Base Language - Feature #5729

implement polymorphic reflection support for properties and variables

10/14/2021 10:58 AM - Greg Shah

Status:	New	**Start date:**	
Priority:	Normal	**Due date:**	
Assignee:		**% Done:**	0%
Category:		**Estimated time:**	0.00 hour
Target version:			
billable:	No	**vendor_id:**	GCD

Description

Related issues:

Related to Base Language - Feature #4384: Builtin OO Implementation	**Closed**
Related to Base Language - Feature #4373: finish core OO 4GL support	**New**

History

#1 - 10/14/2021 10:59 AM - Greg Shah

From Marian in #4384-537:

> Trying to convert some of the tests made for reflection we see the POLY behavior is not implemented for properties/variables get/set methods. In skeleton the return/input parameters are defined as P.L.O although those can be also any base data type like integer, character... Its probably the same approach as for the Class.invoke method and that case seems to be handled by conversion rules. Should that be working already, maybe something we're doing wrong there?
>
> Trying to convert those two files gives a class not found error:
>
> ```
> oo/progress/reflect/property/set_data_type/test_setter_character.p
> oo/progress/reflect/property/set_data_type/SetDataType.cls
> ```
>
> ```
> EXPRESSION EXECUTION ERROR:
> ---------------------------
> nameinfo = loadConvertedClass(fname, false)
>            ^  { Can not find class associated with null }
> ---------------------------
> Elapsed job time:  00:00:04.869
> ERROR:
> com.goldencode.p2j.pattern.TreeWalkException: ERROR!  Active Rule:
> ----------------------
>       RULE REPORT
> ----------------------
> Rule Type :   WALK
> Source AST:  [ Set ] BLOCK/PROCEDURE/BLOCK/ASSIGNMENT/EXPRESSION/OBJECT_INVOCATION/OO_METH_VOID/ @226:10 {
> 983547512819}
> Copy AST  :  [ Set ] BLOCK/PROCEDURE/BLOCK/ASSIGNMENT/EXPRESSION/OBJECT_INVOCATION/OO_METH_VOID/ @226:10 {
> 983547512819}
> Condition :  nameinfo = loadConvertedClass(fname, false)
> Loop      :  false
> --- END RULE REPORT ---
>
>     at com.goldencode.p2j.pattern.PatternEngine.run(PatternEngine.java:1070)
>     at com.goldencode.p2j.convert.TransformDriver.processTrees(TransformDriver.java:576)
>     at com.goldencode.p2j.convert.ConversionDriver.back(ConversionDriver.java:565)
>     at com.goldencode.p2j.convert.TransformDriver.executeJob(TransformDriver.java:985)
>     at com.goldencode.p2j.convert.ConversionDriver.main(ConversionDriver.java:1024)
>     at testcases.Convert.main(Convert.java:22)
> Caused by: com.goldencode.expr.ExpressionException: Expression execution error @1:12 [OO_METH_VOID id=9835
> 47512819]
>     at com.goldencode.p2j.pattern.AstWalker.walk(AstWalker.java:275)
>     at com.goldencode.p2j.pattern.AstWalker.walk(AstWalker.java:210)
>     at com.goldencode.p2j.pattern.PatternEngine.apply(PatternEngine.java:1633)
> ```

```
    at com.goldencode.p2j.pattern.PatternEngine.processAst(PatternEngine.java:1531)
    at com.goldencode.p2j.pattern.PatternEngine.processAst(PatternEngine.java:1479)
    at com.goldencode.p2j.pattern.PatternEngine.run(PatternEngine.java:1034)
    ... 5 more
Caused by: com.goldencode.expr.ExpressionException: Expression execution error @1:12
    at com.goldencode.expr.Expression.execute(Expression.java:484)
    at com.goldencode.p2j.pattern.Rule.apply(Rule.java:497)
    at com.goldencode.p2j.pattern.Rule.executeActions(Rule.java:745)
    at com.goldencode.p2j.pattern.Rule.coreProcessing(Rule.java:712)
    at com.goldencode.p2j.pattern.Rule.apply(Rule.java:534)
    at com.goldencode.p2j.pattern.Rule.executeActions(Rule.java:745)
    at com.goldencode.p2j.pattern.Rule.coreProcessing(Rule.java:712)
    at com.goldencode.p2j.pattern.Rule.apply(Rule.java:534)
    at com.goldencode.p2j.pattern.Rule.executeActions(Rule.java:745)
    at com.goldencode.p2j.pattern.Rule.coreProcessing(Rule.java:712)
    at com.goldencode.p2j.pattern.Rule.apply(Rule.java:534)
    at com.goldencode.p2j.pattern.RuleContainer.apply(RuleContainer.java:585)
    at com.goldencode.p2j.pattern.RuleSet.apply(RuleSet.java:1)
    at com.goldencode.p2j.pattern.AstWalker.walk(AstWalker.java:262)
    ... 10 more
Caused by: java.lang.ClassNotFoundException: Can not find class associated with null
    at com.goldencode.p2j.uast.SymbolResolver.loadConvertedClass(SymbolResolver.java:1992)
    at com.goldencode.p2j.pattern.CommonAstSupport$Library.loadConvertedClass(CommonAstSupport.java:625)
    at com.goldencode.expr.CE6768.execute(Unknown Source)
    at com.goldencode.expr.Expression.execute(Expression.java:391)
    ... 23 more
```

#2 - 10/14/2021 11:01 AM - Greg Shah

Constantin posted in #4384-539:

> Marian Edu wrote:
>
>> Trying to convert some of the tests made for reflection we see the POLY behavior is not implemented for properties/variables get/set methods. In skeleton the return/input parameters are defined as P.L.O although those can be also any base data type like integer, character... Its probably the same approach as for the Class.invoke method and that case seems to be handled by conversion rules. Should that be working already, maybe something we're doing wrong there?
>
> You mean the Progress.Reflect.Property and Progress.Reflect.Variable? These are not implemented in FWD yet.
>
> For the invoke method in Progress.Lang.Class and Progress.Lang.Method, the 'patch' is in ClassDefinition.annotateMethodCall:1580 (this is for

the parser):

```
            if (mdat.name.equalsIgnoreCase("invoke") &&
                (mdat.container.name.equalsIgnoreCase("progress.lang.class") ||
                 mdat.container.name.equalsIgnoreCase("progress.reflect.method")))
            {
               // set the node's type to OO_METH_POLY
               node.putAnnotation("oldtype", (long) node.getType());
               node.setType(OO_METH_POLY);
            }
```

For the conversion, there is also ExpressionConversionWorker.expressionType:1955:

```
               if (oldtype == OO_METH_CLASS && source.getText().equalsIgnoreCase("invoke"))
               {
                  String defClass = (String) source.getAnnotation("found-in-cls");
                  if ("Progress.Lang.Class".equalsIgnoreCase(defClass) ||
                      "progress.reflect.method".equalsIgnoreCase(defClass))
                  {
                     jcls = "BaseDataType";
                  }
                  break;
               }
```

Both cases need to be aware of the Property and Variable APIs which are POLY.

#3 - 10/14/2021 11:02 AM - Greg Shah

And Marian confirmed in #4384-540:

> You mean the Progress.Reflect.Property and Progress.Reflect.Variable? These are not implemented in FWD yet.

> Yes, so no reason to worry just yet... will put those on hold for time being.

#4 - 10/14/2021 11:02 AM - Greg Shah

- Related to Feature #4384: Builtin OO Implementation added

#5 - 10/14/2021 11:03 AM - Greg Shah

- Related to Feature #4373: finish core OO 4GL support added

#6 - 10/14/2021 11:11 AM - Greg Shah

What is the list of items to resolve here?

- Add runtime support to complete Progress.Reflect.Property and Progress.Reflect.Variable for some get/set functionality.
- Modify the conversion in the 2 locations from #57229-2 to fixup the converted code references.

Is there something else to do?

Since the 4GL syntax itself does not have any concept of a polymorphic return type, we have to add special processing to the conversion to handle those built-in runtime POLY features. Considering we may find other cases of this over time, we may want to "mark" references to POLY calls (at parse time) so that we can automatically process them downstream. Otherwise we have to detect each case downstream, which seems to be messy and more error prone.

#7 - 10/15/2021 06:57 AM - Marian Edu

Greg Shah wrote:

> What is the list of items to resolve here?
>
> - Add runtime support to complete Progress.Reflect.Property and Progress.Reflect.Variable for some get/set functionality.
> - Modify the conversion in the 2 locations from #57229-2 to fixup the converted code references.
>
> Is there something else to do?

I do not think so, it's basically just adding the same handling for 'get' method of 'progress.reflect.variable' and 'progress.reflect.property'.

> Since the 4GL syntax itself does not have any concept of a polymorphic return type, we have to add special processing to the conversion to handle those built-in runtime POLY features. Considering we may find other cases of this over time, we may want to "mark" references to POLY calls (at parse time) so that we can automatically process them downstream. Otherwise we have to detect each case downstream, which seems to be messy and more error prone.

I'll try something out and see if we can solve that on our end.

#8 - 10/18/2021 09:44 AM - Marian Edu

Greg Shah wrote:

> Considering we may find other cases of this over time, we may want to "mark" references to POLY calls (at parse time) so that we can automatically process them downstream. Otherwise we have to detect each case downstream, which seems to be messy and more error prone.

I see those are already marked as 'OO_METH_POLY' (oldtype keeps the previous OO_METH value). Hard-coding all cases works but feels somehow not right so thought maybe we can add a new annotation option (say 'poly') in LegacySignature and use that when appropriate. Problem is I can't figure out how/when those annotations are resolved during conversion, thought the place was in collect_names.rules where annotations seems to be 'translated' as attributes or something... adding a new rule for this 'poly' annotation doesn't change anything though :(

```
<rule>isNote("poly") and getNoteBoolean("poly")
  <action>
    nmap.putAttribute(methodRoot, "poly", "true")
  </action>
</rule>
```

Is that the way to go - extra option in LegacySignature, or you mean something else completely?

#9 - 10/18/2021 05:04 PM - Greg Shah

The LegacySignature is mostly used at runtime. For conversion support, it is only used for gap analysis marking. I think the core processing here needs to be done on the AST.

The "patch" that Constantin is referencing, is a "quick and dirty" hack. We could extend it to handle the get() poly return type and the set() poly parameter.

And perhaps we should just do that. In that case, you would add code to ClassDefinition.annotateMethodCall() to overwrite the get() return type in the same way as the code does today (node.setType(OO_METH_POLY);). node is the method call AST itself and the token type of that node is the return type of the method call. So when it is set to OO_METH_POLY, this tells the downstream code that it is one of these special POLY cases. To implement this, you would have to check if it was a Progress.Reflect.Property or Progress.Reflect.Variable and look for get as the method name.

The set() parameter is a little different since we have to mark the child node of the method call which represents the POLY parameter. We can check if it was a Progress.Reflect.Property or Progress.Reflect.Variable and look for set as the method name to find the condition. But then we will have to look at the last child of that node since there are 4 possible signatures:

```
void set(input value as POLY)
void set(input index as integer, input value as POLY)
void set(input instance as Progress.Lang.Object, input value as POLY)
void set(input instance as Progress.Lang.Object, input index as integer, input value as POLY)
```

The problem here is that the last child node can be any since of expression, simple or complex. We currently don't have an approach to handle that part in a standard way. The OO_METH_POLY is a standard way to mark the return type but our approach to set that is a hack. For other kind of 4GL

POLY cases, the parser itself (progress.g) has the knowledge needed to set the return type directly. There problem here is that in these OO cases, we are getting the details of the method definition from the skeleton classes. And these skeletons are written in the 4GL, as you know. But the 4GL has no way to define something as having a POLY type (return type or otherwise).

Although I'd like to see a standard way to encode this, I think we can defer that idea for now. We would have to invent something new here. One idea would be to provide a new 4GL keyword, like POLYMORPHIC which would be specified in the 4GL of the skeleton as if it was a data type name. Then the parser would have to be updated to recognize that and write the AST properly. Then the skeletons could be edited to have this 4GL extension where needed. As I say, don't worry about this now. The above approach for ClassDefinition.annotateMethodCall() will work for the get case. For the set case, I think we can easily match the condition and then we lookup the last child node and then set an annotation in it.

Constantin: Do we need any special handling for wrapping these poly cases? I see some code in method_defs.rules to rename invoke to invokeStandalone, but otherwise there is no significant processing. The set() case may be the first parameter that is being defined as a POLY so, perhaps we have some code to add for that.

The ExpressionConversionWorker.expressionType() needs to have code added to handle the new OO_METH_POLY cases, as noted in #5729-2. The idea is that the expressionType() for these specific cases should be BaseDataType. Other cases should error out.

#10 - 10/20/2021 06:33 AM - Marian Edu

Greg Shah wrote:

> The LegacySignature is mostly used at runtime. For conversion support, it is only used for gap analysis marking. I think the core processing here needs to be done on the AST.

I see now, the only source there is the skeleton content that is what the parser works with.

> The "patch" that Constantin is referencing, is a "quick and dirty" hack. We could extend it to handle the get() poly return type and the set() poly parameter.
>
> And perhaps we should just do that. In that case, you would add code to

ClassDefinition.annotateMethodCall() to overwrite the get() return type in the same way as the code does today (node.setType(OO_METH_POLY);).

Yes, I've added a private method to 'flag' (OO_METH_POLY) the cases when a POLY return type is used by a 4GL method.

> The set() parameter is a little different since we have to mark the child node of the method call which represents the POLY parameter.
> There problem here is that in these OO cases, we are getting the details of the method definition from the skeleton classes. And these skeletons are written in the 4GL, as you know. But the 4GL has no way to define something as having a POLY type (return type or otherwise).

Well, it does look like a hack but since the skeleton classes is something used internally by FWD for conversion only my thought was to use method overload in 4GL and actually have set methods for each data type (primitive and P.L.O).

That works fine so the skeleton class does convert but it has a bunch of set methods and I would rather avoid that and use BasseDataType instead. The problem is resolving the method from it's signature later on when matching method signature from javaMethodNames hash map. I can use all those methods from conversion and then delegate to a private implementation if that is the only option.

> The ExpressionConversionWorker.expressionType() needs to have code added to handle the new OO_METH_POLY cases, as noted in #5729-2. The idea is that the expressionType() for these specific cases should be BaseDataType. Other cases should error out.

Right, what I don't understand though is why checking again the method name there instead of the type which is OO_METH_POLY but maybe that was just a code used before even using the OO_METH_POLY annotation.

#11 - 10/20/2021 09:35 AM - Greg Shah

> Well, it does look like a hack but since the skeleton classes is something used internally by FWD for conversion only my thought was to use method overload in 4GL and actually have set methods for each data type (primitive and P.L.O).
>
> That works fine so the skeleton class does convert but it has a bunch of set methods and I would rather avoid that and use BasseDataType instead. The problem is resolving the method from it's signature later on when matching method signature from javaMethodNames hash map. I can use all those methods from conversion and then delegate to a private implementation if that is the only option.

Yes, I really want to avoid that "explosion" of fake methods. The BaseDataType is a better approach. I would rather add a special type name to the 4GL so that you could encode the POLY into the parameter data type and the parser would handle it nicely. Then we need to ensure that the downstream processing is OK with that type name as well.

Constantin: Where do you think it will hurt us in the code generation? I suspect we need some wrapping for this at a minimum.

In regard to the resolution of the method signatures, I want to minimize the impact there. By changes for #4350 are a major rewrite of that code and I have to stablize it and rebase to the latest level. Any changes in this area will complicate that work.

#12 - 10/21/2021 09:55 AM - Marian Edu

Greg Shah wrote:

> Yes, I really want to avoid that "explosion" of fake methods. The BaseDataType is a better approach. I would rather add a special type name to the 4GL so that you could encode the POLY into the parameter data type and the parser would handle it nicely. Then we need to ensure that the downstream processing is OK with that type name as well.

Adding a POLY data type (*poly*) inside the parser kinda works, we can use any name that isn't a valid 4GL data type there as this will only be used in skeleton classes anyway - this translates to BaseDataType and everything seems to work, up to a point.

> Constantin: Where do you think it will hurt us in the code generation? I suspect we need some wrapping for this at a minimum.

Yes, the issue in code generation is that when passing a primitive data type as input the generated code unnecessarily 'wraps' the value in an BaseDataType instance and since that is an abstract class it simply don't compile :(

No idea where to look for nor what needs to be done to avoid this. This is how it looks if a variable is being used:

```
silent(() -> oClass.ref().setPropertyValue(oReference, cPropName, new BaseDataType(cInputValue)));
```

and when using directly a string that is wrapped twice but that we've saw for any primitive values.

```
silent(() -> oClass.ref().setPropertyValue(oReference, cPropName, new BaseDataType(new BaseDataType("input_val
")));
```

> In regard to the resolution of the method signatures, I want to minimize the impact there. By changes for #4350 are a major rewrite of that code and I have to stablize it and rebase to the latest level. Any changes in this area will complicate that work.

The only change I made there is for 'fuzzy' resolution when matching parameters consider a match when the candidate type is BDT, right now it's a match if the caller type is a BDT (guess this happens when the parameter is a return of a poly function/method/attribute).

#13 - 10/22/2021 03:50 AM - Marian Edu

OK, while using a predefined 'ANY' poly data type in skeleton works we need to decide if this is the way to go forward or we leave setParameterValue from Class and set methods in Variable/Property alone for now. The handling for ParameterList.setParameter is already complicated as it is and there is always one poly parameter on 4th position, much more complex this time so I wouldn't pursue this path - handling based on the parameter type would be more appropriate imho.

#14 - 10/22/2021 04:33 AM - Constantin Asofiei

Marian, as you mentioned, we already handle a POLY parameter for setParameter and CALL:SET-PARAMETER - see dynamic_set_parameter usage (defined in common-progress.rules).

In the property setter case, there should be no wrapper needed, as the argument is always INPUT. For setParameter case, the argument can be OUTPUT or INPUT-OUTPUT, too, and that's why things get more complicated (by wrapping in a BaseDataTypeVariable, PropertyReference, etc). The Java code defining this property setter should have an java.lang.Object type for the last parameter (and not BaseDataType) - otherwise, you will not be able to pass extent values.

I suggest adding -Drules.tracing=true to your conversion command, as this will inject annotations like this in the AST:

```
      <annotation datatype="java.lang.String" key="peerid-byrule" value="convert/buffer_definitions.rules:1109
"/>
```

By looking at the .jast for the new BaseDataType code, you can backtrack which rule is responsible for adding the BaseDataType constructor wrapper, and analyze that code. I suspect the easiest way is to add exceptions for this property setter method call, to ensure the argument is never wrapped.

While on the argument, you can use this.indexPos == parent.numImmediateChildren to ensure you are on the 'last argument' of this setter call.

Greg: for the property getter, as with LegacyClass.invoke, we can set the return type to BaseDataType, but this will not solve the case when an extent is returned... the same issue is with DYNAMIC-FUNCTION return type.

#15 - 10/22/2021 05:59 AM - Greg Shah

> Greg: for the property getter, as with LegacyClass.invoke, we can set the return type to BaseDataType, but this will not solve the case when an extent is returned... the same issue is with DYNAMIC-FUNCTION return type.

We could add code to detect the extent using ECW.analyzeContext() for cases where we are inside a more complex expression. The idea would be to return "BaseDataType[]" and then call a differently named version of the runtime method which has the array return type.

One thing we would need to do is to handle the assignment cases. That would be a bit weird for ECW since it by nature requires us to look outside of the EXPRESSION node. The other approach is to add TRPL rules to do this in advance and leave behind some annotation at the EXPRESSION node.

Constantin: If you feel comfortable with this idea, you can go ahead with the change. I could be wrong, but Marian might appreciate it. :)

#16 - 10/22/2021 06:16 AM - Marian Edu

Greg Shah wrote:

> > Greg: for the property getter, as with LegacyClass.invoke, we can set the return type to BaseDataType, but this will not solve the case when an extent is returned... the same issue is with DYNAMIC-FUNCTION return type.

Property getter can't be extent, Get methods there have an overload for when variable/property is extent one must pass in the extent index so BaseDataType should work just fine. Same for Class.getPropertyValue methods.

> Constantin: If you feel comfortable with this idea, you can go ahead with the change. I could be wrong, but Marian might appreciate it. :)

Based on Constantin insights in previous message I'm almost done with it (I hope).

#17 - 10/22/2021 06:47 AM - Constantin Asofiei

Greg Shah wrote:

> Greg: for the property getter, as with LegacyClass.invoke, we can set the return type to BaseDataType, but this will not solve the case when an extent is returned... the same issue is with DYNAMIC-FUNCTION return type.

> We could add code to detect the extent using ECW.analyzeContext() for cases where we are inside a more complex expression. The idea would be to return "BaseDataType[]" and then call a differently named version of the runtime method which has the array return type.
>
> One thing we would need to do is to handle the assignment cases. That would be a bit weird for ECW since it by nature requires us to look outside of the EXPRESSION node. The other approach is to add TRPL rules to do this in advance and leave behind some annotation at the EXPRESSION node.
>
> Constantin: If you feel comfortable with this idea, you can go ahead with the change. I could be wrong, but Marian might appreciate it. :)

The problem with POLY is that we can't always determine from the surrounding expression or the caller, that this usage is extent POLY - for example, a RUN or another dynamic call, we don't know if the target's parameter is extent; so, which one do you use at conversion time? The return BDT or return BDT[] version?

In any case, this requires some more thought into it, as this would affect OO calls (i.e. if the target parameter is extent, the POLY argument can be forced as extent with the target's type, if we can disambiguate the target - in other cases the conversion will fallback to a dynamic OO call, and not direct call... IIRC we already do something like this for POLY cases).

An additional note is AFAIK these POLY cases (where we use it as an argument) can't be used for OUTPUT/INPUT-OUTPUT, only INPUT - this should simplify things a little.

#18 - 10/22/2021 07:29 AM - Greg Shah

> The problem with POLY is that we can't always determine from the surrounding expression or the caller, that this usage is extent POLY - for example, a RUN or another dynamic call, we don't know if the target's parameter is extent; so, which one do you use at conversion time? The return BDT or return BDT[] version?

That is true, but we have found that in the past we can calculate many cases this way so it is worth implementing to at least handle those cases cleanly.

Constantin: Please create a separate task for this extent problem (including all the different forms like DYNAMIC-FUNCTION). It seems like we can avoid it for this current task.

#19 - 10/26/2021 07:35 AM - Marian Edu

Looks like I've spoke too early, get methods in Variable/Property and getPropertyValue in Class can return an array if the property is extent albeit it is possible to get a specific entry in that array by using one get method that takes the index as input. Same for set methods, although one can set an entry using the index input approach the whole property can be directly set to an array. So 'poly' really must translate to 'Object' (anything, including array) and not just BaseDataType as I've thought :(

#20 - 10/27/2021 08:54 AM - Marian Edu

OK, what I've end-up doing is this:
- added 'POLY' data type in grammar ("_BDT_", use underscore to make sure it's not valid class name)
- use that in skeleton files when return or parameter type can be any data type (not possible otherwise in 4GL code so this is the only use case)
- use 'Object' as return or parameter type in Java classes, use 'BDT' type annotation in LegacyParameter
- if method return type is 'POLY' then the method is flagged as poly so nothing left to do in annotateMethodCall of ClassDefinition since the method is already seen as POLY
- changed literals.rules to avoid cast when method expects POLY (class is Object), when input is string literal just cast it to character if poly

It seems to work as-is now but I might just be missing something obviously here (why the complex handling for ParameterList.setParameter or Class.invoke methods).

Otherwise I've fixed an issue with properties that only had setter (the code generation was missing classname which was only set in getter rule). There seems to be an issue with accessing extent variables, for properties the getter/setter methods gets overloads that uses the index and also lengthOf respectively resize methods for each property. This is not the case for variables hence when accessing a static/instance variable the default substitute call is injected in AST only that the code generated is wrong as substitute node is included right before the variable name so the instance reference is left out of it:

```
// wrong
oPlo.ref()subscript(extInd).objectVar

// probably better
subscript(oPlo.ref().objectVar, extInd)
```

#21 - 10/27/2021 09:38 AM - Greg Shah

Do you have the code checked in to 4384m? If so, let us know the revision(s) so that we can do a code review.

#22 - 10/28/2021 04:18 AM - Marian Edu

Greg Shah wrote:

> Do you have the code checked in to 4384m? If so, let us know the revision(s) so that we can do a code review.

Pushed all changes in #rev12945. I will pack the skeleton changes as well if we stick to this approach. Using the return from a POLY method directly in a primitive function (min, max) doesn't work as-is, DynamicOps was designed to work with BaseDataType inputs while our 'POLY' really means everything (including arrays/extents). 4GL accepts that (is valid syntax since it doesn't know the type till runtime) and it throws error if the input is an

array: ** An array was specified in an expression, on the right-hand side of an assignment, or as a parameter when no array is appropriate or expected. (361)