

Abstract Data Type

Joop Ringelberg

24-01-20

Version: 1

Introduction

There are three use cases that underline the need for *abstract data types* to reason about roles, contexts and properties. Without explaining the concept, we'll give these three cases to build some intuition.

The binding of a Role can be a set of alternatives¹. We call such a set a Sum of types. Alternatively, we can stipulate that a Role must be bound to multiple other instances *at the same time* and that we call a Product of types.

The query language supports concatenating two queries. This means we can create a set of role instances of two or more types. The type of such a set is a Sum type, too. In a sense we create a new type of role in this way, just like an Enumerated Role with multiple alternative bindings (but in a query we construct it on the fly and we do not add own properties, or aspect roles).

An EnumeratedRole is described by giving it aspects and a binding. If we reason about roles and their relations, it is convenient to lump these three facets together in a single Product type. The role itself (as a set of Property types), its aspects and its binding form a Product.

In the text *Semantics Of The Perspectives Language* we have explained these issues in more detail. In this text we describe the representation of such compound types and the functions we have for manipulating them.

Representation

We represent compound types with an Abstract Data Type, or ADT for short:

```
data ADT a =  
  ST a |  
  SUM (Array (ADT a)) |  
  PROD (Array (ADT a)) |  
  EMPTY |  
  UNIVERSAL
```

¹ We're talking *type level* here, so a Role is a type, its binding is a requirement on actual bound instances.

ST stands for Single Type or Simple Type. When we describe roles, the parameter `a` is bound to a `RoleType`.² Empty and Universal are edge cases of, respectively, a role without properties and a role with all properties. The former we use to specify that no restrictions hold, on binding a role; the latter says that nothing can bind to a role³. In the syntax we use the keyword `NoBinding`.

Ordering and type specificity

The compiler checks the validity of bind assignment statements. Such a statement, in the right hand side of a rule, is the intention of a user to have his bot make an assignment if the conditions are right. But is this binding allowed? Obviously, when the type of the required binding equals that of the binding, the answer is yes. But we should allow the binding, too, when the type of the required binding is *less specific* than that of the binding.

In Semantics Of The Perspectives Language we have defined specificity as the relation between the property sets of Roles: 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)

Property Sets

Specificity is defined in terms of the Property sets of Roles. This means that we must have a way to compute the Property set of an ADT. Here is an algorithm:

1. For an ST ENR `EnumeratedRoleType` just take the role's own properties.
2. For an ST CR `CalculatedRoleType`, retrieve the calculation of the role and compute the property set of (the ADT in) its range.
3. For a Sum type, form the intersection of the properties of the terms. If one of the terms is Empty, the intersection is Empty. Ignore Universal.
4. For a Product type, take the union. If one of the terms is Universal, the union is Universal. Ignore Empty.

Because we have to handle Empty and Universal, we cannot represent a Property set merely by an Array or a List. Hence we define the following type:

```
Data PropertySet = Universal | Empty | PSet (Array PropertyType)
```

Other sets: views and aspects

What are the Aspects of an ADT? This is a relevant question for a modeller. Each `EnumeratedRole` can have Aspects, much like it can have properties. Hence, we can re-use

² Not just `EnumeratedRoles`, `e` may want to bind to a `CalculatedRole`. Consider the case where you want to restrict a binding to grown up participants (bind to a filtered role). Or to parents (Sum of `Father` and `Mother`).

³ Because all properties include the property without values and no role instance can have a value for that.

the algorithm we used for computing a `PropertySet` for computing the set of Aspects of an ADT.

For Views, the situation is slightly different. We handle `ST` and `Product` just like with properties, but the Views of a `Sum` is not just the intersection of the Views of its terms. Rather, we have to compute the `PropertySet` of the `Sum` and then use it to filter each View in the *union* of the view set per term. This is because we have only use for a View if the role it is applied to, has all the properties it wants.

Functions on ADT's

The Description Compiler translates the Abstract Syntax Tree that results from parsing a Query into a description of a function that, in its turn, can be translated into a function that actually runs the Query. While doing so, the Description Compiler checks whether the result of applying a query step is suitable as input for the next step. That is, it compares the range of a step with the required domain of the next step.

It therefore must be able to determine the range of a `querystep`, given the domain that is supplied to it, and the type of the function that is applied.

To make this less abstract, consider the query step type *binding*. Given a particular Role, what is the binding of that role? This is easy; we just look it up in the Role's definition. But what if we then again take the binding of the result? Then we have to compute the binding of an ADT, because binding types are represented as ADT's. In this chapter we explain how the five relevant functions work out for the various ADT type constructors.

The functions are: from context to role and vice versa; from role to binding and vice versa; and from role to property.

RoleX: From Context to Role

The simple case is `ST ContextType`. To take a particular role of such an ADT simply yields that Role type: `ST RoleType`.

If the modeller wants to take the role from `Empty` we have to throw an error. After all, `Empty` here means a `Context` without roles.

By similar reasoning, taking a role from `Universal` succeeds and yields the same result as with `ST ContextType`. However, notice that this would only work if the Role type has been specified completely by the modeller⁴.

⁴ But in reality this case will never arise. To see why, we must ask ourselves under what circumstances the Description Compiler would have to handle a domain of `Universal`. `Universal` is only introduced by the modeler as a requirement on role binding. He thereby specifies that that role can never be bound. So `Universal` is the range of the function `binding`, when applied to such a role. But, by definition, if the modeler does so, we signal an error. So no function will ever return `Universal`; and so the Description Compiler will never meet `Universal` as domain.

A role from a Product succeeds if the role is in the union of the roles of the terms of the Product. But what if there are multiple roles in that union that match a local role name? Then we can return a Product of those roles.

A role from a Sum of Contexts only succeeds if the role is in the intersection of the roles of the terms. Notice that this is only meaningful for local role names: each Context in the Sum must have a Role with that local name. Again, if multiple roles match a local role name, we return a Product of those roles.

Context: From Role to Context

The simple case (`ST RoleType`) is obvious.

For Empty, we return Empty. This is because we know nothing about the Empty role. To make this concrete: suppose no binding requirements have been set for a Role. We are allowed to take its binding, but from then on we are ‘off the chart’. Any role might be bound to such a Role, so any context can be the result of taking the context of its binding. And the only thing we know about every context type is that it is a subtype of Empty.

For Universal it can only work for fully qualified role types (but we have seen that the case will never arise).

What is the context of a Product of Roles? It is the Product of each Role’s context and that can be considered to be a ‘super-context’, holding all roles of the terms.

What is the context of a Sum of Roles? Again, it is just the Sum of Contexts.

Binding: From a Role to its binding

We find the binding of `ST ENR EnumeratedRoleType` simply by looking it up in the definition of the `EnumeratedRole`. For a `CalculatedRole`, we look up its calculation and take the binding of the ADT that is in its ranges.

`binding` applied to Universal is simple, as the modeller uses Universal to stipulate that no binding is allowed. Hence, the modeller should not take the binding of Universal, so we throw an error.

Similarly, `binding` applied to Empty just results in Empty. Anything goes.

For Sums and Products, we construct the Sum of the bindings and the Product of the bindings, respectively.

A word on normalisation. There is no need to create a normal form of an ADT (e.g. Conjunctive Normal Form), though we could. However, if, on creating a Sum or Product,

⁵ This range must be of the form: `RDOM (ADT RoleType)`.

we detect `Empty` or `Universal` among its terms, it is advisable to normalise the result⁶. Thereby we avoid unnecessary work later.

BinderX: from a role to its binders

The function `binder` takes an argument that identifies a role type. The Description Compiler checks if this is a legal move by looking up the required binding in the definition of that `EnumeratedRole` or `CalculatedRole`. The domain of the binder function (the assumed binding, an `ADT RoleType`) must be equal to or more specific than the required binding.

It is important to understand that we never try to compute the binder of an arbitrary ADT. That has no meaning.

Notice, however, that the ADT that results from the function `binder` is always of the form `ST ENR EnumeratedRoleType`. This is because only enumerated roles can bind other roles.

PropertyX: from a role to a property value

We can think of Properties in abstract terms, too. But a Property has no sub-parts, as do roles. Instead, a Property is characterised by its Range. However, we have also found⁷ that we cannot ignore the Property name itself. It is the carrier of semantics we cannot afford to lose. Hence we have the Description Compiler handle abstract descriptions of Properties in terms of

- Its name, and
- Its range.

We can have abstract data types constructed from these pairs.

The Description Compiler constructs descriptions of functions on Property values. For example, it may construct a function that adds the values of two properties, or counts the number of values. It guards the compatibility of these functions with the ADT's that describe the properties.

So how do we construct an ADT if we take the property value of an arbitrary `ADT RoleType`?

The simple case is `ST ENR EnumeratedRoleType`. We find the ADT of the results of the property-taking function by looking up the range in the definition of `EnumeratedRoleType`.

⁶ A Product with `Universal` is just `Universal`; we can leave out `Universal` from Sums. We can leave out `Empty` from Products, and a Sum holding `Empty` is just `Empty`.

⁷ Not explained in this tekst.

For an `ST CR CalculatedRoleType` we take the range of the calculation and work from there (i.e. compute the ADT of that `ADT RoleType`).

Taking a property of `Empty` should result in an error. The empty role has no properties.

Taking a property of `Universal` should be allowed, as the `Universal` role has all properties. On the other hand, this case will never occur. The modeller uses `Universal` to stipulate that no binding can exist for a particular `Role`. Taking the binding of such a `Role` results in an error. Hence we can never arrive in the situation that `Universal` is the domain of a function.

Taking the property of a `Sum` of roles succeeds only if the property is in the `PropertySet` of that `Sum`. If multiple `Role` types have names that match with a local property name, the result will be a `Product` of those properties.

Similarly, taking the property of a `Product` of roles succeeds only if the property is in the `PropertySet` of that `Product`. Again, if multiple `Role` types in that set match the local name of a property, we combine those properties in a `Product`.