Query Inversion

Joop Ringelberg 30-12-19 Version: 3

Introduction

In the text *Perspectives across context boundaries* we've described how computed roles and properties can extend across the border of a context. Such *queries* reach out of context and bring roles and values in the perspective of a user agent. As a user agent can delegate actions to a *bot*, the bot needs to be aware of state changes that would cause new query results. A bot is programmed as a set of *rules*, where the left hand side of the rule – its condition – is a query. In short: a state change outside a context may trigger a bot into action.

The question is: how do we implement this mechanism? The situation is further complicated because at any moment, just a fraction of the contexts in which rules may be triggered are in the computer memory. By far the greatest number will reside only on disk. In the context of this text we'll call this the *sleeping context problem*

Inverting queries

Imagine a query as a piping system, fanning out from some context with a bot (and thus rules). Call it a *bot (or user) context*. Role instances and property values 'flow' from *source contexts* through the pipes to the bot context and the combined and filtered result ends up as a Boolean value[1]. It is as if the bot context 'pulls' items towards it.

Now switch your point of view and 'look through' the same pipes from the other end. Looking from some source context, we see bot/user contexts at the other end. In other words, invert the queries. As contexts and roles form a graph constructed from bindings that fans out and fans in, the inverse queries again travel over a subnetworks that are trees. But this time these queries will pull in contexts with bots.

This solves the sleeping context problem. A change to a context or role or property can only be realised when that context is in memory. If the Perspectives distributed runtime (PDR) receives a delta (the unit of description of change) from some other PDR, it will first retrieve the affected context or role into the computer memory. Now, if it is a source with respect to the condition (query) of a bot in some other context, it will have inverted queries. The PDR runs these queries and thereby retrieves all bot contexts that depend on it, fetching them from disk if necessary. In the process, it fetches all roles and contexts on the path between them. It then runs the rules in those contexts. The conditions of those rules will pull in further role instances and values. The changed or new item will be one of

[1] In general, queries are computed roles or computed properties. The condition of a bot rule is a computed property with a Boolean value.

them (or it will be missing if it was removed), possibly leading to a different Boolean result from before the change.

A mechanism for inverting a query

Queries are stored in a model file in the form of a *description*[2]. Such a description holds the domain and range of the query function, and a description of the actual computation. It turns out that we can turn this description inside out, as it were, ending up with a description of an inverted query.

Consider the straightforward case of a query that is just a composition of simple steps – a path. The inverse query is just the inversion of each individual step, run in inverse order (starting with the last step first). But can we invert each simple step? Yes we can[3]. Remember that with a query we traverse the network that consists of contexts and roles, connected through role binding. The simple steps are:

- Move from a role to its context;
- Move from a context to some role;
- Move from a role to its binding;
- Move from a role to its binder.

Each role has a single context and a single binding[4]. However, contexts can have many roles and roles can be bound by many other roles. So the inverse function of role-instance-to-context-instance must be a function that is informed with the *type* of the role instance. The same holds for the inverse function of role-instance-to-binding.

But this information is available in the query function description, so we can draw up a description of the inversion of each simple step.

A word on cardinality. If the original query condition moves from a context to the instances of a particular role, the path 'fans out' over multiple instances. However, the inverse path will come from a single instance. In contrast, if the original query moves from a role instance to a context, the inverse query will fan out. This is not a problem, because queries have sequences of values as result. So queries are particular functions, with multiple results.

Besides simple composition of steps, we have (very few) functions that combine simple paths through the context-role network, filter being the most prominent example. In a filter operation, the results of one path are filtered by the outcome of the result of another path.

[2] See module `Perspectives.Query.QueryTypes`.
[3] Except for roles computed by *external* functions that have no inversion.
[4] This is not so in the generalised version of Perspectives where a role can have multiple bindings.

Where we store inverted queries and how we use them

Where we store

Queries are inverted in design time, as we process a model. As we follow an inverted query back to its origin through the web of types, *we store the remaining query with the node we've arrived at*, along the way.

We can pass through a role instance node in a number of ways:

- From its context, using the `role <type>` step;
- To its context, using the `context` step;
- From its binding, using the `binder <type>` step;
- From a role that binds it, using the `binding` step.

We only store inverted queries with Role types. We let the type of the first step of the inverted query determine the member of the type where we store it: so if the first step is `binding`, we'll store the inverted query in `onRoleDelta_binding`.

The step from a context to a role must be the odd man out (we don't store inverted queries in context types). Here, we store the inverted query with the Role type *that the first step takes us to* (the `role <type>` step takes us from a context instance to a role instance: we'll store the inverted query in the onContextDelta_role of the type of that role instance).

The table below gives the overview for all four steps.

step type of inverted query	query stored in	Query stored in node
binder R	onRoleDelta_binder	of departure
binding	onRoleDelta_binding	of departure
context	onContextDelta_context	of departure
role R	onContextDelta_role	of arrival

Figure 1 illustrates all four cases.

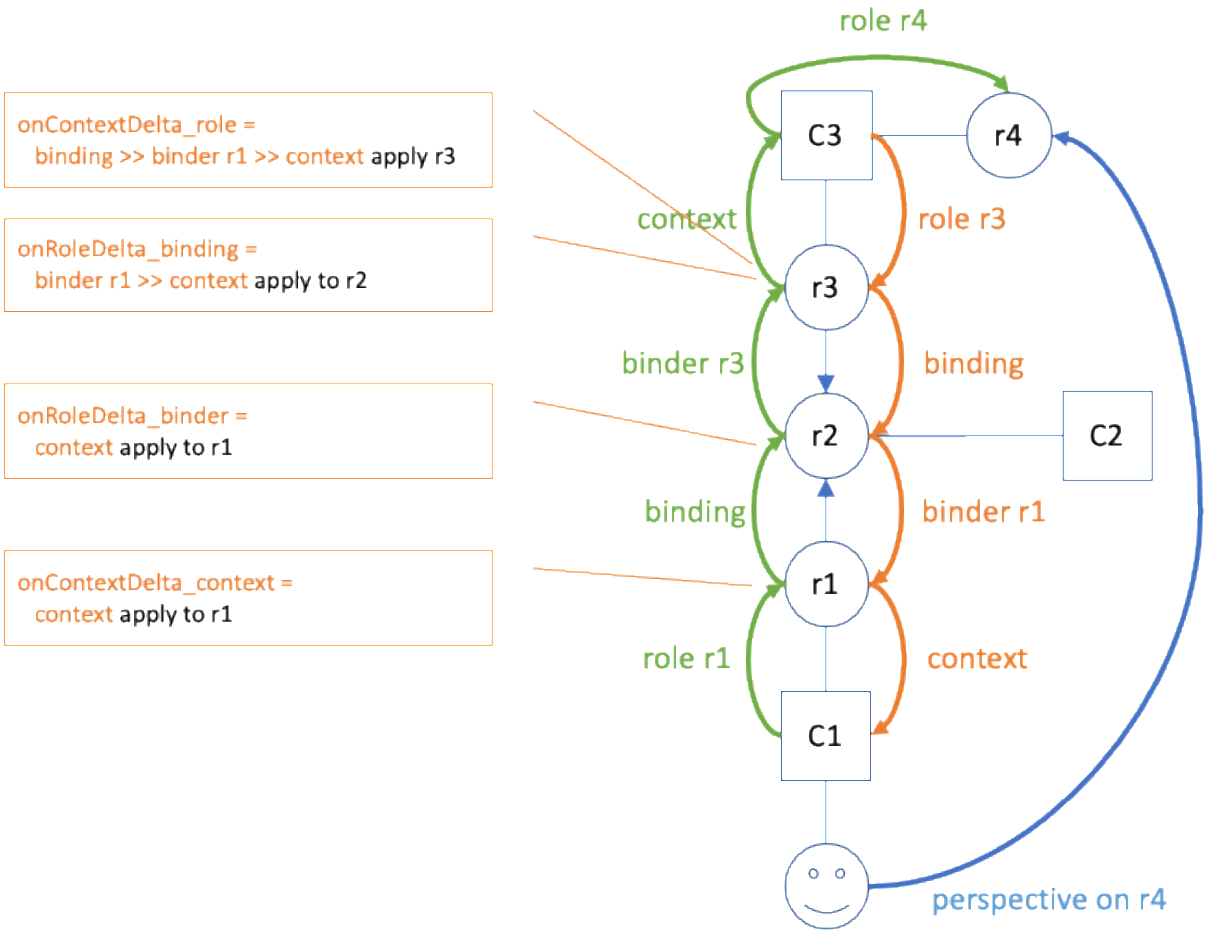

Figure 1. Inverted queries in relation to various nodes in the graph. Green lines and text represent the original query; red lines and text represent the inverted query. The user has a perspective on role (type) r4. Blue lines connect a binder to its binding. The boxes show inverted queries as stored in various members of role types.

What we store and what we apply it to

Consider the example of the inverted query stored in `on_contextDelta_role` of `r3` in Figure 1. The query step that we would apply to `c3` would have been: `role r3`. So we would expect `on_contextDelta_role` to hold the full query

```
Role r3 >> binding >> binder r1 >> context
```

That will take us from `c3` to `c1`, as intended. Yet, as the diagram shows, we skip the first query step (storing just `binding >> binder r1 >> context`) and apply it to `r3` (instead of `c3`). Why?

We will apply the inverted query when we handle a `ContextDelta`. Let's assume the delta represents a new instance of `r3`. Now the *whole point* of applying the inverse query is to find contexts and roles that are now available to the user having the perspective, but were not so before. In other words: a new path has been formed and we want to travel that to its root. Obviously, the new connection must be part of the path we travel. But

then we should start at the new instance of `r3`! Otherwise, on starting with `c3`, we *would also travel down all paths that begin with siblings of the new instance*.

Hence we shorten the query and start at the new role instance.

A similar consideration holds for the inverted query stored in `onRoleDelta_binder` stored in `r2`. Instead of applying the full inverted query to an instance of `r2`, we apply the shorter version to the new binder of type `r1`. This is because there can be many binders of `r2`!

This shortening-and-skipping does not hold for queries in `onRoleDelta_binding` and `onContextDelta_context`, because these steps are functional: there is always just one binding and just one context. No confusion can arise. So you see it is the cardinality of the step that determines how we handle it.

Implementation complication: two types of trees

Module `Perspectives.Query.Inversion` holds the code that actually inverts a query function description. This code deals with a complication. In this document, we've imagined query execution to trace a path through the graph of instances of contexts and roles, fanning out from a single bot context to many source contexts. Why the branching? Because of two reasons:

1. a context may have many instances of a role type;
2. a role may be bound by many other roles.

In other words, the path traced by executing a query stands out as a tree selected from the underlying graph of context- and role instances.

However, the way back from a source context (a path endpoint) is always a straight path without branches to the bot context (the path starting point).

Confusingly, the *description of a query itself can have a tree-shape*. This is a tree selected from the graph of *types* of contexts and roles. Why the branching? Because we have several operators on two arguments, for example:

1. filter
2. join

(Composition is an operator on two arguments, too, but we use it to construct a single path through the graph of types).

Being a tree, its inverse is, again, a collection of paths. This time, however, these are paths through the underlying graph of types of contexts and roles.

Some cases

Variables

`letE` and `letA` expressions introduce variables. Furthermore, in calculated properties the variable `object` is automatically bound to the current object set and in calculated roles we have the variable `currentcontext`. How should we treat an expression using, for example, this `object` variable? Consider:

```
perspective on: SomeRole
     on entry
        bind object >> binding to AnotherRole
```

If we invert the sub-expression between `bind` and `to`, we should get

```
binder SomeRole >> context
```

in order to arrive at the context of this rule from the role (whatever it is) that is being bound by it. Explanation:

1. the `binding` step inverts to `binder SomeRole`. `SomeRole`, because that is the type of the object of the perspective (it is the type of the step `object`).
2. the `object` step itself inverts to `context`, because *underlying the object variable* is the expression `SomeRole`, evaluated in the current context. That is how we arrive at the value of `object` (the inverse of `SomeRole` is `context`).

This gives us a recipe for the general case in which a variable is bound to an arbitrary expression. Substitute the inverted expression that defines the variable into the syntactical location occupied by the variable.

So while we invert queries, we add bindings to the compile time environment. Because the same variable name can be re-used arbitrarily often, we push a compile time frame before each block[5].

Can we look up the variables, in compile time?

In compile time, we store with the name of a variable a description of a function that will compute its value (an instance of `QueryFunctionDescription`): a *compile time variable binding*. A variable has a limited *visibility*; we will call the area of Perspectives Language code where we can refer to the variable, its *scope*. There are two scopes we have to consider:

[5] In the perspectives language, we can use LetE and LetA. This translates to a QueryFunctionDescription with function name WithFrame. The query inversion code pushes a frame as it encounters this instruction. The variable bindings that follow, lead to additional bindings in this frame. Finally the expression (or statements) in the body of the LetE or LetA are inverted in this environment.

- the right hand side of a bot rule. It is the scope of the `object` variable.
- the `letE` or `letA` expression. Each binding (from left to right or top to bottom) introduces a new scope: for the rest of the expression (i.e. the rest of the bindings and the body).

Scopes may be nested. We keep, in the state of the compiler, a stack of `Environment`s to reflect that recursive structure. An Environment is a collection of compile time variable bindings. We introduce, in our Purescript code, a new Environment with the function `withFrame`. The argument to `withFrame` is a computation with state in which we save variables and their (compile time) binding.

This makes it as if we can read the Purescript code as a lexical Perspectives Language scope: the computation (Purescript) corresponds to a particular scope (PL).

It so happens that we invert all queries that can hold variables exactly in the `withFrame` computations that hold their definition, meaning we have all variables in scope: we can actually look them up and find their `QueryFunctionDescription`.

Treatment of properties

Consider a somewhat degenerated Calculated property:

```
property P1 = P2
```

We should invert this expression, for two reasons:

- if P2 changes, every user with a perspective on P1 should be informed (synchronisation);
- if P2 changes, P1 changes and it might be (part of) the condition of a rule somewhere.

So how do we go about it? The update function that actually changes the value of property P2 on a role, obviously has access to that role. We do not need to trace a path back from the property value to the role; property values are represented on role instances. In other words, to move from a Value to a Role is a no-op. On inverting queries, we represent this operation explicitly, because it carries type information:

```
Value2Role Propertytype
```

But an inverted query should yield contexts, not roles. Hence, for the update function to find the context in which a property has changed from the role on which it is represented, the no-op is insufficient. It needs to be followed by the `context` step. So, on inverting a calculated property, we postfix the context step on the inversion of the expression.

Functions that operate on values

Consider:

```
thing: SomeRole
```

```
property Sum = Prop1 + context >> AnotherRole >> Prop2
```

Can we invert that? We've seen above how we invert an expression that consists of just a single Property, so that deals with the first operand. If we invert the second operand, we get:

```
Value2Role Prop2 >> context >> SomeRole
```

Why `SomeRole`? Because the property is defined on it. Visualise the original query path, as it moves from `SomeRole` to its context, then to `AnotherRole` and then to `Prop2`. Moving back, we start with the no-op Value2Role ('arriving' at `AnotherRole`), then we move to the context, *and then we have to move back to* `SomeRole`.

But we're not done yet, because we need a context as the result. In fact, we're in exactly the same position as with the simple property `P1` defined in the previous paragraph. So the easy solution is to postfix the inversion with a `context` step:

```
Value2Role Prop2 >> context >> SomeRole >> context
```

It is glaringly obvious we could, alternatively, have removed the last step of the original inversion, too:

```
Value2Role Prop2 >> context
```

This is an implementation detail.

So we now have two inverted queries for our two operands:

```
Value2Role Prop1 >> context
Value2Role Prop2 >> context
```

The first will be used when `Prop1` changes value; the second when `Prop2` changes value. Both will return contexts of the same type.

And we're done with that. The (+) function does not change anything: it does not 'move' over the underlying graph of context and role instances. The end result of the application of the function `invertFunctionDescription` (module `Perspectives.Query.Inversion`) is an instance of `Paths`, the representation of a series of query paths (see the previous chapter for an elaboration).

Join queries

We can join the result of two (role) queries:

```
property: Channel = (binder Initiator union binder ConnectedPartner) >>
context >> extern >> ChannelDatabaseName
```

The sub-expression `(binder Initiator union binder ConnectedPartner)` has a Sum type.

We invert queries of this type by treating them as two separate queries:

```
binder Initiator >> context >> extern >> ChannelDatabaseName
```

```
binder ConnectedPartner >> context >> extern >> ChannelDatabaseName
```

Both can be simply inverted.

Filter queries

A filter combines a source and a criterium:

```
context: UnloadedModel = filter ModelsInUse with not available (binding >> context)
```

Again, we invert these queries by considering them to be two separate parts:

```
ModelsInUse
ModelsInUser >> not available (binding >> context)
```

Functions with arguments

A function like available takes an expression as argument. On inverting, we just ignore the function. So we treat

```
ModelsInUser >> not available (binding >> context)
```

just like

```
ModelsInUser >> binding >> context
```

(both `not` and `available` are functions with a single argument). Functions with more than one argument just lead to multiple queries, as with the `join` and `filter` operators.