Semantics of the Perspectives Language

Joop Ringelberg 29-10-19, 21-02-22 Version: 2

Introduction

This text gives an informal semantics for the Perspectives Language. We use the concepts of Sum- and Product types loosely in the way of Category Theory.

Context

A Context is a product type. A Context is the product of a number of roles, at the very least just a single one. In our programs we attribute special meaning to that single role: it *represents* the Context. In this account of semantics of PL it does not figure, however, otherwise than that there must be at least a single role for a Context. There is no *empty context* in PL.

Property

A Property is an atomic type, as far as the semantics of PL concerns. There are but three things of interest to a Property (we call them *attributes*) and those are

1. Whether it is *mandatory;*
2. Whether it is *functional;*
3. What is its *range type*.

A Property is a single dimension to compare Roles on. Range types are for example Integer, Boolean, String and Date. However, an implementation may add other Range types.

Of theoretical interest is a Range type without values. We use it to define the ValueLess Property.

Role

Roles are by far the most interesting type in PL. A Role is one of three things:

1. A *Simple Role*, which is a Product of Properties;
2. A *Product Role*, which is a Product of Roles;
3. A *Sum Role*, which is a Sum of Roles.

Because we can transform a Product of Sums or a Sum of Products to a normal form that has just Simple Roles, we can simplify the above definitions by stating that a Product Role is a Product of Simple Roles (and a similar thing holds for Sum Roles). So, without loss of

generality, we assume some normal form and just talk about Sums and Products of Simple Roles.

Role instances

Above we have defined Role types in terms of properties. Alternatively, we can define a Role type as a set of *Role Instances*. We say that the type is *inhabited* by its instances. An instance is a collection of values for Property Types.

A Role Type constrains the instances that can inhabit it. It requires that an inhabitant has a value for each of the Property Types that define the Role Type. Each such value must be one of the values of the Range type of the Property[1].

Properties of a Product Role

A (normalised) Product Role is just a product of Simple Roles. A Simple Role is a product of properties. Hence, the properties of a Product Role is just the Product of the properties of all Roles in the product.

An instance of a Product Role of A and B is an instance of both A and B and carries all properties of A and all properties of B. Consider an instance 'Joop' of the product of Person and of PhysicalObject. As a Person, Joop has a familyname (Ringelberg). As a PhysicalObject, Joop has a dimension length (1.93 meter).

Properties of a Sum Role

A (normalised) Sum Role is just a sum of Simple Roles. A Simple Role is a product of properties. An instance of a Sum Role of A and B is either an A, or a B. What properties can we be sure of? Those that are in the intersection of the properties of A and B.

Consider a limited definition of a Vehicle as the Sum of either a Car or a Bicycle. This sum will have a property NumberOfWheels, but it will not have a property CapacityOfFuelTank – even though occasionally an instance of this sum will have both properties.

The Empty Role

We can imagine a Role with no properties: it is called the Empty Role. A Product of a Role with the Empty Role is just that Role itself; a Sum of a Role and the Empty Role is itself the Empty Role.

[1] Here, it is of importance to distinguish the requirement that a role instance has *at least* the required properties, or that it has *exactly* the required properties. The latter we can interpret as a form of the *closed world assumption* (CWA): the descriptions are definite and something that is not described, is not the case.

Explanation of EmptyRole x Role = Role. When we form a Product of Roles, we take the *union* of their properties. However, the EmptyRole does not have properties, so contributes nothing. Hence the result is just the original Role.

Explanation of EmptyRole + Role = EmptyRole. A Sum of Roles A and B is the role with the *intersection* of their properties. But the Empty Role has no properties, hence its intersection with any other Role is just an empty set – in other words the Empty Role itself.

Does the Empty Role have instances? A Role type constrains its instances by requiring that they have a value for each Property in the type. But the Empty Role has no Property Types, so is not much of a constraint. Hence, all role instances are an instance of the Empty Role[2].

The Universal Role

Now turn to a Role with all properties: let's call it the Universal Role. A product of the Universal Role and another Role is just the Universal Role; the Sum of the Universal Role and another Role is just that other Role.

Explanation of UniversalRole x Role = UniversalRole. When we form a Product of Roles, we take the *union* of their properties. However, the UniversalRole already has all properties, so Role contributes nothing. Hence the result is just the UniversalRole again.

Explanation of UniversalRole + Role = Role. A Sum of Roles A and B is the role with the *intersection* of their properties. But the Universal Role has all properties, hence its intersection with any other Role is just that other Role.

Universal Role instances

The Universal Role is funny when it comes to instances. To start with, an instance of the Universal Role is always an instance of any other Role (it always qualifies because it has values for all properties!).

But does the Universal Role have instances at all? Yes, infinitely many, because there are infinitely many properties and each instance of the Universal Role has a value for each of them. It follows that each such instance must be infinitely large.

Nevertheless, each ordinary Role has instances that the Universal Role does not have (because the Universal Role is the pickiest of all Role types). So, while being infinite, its instance set still is less than that of any other Role (for a discussion, see *Instances of a Product*, below).

[2] An alternative interpretation, based on the CWA, holds that the role instance *can have no properties as they are not described*. Hence, we come to the opposite conclusion: just a role without any properties at all can be an instance of the Empty role. But what is a role without properties?

Importantly, the infinite character of Universal Role instances severely limits its usefulness, however, because in our programs we only handle finite instances. In other words, in practice the Universal Role cannot be inhabited. So if we require that a Role be filled with instances of the Universal Role type, we say in that – again, in practice, in our programs - it will never be filled.

Empty Property construction

We have postulated a Property with a Range without values: the ValueLess property. If we concede that a Role instance with this Property cannot exist (because an instance must have a value for each Property), it follows that the Universal Role is an uninhabited type. After all, the Universal Role has all Properties, including the ValueLess Property.

So if we assume the ValueLess Property, the Universal Role has no instances, even in theory.

Ordering of Role types as property sets

Considering a Role to be a set of properties means we can compare Roles and partially order them. For example, the Sum A+B has just the properties that are in the intersection of A's properties and B's properties. Hence the properties of A+B are a subset of both A's properties and B's properties. We say that A+B is *less specific* than either A or B.

Definition: Role X is *less specific* than Role Y if X's properties are a subset of Y's properties (less specific = isSubset over property-set);

It follows that AxB is *more specific* than either A or B. It has more properties than either A or B.

Empty Role is least specific

What about the Empty Role? It's properties are the empty set. The empty set is a subset of every other set, hence **the Empty Role is less specific than any other Role**.

Universal Role is most specific

The Universal Role has all properties. It is a superset of all other sets, hence **the Universal Role is more specific than any other Role.**

Instances and ordering

A Sum Role like A+B has more instances than either A or B. After all, each instance of A is an instance of A+B and the same holds for instances of B. So while A+B is *less* specific than either A or B, it has *more* instances. This is understandable if we consider a Role type as a constraint on instances. So we can give the definition of less specific in terms of instances too:

Definition: Role X is *less specific* than Role Y if X's instances are a superset of Y's instances (less specific = isSuperset over instance-set).

Instances of a Product

The instances of a Product may confuse you. The Product AxB actually has *less* instances than either A or B. But nevertheless it is created by multiplying A and B together! In our minds eye the multiplication table is larger than the row of column labels or the column of row labels.

But consider: each instance of AxB is an instance of A (it has the required properties). But an instance *with just the properties required by A* is not an instance of AxB – though it obviously is an instance of A. So A has instances that AxB has, but not vice versa – AxB is a subset of A.

Another ordering of Role types

There is, however, another base for comparing roles and that is to consider instances as 'tagged individuals'. An instance is an instance of a role R by declamation, as it were; not because it has a particular structure (in terms of properties).

Let's briefly diverse into a discussion of Aspects to underline this[3]. Consider a rather universal role Driver, that can be used in many contexts, e.g. as the driver of an Ambulance or the driver of a Taxi. Driver-in-an-Ambulance has a different meaning than Driver-in-a-Taxi. A model may, for example, contain a query to count the number of hospitals that the Driver-in-an-Ambulance has visited in a period of time. Clearly, if the same person drives both Taxi's and Ambulances, we want to separate out both 'contextualized' roles when performing that query.

Product types

There is no `natural` ordering of simple role types. But there is a way to consider compound roles to be ordered. Take, for example, the Product of two roles A and B and compare it to A itself. Surely, A is less specific than AxB. Any instance of AxB is, by construction, also an instance of A, but not the other way round.

From this we derive the notion that, for Product types, when we consider the products as the sets of their terms, subset means: less specific.

Sum types

What about a Sum of A and B? How does that compare to A? Well, an instance of A is always an instance of A | B, but not the other way round (after all an instance of B is an instance of A+B as well – but clearly not an instance of A!). So, A+B is less specific than A.

[3] See the chapter below for a brief explanation of Aspects.

Paraphrasing it may be easier to understand: A is more specific than A+B ("I will talk not to just any Parent, only to the Mother!").

So, for Sum types, considering them as sets of terms, subset means: more specific.

Arbitrary types

Taking this as the base of our alternative definition of ordering, how can we generalize it to any role defined in terms of Sums and Products?

Any construction in terms of Sums and Products can be converted into Disjunctive Normal Form (DNF). A formula is in DNF iff it is a disjunction of products, where the term of each product is atomic.

Let's work out how we can compare two types written in DNF.

We know that if terms(type1) is a subset of terms(type2), where type1 and type2 are Products of simple types, means: type1 is less specific than type2.

What if type2 is a disjunction of many products?

Any of these products will qualify as a type2. For type1 to be less specific than type2, it must therefore be less specific than any of the products in type2. *terms(type1) must be a subset of terms(type2$_x$)* (for any of the disjuncts of type2).

This means that terms(type1) must be a subset of the *intersection(terms(type2$_x$))*.

Now, for the last step, we consider the case that type1 itself is a disjunction of products. Now the above must hold for any of these products. We can translate that into the requirement that, *for type1 to be less specific than type2, the union of terms(type1$_x$) must be a subset of the intersection of terms(type2$_x$)*. This is implemented in the following function (module Perspectives.Representation.ADT, PDR version v0.18.0):

```
equalsOrGeneralisesADT :: forall a. Ord a => Eq a => ADT a -> ADT a -> Boolean
equalsOrGeneralisesADT adt1 adt2 = let
  union' = allLeavesInADT adt1
  intersection' = commonLeavesInADT adt2
  in subset (fromFoldable union') (fromFoldable intersection')
```

To compute the union of the terms over a DNF is simply to swap the disjunction operator for a conjunction operator. Obviously, doing so does not depend on DNF.

Aspects

In the previous paragraph we used reduction to Disjunctive Normal Form (DNF) to prove the soundness of the function equalsOrGeneralisesADT. This function does not take Aspects into account, because we treat terms in Products and Sums as atomic. However,

this is insufficient to capture the semantics of Perspectives types, as the following example shows. Let T have aspect A, then clearly A `equalsOrGeneralises` T. However, our function will return false! This is because union' will be {A}, while intersection' will be {T} and {A} is not a subset of {T}. Luckily, we can easily improve the function to cover aspects, too.

When we give a type T an aspect A, we actually state that any instance of T is an instance of A as well. We can think of a type-T-with-aspect-A as the Product TxA. Imagine that we expand all terms to the full set of their aspects (i.e. the type itself and its aspects). Then we would have:

- union' = {A}
- intersection' = {A, T}

and now the first is a subset of the latter, hence clearly A `equalsOrGeneralises` T returns true! In other words, we can capture the semantics of Perspectives types in the function equalsOrGeneralises by expanding all types to their transitive closure over the aspect relation.

Creating products and sums of roles

So how do products and sums of roles arise? PL allows for several ways to construct them.

Filling

The primary way to construct a Product Role is by specifying a Role with a filler:

```
user: Person (..) filledBy: PhysicalObject
```

Here we create the product of Person and PhysicalObject. All properties of PhysicalObject 'can be reached' through Person. We can specify a View with all those properties.

FilledBy as a constraint

The type we specify as the possible filler of a Role constrains what instances we may actually bind to that Role. Only inhabitants of the filler type may be bound to an instance of the Role.

When we say that the filler of a Role is the Empty Role, we say in effect that anything goes. There is no constraint. So, by default, each role has the Empty Role as its filler type.

In contrast, the Universal Role as the filler type of a Role is a tall order. We've seen that in the practice of our finite programs there can be no instances of the Universal Role. So in effect, by requiring the Universal Role as filler type of a Role, we say that that Role can have no filler.

Multiple fillers

This is how we create a Sum of Roles:

```
thing: Vehicle (..) filledBy: Bicycle, Car
```

Vehicle can be filled by either a Bicycle, or by a Car. Notice that we can add Properties to the definition of Vehicle itself, too. Then we would have a product of Vehicle and the sum of Bicycle and Car. All instances of Vehicle would have Vehicle's properties.

When Vehicle has no properties of its own, it is in effect the EmptyRole. A product of R and the EmptyRole is R itself, so in that case Vehicle is just the sum of Bicycle and Car.

Multiple fillers in a product

We also have the option to specify multiple fillers in a product[4]:

```
user: ClientOfPharmacy (..) filledBy: BankAccountHolder + Patient
```

Here, an instance of the role ClientOfPharmacy **must** be filled by both an instance of BankAccountHolder and an instance of Patient. This is because the pharmacy needs access to my medical records and it also needs access to my bank account number (assuming it will automatically collect the costs incurred).

Why do we need the `BankAccountHolder + Patient` syntax? Can we not instead accumulate properties by filling BankAccountHolder with Patient? We could, and we could do it the other way round, too. And that is the catch: by choosing either way, we *tie both roles together forever*. Obviously, I don't need my medical records in all situations where I need my bank account and the same holds vice versa. There is no 'natural model': it just depends on the situation.

Even worse, we can never be sure what combinations of properties might be required in future models. The best way we can prepare for that future is to lump everything together in the 'role at the bottom', whatever we choose that to be. It is an *include everything even the kitchen sink* approach to modelling we obviously want to avoid.

This is why we need the opportunity to bring in packets of properties as desired, using the product syntax given above.

Another way to think of this is that a product created by filling is 'on the supply side', whereas a product created by + is 'on the demand side'. We create that product when we need it, instead of creating it because we might need it in the future.

[4] Attention: we use the + symbol here to denote a product in the sense that we've defined it above. This is because +, interpreted as 'and', reads more natural to a lay person than *.

Aspects

By giving a Role an Aspect, we bring in the properties of that Aspect. So, on the type level, Aspects create a product of roles, too. For example:

```
thing: HomeAddress (..)
      aspect: Location
```

creates the product of HomeAddress and Location. Instances of a HomeAddress will have properties of both, for example TypeOfHome for HomeAddress and X and Y from Location.

The following lines create the same product type:

```
Thing: HomeAddress (..) filledBy Location
```

However, on the instance level things are different. With the latter model, we would have to bind an instance of Location to an instance of HomeAddress. With the former model, there would be no such instance. Instead, the HomeAddress instance would have the Location X and Y properties, too.

Aspects of Contexts work in the same way. A Context Aspect introduces Role types into a Context. However, there is no alternative modelling to consider, as we cannot ‘fill’ Contexts.