bound to RoleType in <contextExpression>
```

It goes without saying that actually the external role of the fresh context is bound to the new role instance.

bind

The `bind` function is an assignment operator. Its general form of use is:

```
bind <expression> to EnumeratedRoleType [in <contextExpression>]
```

It takes the value of its Expression argument and makes it the filler of a new instance of its EnumeratedRole type argument (in other words: it binds the expression value in a new role instance).

In our example, as the value of `createRole TaxiRide` is an External Role, we need to name a Context Role in order to legally bind it. `Rides` is such a Role.

bind_

Sometimes, we already have an instance of a Role that we want to bind a value to. In such cases, we use the `bind_` assignment operator:

```
bind_ <bindingExpression> in <binderExpression>
```

Here, the first expression must select a single binding while the second expression is used to fetch an unbound Role instance (the binder).

unbind

We have two ways to break the binding between a Role and its filler. The simple way is to select an instance:

```
unbind <expression>
```

Obviously, `<expression>` must have a role instance as value. We can select all instances by just supplying the Role name, or we can filter the instances of that Role however we like it. We consider these instances to be *bindings* and unbinding means removing them from all binders that bind them.

Usually, that is rather strong and we want to be more picky about what we want to unbind from. So we add the type of binder that we want to unbind from:

```
unbind <expression> from RoleType
```

As with bind, there is a variant unbind_ that allows us to select both a single binding and a single binder and break them apart.

Delete

Sometimes we just want to remove all instances of a role. Then we use `delete`:

```
delete <roleExpression>
```

Select the instances to be removed. To select from another context, just extend the query.

Aspects

We have seen how we can define Context and Role types. We construct a Context by constructing Roles inside it, for example. The language as we have exposed it so far enables us to create arbitrary complex models. However, there is yet another facet of the Perspectives Language that enhances our powers of abstraction and re-use, and that are Aspects.

Any Context or Role type can be thought of as an Aspect. We can add a Context type to another Context type as an Aspect. By doing so, we *add the roles of the aspect to the*

context type. So Aspects can be thought of as components that we can build more complex types from. This is how we add an Aspect to a Context

```
case Car
  aspect Vehicle
```

(let's assume that Vehicle contributes roles like `Driver` and `Passenger`).

The same holds for Roles. By adding a Role as an Aspect to another Role, we *add the properties of the Aspect Role to the role type*. Here is how to add an Aspect to a Role:

```
thing Home
  aspect Location
```

Here we assume that `Location` contributes properties like X and Y coordinates, for example.

Aspect is different from binding

We might be tempted to define `Home` like this:

```
thing Home filledBy Location
```

On the type level, we make `Location`'s properties available to Views on `Home`, too, just like with Aspect. But on the instance level it is completely different. There we have to provide *an instance* of `Location` to bind it to an instance of `Home`. With the Aspect-modelling, we have to provide values for `Location`'s properties to the instance of `Home`. There are no instances of Aspects!

Aspect exclusively is *a type-level concept*.

Appendices

I. Syntax of the Perspectives Language
II. Semantics of the Perspectives Language