Indexed Names

Joop Ringelberg 06-04-20 Version: 1

Introduction

Indexing is about instances. With an indexed role, we provide a *universal name for an instance*. However, that name resolves to a *different instance for each user*. Indexing is a mechanism restricted to functional roles (which of the instances of a non-functional role would the indexed name resolve to?

How do we procure an indexed role? The answer is that we create it in a CRL file that belongs to the model. In other words: an indexed role must be a model-instance.

We have two use cases for Indexed Names:

- modellers may use them in queries (e.g. to retrieve a subset of one's hobbies)
- end users may use them to navigate.

In both cases, we want the Indexed Name to be effectively replaced by a unique name, so the unique (indexed) resource can be retrieved. We will focus on the use in queries. We can extract three requirements:

1. We have to be able to recognize a name as 'indexed' on parsing a query;
2. We have to be able to look up the type of the resource identified by the indexed name, because, on compiling a query, we make a description of the composition of a series of functions;
3. We have to actually produce a function as the compilation result of the querystep, that, indeed, looks up the indexed name and comes up with the name of a unique resource.

Uniqueness

An indexed name is qualified with a model name. Within a model, the local part of such a name must be unique. Model names must be unique, too[1], so indexed names are unique.

Private-ness

An indexed context or role is not a *private* context or role, where we mean with 'private' that only one user has a perspective on it[2]. Actually, we have to be careful when speaking

[1] In order to ensure this uniqueness, we will have to replace model names by GUIDs in the future. A model name is like an indexed name: a particular user will attribute a single value to a readable model name. Notice, however, that he may encounter another model with the same name. This is a problem to be solved.
[2] Note that perspectives are on *roles*, rather than contexts. Here we use the term loosely, actually meaning a perspective on the *external role of the context*.

about perspectives in relation to *instances*. Imagine a context type with a single, functional user role. We give that role a perspective on the context. Now, any instance of this context type will be private – at least if we do not give a perspective on it to user roles in *other* contexts. It is private, precisely because the user role having the perspective is functional. Were it not, there could be *two* (or more) users in the same context and it would no longer be private. So we reframe our definition of 'private' to: only one *functional user role* has a perspective on it.

As a matter of fact, an indexed role or context should be indexed in the first place precisely because it is *not* private. Were it private, we could use a universal name for the instance. Each user would have his own instance, never sharing it with anyone else – but all instances would have the same name.

So here are the design rules:

- If we like to use a universal name for a role instance, as a query step, that should resolve to a different instance for each user, make the role type indexed;
- If the role is private (just a single, functional user role has a perspective on it) it need not be indexed.

In both cases, construct the instance in design time, with the model.

Modelling indexed names

To begin with, we extend the language with another keyword: `indexed`. We use it in the definition of Contexts and Roles:

```
case: PerspectivesSystem
   indexed: sys:MySystem
   ...
  user: User (mandatory, functional)
   indexed: sys:Me
```

The keyword is followed by the name we intend to be indexed. This name must be qualified with the namespace of the model that is being defined. So here we would have `sys:Me`, or, expanded, `model:System$Me`.

Representation

We extend the data type Context that we represent types of contexts with, with another member: `indexedContext`. It's type will be `Maybe ContextInstance`. Here we store the indexed name itself, e.g. `ContextInstance model:System$MySystem`. Context types that are not indexed will have `Nothing` as value for this new member.

Similarly, we extend the data type `EnumeratedRoleType` with a member `indexedRole`. Note that Calculated Roles cannot be indexed.

Two useful functions

We construct (in module Perspectives.Instances.Indexed) two functions that will extract all indexed contexts and roles from a model file:

```
indexedContexts :: DomeinFile -> Object Context
indexedRoles :: DomeinFile -> Object EnumeratedRoleType
```

We use these functions on compiling models and queries. Implementation is straightforward: e.g. run through all EnumeratedRoleType definitions and discard the ones that have Nothing for their `indexedRole` member.

Notice how we provide two bits of information in these function results:

- The keys are the indexed names themselves, as they are stored in the members `indexedContext` and `indexedRole`;
- The values are the *types* of the indexed resources.

Usage in queries

An indexed name, let's say `sys:Me`, may occur in a query. It can function as a complete query in itself. When used, it should be properly qualified, so either with a valid prefix defined in the model for the right namespace, or as an expanded name. Semantically, it should be thought of as a query producing a singleton result.

Because an indexed name in a query is always qualified, from the text we know exactly in which model it is defined.

Compiling a query with an indexed name

As we encounter qualified names in queries, we have to establish whether they represent a type, or an indexed individual. We cannot say from the name only. So we find the model the name is qualified with, apply the functions `indexedContexts` and `indexedRoles` defined above, and look the qualified name up in the tables produced by both (function `compileSimpleStep`, case `ArcIdentifier`, of `Perspectives.Query.DescriptionCompiler`).

If the name is, indeed, indexed, we produce a query step that describes a function that will, in runtime, look up the indexed name. Note we cannot do that in compile time! There must be, by definition, a different result for each user.

However, the description of this step must contain the type of its result. So we must know what type of role or context this indexed name represents. This is exactly what is returned by the two functions above.

Starting to use a model

When a user first starts with a model, its indexed contexts and roles must be created. Like all other context- and role instances that come with a model, they are defined in a CRL file. However, in their definitions, indexed names are used. So before we parse the CRL file and create instances, we have to replace the indexed names by unique names. This we do by

1. Applying the functions `indexedContexts` and `indexedRoles`, to retrieve all indexed names;
2. Creating a GUID for each of them, building a translation table;
3. Replacing each indexed name in the CRL text with stringbased search and replace.

Notice that we have to replace *all* indexed names in the model. The CRL file may refer to indexed names declared outside the model and we should replace them. We introduce a convention: only indexed names written with a prefix (that should be declared in the CRL file) will be replaced. Fully qualified names will not be replaced[3].

Looking up indexed names in runtime

As stated at the start of the text, we have two use cases for Indexed Names:

- modellers may use them in queries (e.g. to retrieve a subset of one's hobbies)
- end users may use them to navigate.

Let's again focus on the first use case. When the PDR starts, it must build a translation table. This table is used by the functions that our querysteps are compiled into. So how do we build that table?

Notice that there is nothing special about the representation of an Indexed context or Indexed role. They are not even singletons, even though there is just one instance for each user. For example, `sys:Me` represents an instance of `sys:PerspectivesModel$User`, but we have many of them in a PDR installation; one for each peer and the indexed one. But which is which?

We solve that problem by requiring the modeller to list the indexed resources in the model description. We enable him to do so by introducing two roles in `sys:Model`:

```
context: IndexedContext filledBy: sys:NamedContext
thing: IndexedRole
```

Notice how we have put no restriction on the binding of `IndexedRole`. Similarly, `sys:NamedContext` is a very lightweight context type, merely requiring a name.

[3] Rationale: it is pretty hard to detect fully qualified indexed names in a text string. We would have to recognize qualified names, to start with; then to retrieve their model part; then to retrieve all indexed names in those models and then check each qualified name. So as an optimization we require indexed names to be prefixed in the CRL file.

On starting the PDR, we run two queries on Couchdb that produce all instances of these two roles. This gives us all the information needed for our translation table:

- the actual resource identifier is the unique value for this user;
- from the resource we retrieve its type and from the type we retrieve the indexed name itself.

From this we build a table with indexed names as keys and unique identifiers as values.

How to write the right instances

The success of this system depends critically on writing the right instances in the CRL file that accompanies the model file.

We require two things:

1. For each context type that is declared `indexed`, we require a binding for the role `sys:Model$IndexedContext` in the model instance in the CRL file.
2. For each role type that is declared `indexed`, we require a binding for the role `sys:Model$IndexedRole` in the model instance in the CRL file.

Notice that the *type* of the model instance in the CRL file is a unique specialisation (through the use of `aspect model:System$Model`) for each model.

IndexedContext

We furthermore require the following for the binding of `IndexedContext`:

1. The name of the context instance we bind to, must be the indexed name! So, for example, in `model:System` we would have the name of this context instance to be `sys:MySystem`; for `model:SimpleChat` it would be `chat:MyChats` (the prefixed name is allowed).
2. The role must have a value for the property `IndexedContext$Name`. That name must be, again, the indexed name – however, **without the `model:` part**. So we would have `SimpleChat$MyChats` and `System$MySystem` respectively[4].

IndexedRole

The same requirements hold for IndexedRole:

1. The name of the role instance we bind to, must be the indexed name! So, for example, in `model:System` we would have the name of this role instance to be `sys:Me`.

[4] The reason for this is rather arcane: we do textual search and replace on the string value of the crl file, replacing all occurrences of the indexed names. However, here we need a hook back from the indexed name to its replacement. By omitting the "model:" part, we prevent replacement of the property value.

2. The role must have a value for the property `IndexedRole$Name`. That name must be, again, the indexed name - however, **without the `model:` part**. So we would have `System$Me`respectively.

Example

The relevant fragment of the CRL file for `model:SimpleChat` is this.

```
chat:Model usr:SimpleChatModel
  …
  sys:Model$IndexedContext =>
    chat:ChatApp chat:MyChats
      …
    sys:Model$IndexedContext$Name = "SimpleChat$MyChats"
```

Notice

- The specialised type `chat:Model`;
- The indexed context name `chat:MyChats`;
- The value of the property `IndexedContext$Name`: it is the (expanded) indexed name without the “model:” part: `SimpleChat$MyChats`.