Technical design standard variables

Joop Ringelberg 22-09-21 Version: 1

Introduction

The modeller can use a number of standard variables in his expressions. The semantics of these variables is given in the text *Understanding a Perspective Model Text*, but here is a summary:

- `origin` can be used in every expression. It is the role or context instance the expression is applied to.
- `currentcontext` can be used in every expression. It is the lexical current context of the expression.
- `currentactor` can be used in the scope of `do` and `action`. It is the lexical current subject.
- `notifieduser` can be used in the scope of `notify`. It is the lexical current subject.

In this text I explain how these variables are implemented.

Lexical concepts in the state of the parser

The parsers keep current context, current subject and current object in state. This means that any parser can retrieve a representation from state that represents these lexical concepts for the lexical position it starts to work on (if they are defined at all, of course, at that position).

Current context

The current context is represented in `ArcParserState` (module Perspectives.Parsing.Arc.IndentParser) as the member `currentContext`. Every parser can extract the lexical current context from state. For our purposes we need to know that this current context is part of the representation of current subject, current object and current state. The representation is a fully qualified type.

Current subject

In the body of the following constructs, a current subject is defined:

- the body of a `perspective of <user>`;
- the body of a `user` role definition;
- the block of assignments following `do for <user>`, defining an action
- the expression following `notify <user>`.

In ArcParserState, the current subject is represented by a RoleIdentification:

```
data RoleIdentification =
  ExplicitRole ContextType RoleType ArcPosition |
  ImplicitRole ContextType Step
```

The ContextType part represents the current context (as stated in the previous paragraph). For ExplicitRole, this means that RoleType is defined in the context of the ContextType. In an ImplicitRole, the Step is the parse tree of an expression. It should be evaluated with respect to its ContextType: we interpret an expression in its current context.

It turns out that in all of the situations above, an ExplicitRole is constructed[1]. In all cases *except for* perspective of <user>, the RoleType part is a fully qualified name[2].

ArcParserState holds a member subject that represents the lexical current subject.

A note on calculated roles. The role type in ExplicitRole can be calculated (except when it is the external role) and will then be a CalculatedRoleType.

Current object

The current object is in scope in:

- the block following a perspective on;
- the body of a non-user role definition.

As with current subject, current object is represented in the parse tree by a RoleIdentification. A role definition results in an ExplicitRole with a fully qualified name, but a perspective on results in an ImplicitRole.

ArcParserState holds a member object that represents the lexical current object.

Lexcial concepts in parse tree elements

The lexical concepts relate to the parse tree in a straightforward way. We discuss the case of AutomaticEffectE, the lexical representation of do (module Perspectives.Parsing.Arc.AST).

current subject and current object

The newtype AutomaticEffectE has members subject and object. The former is a RoleIdentification; the latter a Maybe RoleIdentification (not every do has a current object in scope). The RoleIdentification that is subject is precisely the value

[1] In fact, as shown below, an ImplicitRole is only ever constructed on parsing a perspective on clause.
[2] We qualify this case in PhaseThree of the parser, when all role types of the model are available.

of the member `subject` of ArcParserState as it was available to the parser handling the body of the `do`. In other words, it represents the current subject in the scope of the `do`.

Mutatis mutandis, the same holds for `object`.

Current context

The `AutomaticEffectE` is constructed with a member `transition` whose value is a `StateTransitionE`:

```
data StateTransitionE = Entry StateSpecification | Exit StateSpecification
```

A `StateSpecification` is:

```
data StateSpecification =
       ContextState ContextType (Maybe SegmentedPath)
     | SubjectState RoleIdentification (Maybe SegmentedPath)
     | ObjectState RoleIdentification (Maybe SegmentedPath)
```

Let's dissect the three data constructors. For a `ContextState`, the `ContextType` represents the current context from lexical analysis.

For a `SubjectState` and `ObjectState`, the `RoleIdentification` contains (as we've seen above) a part `ContextType`, that represents the current context.

How do we know? The parsers keep members `onEntry` and `onExit` in `ArcParserState`. These end up in `AutomaticEffectE`. They are built from `currentContext`, `subject` and `object` in `ArcParserState`: meaning they are constructed from the representation of the corresponding lexical concepts for the body of the `do`.

Other lexical representations

We've discussed `AutomaticEffectE` above in detail. It happens that the following lexical representations exhibit the same pattern of a subject, object and transition, or, in some cases, a state member:

- `RoleVerbE`
- `PropertyVerbE`
- `SelfOnly`
- `ActionE`
- `NotificationE`
- `AutomaticEffectE`

The first three represent various qualifications of a perspective. The latter three have to do with state transition, assignments and notification.

Standard variables: the lexical concepts in QueryFunctionDescriptions

Expressions are parsed to `Step` and `Step` is compiled to a `QueryFunctionDescription`. Runtime, a `QueryFunctionDescription` is compiled to an executable function. We've seen above how the lexical concepts current context, current subject and current object are represented in parse tree elements. How do these representations relate to standard variables?

The expression compiler

First, a quick recap on `QueryFunctionDescription`. These represent functions, with a domain and range and some information on functionality and being mandatory, along with a representation of the actual function to compute. The expression compiler (module Perspectives.Query.ExpressionCompiler) uses these representations to reason about the correctness of the function expressions entered by the modeller (it checks, among others, whether properties that are used do in fact exist on the roles they are supposed to be on, whether both sides of (in)equalities have the same type, etcetera).

Inserting standard variables into expressions

Standard variables are just like the variables that the modeller himself can introduce, in a `letE` or `letA` construct, but he need not bind them himself. Consequently, expressions will only have *references* to those variables but never their *bindings*. This would cause the expression compiler to throw an error. To prevent these errors, we *add bindings automatically for the standard variables to the parse tree of expressions*. This allows the expression compiler to reason about their use as usual.

These bindings are added to existing letE and letA constructs, or an expression or statement is wrapped in a letE to hold the new bindings. We don't actually always add all standard variables to expressions: the code analyses the expression to see what standard variables appear.

By modifying the expression itself, we can evaluate it in the standard way in any location in the code of the core. We guarantee that each expression is self-contained: the core code does not have to add standard variables on computing the value of an expression.

This is very convenient, as calculated roles and properties can be thought of as named functions that are, indeed, called by name from other queries. We can therefore freely compose such functions without the need to add bindings to the runtime environment.

Treatment of statements

Statements are treated differently. We do always add standard variables to statements (single statements or sequences of them, or a letA construct)[3]. This is because when we execute an automatic action or a notification, the core code itself needs the value of these variables. The origin is available anyway, because it is the argument that the compiled expression function is applied to. But in order to distribute delta's for the changes incurred by an automatic action, we need to know the current actor. And to compute the current actor, we need to have the current context – because the actor is computed relative to the current context. So, before actually executing statements, we have the value of all three standard variables available.

It would therefore be a waste of resources to recompute them with the statements while it introduces almost no overhead to add the already computed values to the runtime environment that holds values for our variables.

We handle this in runtime as follows, in the execution machine (modules Perspectives.RoleStateCompiler and Perspectives.ContextStateCompiler):

1. we let the execution machine push a runtime environment (the data construct that we store variable bindings in);
2. we then add the values of the standard variables to that environment.

But what about the bindings added to the statements? Well, in the very last step that creates an executable function (in module Perspectives.Query.UnsafeCompiler), we remove the bindings for the standard variables. The executable functions will never compute their values, nor add them to the runtime environment. A variable reference will just pick up the value added by the runtime execution machine[4].

Expressions in statements

We just add bindings to entire statement groups, not to each and every individual expression in them. This is because each expression is applied to the same origin and because there are no syntactical constructs within statements that change the current context, subject or object. Neither can statement groups be nested inside other statement groups (do, action and notify do not contain lexical constructs that introduce scopes). Hence a single set of bindings suffices.

[3] This happens in module Perspectives.Parsing.Arc.PhaseThree.

[4] You may have noted that an extra environment is pushed by the compiled statement (we just remove the variable bindings). This is unnecessary, but has no effect: variable lookup scours the entire environment stack, so the empty environment is just passed (adding very little overhead).

Simple treatment of compile time only bindings

As the values of standard variables are never computed with the assignments that they occur in, we do not actually have to construct a full function to compute it. Just having a `QueryFunctionDescription` with the right domain, range, functionality and being mandatory information, will do: it is all the expression compiler needs to do its checks. For that reason we introduce a `QueryFunction` that can be considered a no-op. It is actually this `QueryFunction` that makes the unsafe compiler remove a binding.

Consider the computation of the current context from the origin, for a `do`. Now when we have context state, it is trivial: just the identity function. But when we have role state, we take the current context value from the `RoleIdentification` and put that into this parse tree construct:

```
Simple $ TypeTimeOnlyContext pos <currentcontext>
```

(where `<currentcontext>` is the name of the context type that is the current context) The expressioncompiler will turn it into:

```
SQD currentDomain (QF.TypeTimeOnlyContextF <currentcontext>) (CDOM (ST $
<currentcontext>)) True True
```

This can be used to reason with: whenever `currentcontext` occurs in the statement, the compile knows its runtime type.

Overview: standard variable computing in runtime

What variable should be computed when? With both context- and role situations, actions, automatic actions and notifications, we have a lot of cases. We do not always have to compute a value:

- `origin` never needs to be computed runtime;
- `currentcontext` must be computed for roles;
- `currentactor` (or `notifieduser`) must always be computed.

Actually computing the current context

The execution machine must compute the value of the current context. This is trivial in many cases, as expressions are quite often applied to contexts. However, in a `do` (or `notify` or `action`) things can get difficult if they hold for role state. Remember that the expressions in the body of such a `do` will be applied to a role instance.

Isn't the current context just the context of that role instance? That may be the case, but it is not so if the object- or subject state is for a *calculated role*. The current context should be the lexical context, i.e. the context that one can verify, in the source code, to contain the `do` expression somewhere in its body. But the `origin` that the expression function is applied to, may be a role in an entirely different context. That is the usefulness of calculated roles!

Let's consider the case of object state. The state specification will have a `RoleIdentification` that represents the origin. Consider the case of an `ImplicitRole`: the `RoleIdentification` contains a context type that represents the current context, and a `Step` that represents the expression that computes the role from that context.

To find the context from the role, we just have to reverse the expression. This we can do; we have machinery to invert `QueryFunctionDescriptions` (for triggering state changes and query updates, for example). So we compute a `QueryFunctionDescription` for the `Step` with respect to the current context type, and reverse it.

The execution machine uses that inverted function to compute the context from the role instance that changes state.

Future extensions: currentobjects, currentsubjects

It seems possible to add yet more standard variables. A `do` expression might be in scope of both a current subject and a current object, but the state transition it is contained in will be just one of them (or, of course, it will be on a context type transition).

So, for example, an expression in a `do` in subject state cannot refer to the current object. Yet, as we can compute the current context, we can compute the object from it.

However, if the object role is relational, there may be more than one instance available. This causes a curious phenomenon: when an expression occurs in the body of a `do` in object state, it can refer to *one* object as `origin`, and to *all* objects as `currentobjects`.