Conversion Tools - Feature #3695

instrument 4GL code for automated call-graph hint generation

08/23/2018 07:53 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 Conversion Tools - Feature #3277: implement call-graph v3	**Closed**
Related to Conversion Tools - Feature #3696: implement Progress AST anti-parser	**WIP**

History

#1 - 08/23/2018 08:11 AM - Greg Shah

- Related to Feature #3277: implement call-graph v3 added

#2 - 08/23/2018 08:11 AM - Greg Shah

- Related to Feature #3696: implement Progress AST anti-parser added

#3 - 08/23/2018 08:19 AM - Greg Shah

The idea is described in #3277-208 through #3277-212.

We will want to make the performance impact negligible so that it is feasible to use in a wide range of environments. It also needs to be foolproof and easy to use. The entire point is to make it so easy to do this that any 4GL developer can be successful.

I can accept the limitation that the correctness of the result is dependent upon the quality and completeness of the scenarios run through the instrumented code.

Since that discussion, I have started work on #2110 and #3696. The result from those tasks will be a fully functional Progress AST transformation toolset. I would prefer to use this for instrumentation if possible. One problem I see here is that the currently planned approach for #3696 is to ignore include files.

Considering that Progress developers don't often work with patches/diffs, I think we need to plan the solution accordingly.

Constantin: Please provide your thoughts. Can we make it exceptionally easy for a customer to generate the correct call graph?

#4 - 08/23/2018 01:40 PM - Constantin Asofiei

Greg Shah wrote:

> I would prefer to use this for instrumentation if possible.

Yes, these will be needed. The other (more complex and ugly approach) would be to inject these somewhere at the parser level.

> One problem I see here is that the currently planned approach for #3696 is to ignore include files.

As long as the resulted 4GL code can be ran in 4GL, we should be OK - as this instrumented code will not be used in production.

> Considering that Progress developers don't often work with patches/diffs, I think we need to plan the solution accordingly.

Not sure what you mean here - are you referring that a 4GL dev would want to bring the instrumented code into their main/uninstrumented application?

> Constantin: Please provide your thoughts. Can we make it exceptionally easy for a customer to generate the correct call graph?

My first step would be to create 4GL tools to:

- load/save the hints. Loading would be done on first access of a program file, and save would be done when the application is closed - with all hints being saved within a temporary table.
- APIs to set/reset the hints for a call-site.
- APIs to create the linkage from a call-site to the target (i.e. add a hint to the SOURCE-PROCEDURE with the specified target of the RUN statement) - these would be added at the beginning of each entry point (external program, procedure, function, OO method).

I think this can be done in 3-4 days, including the manual instrumentation of the Hotel GUI code. Note that:

- we will not be able to instrument calls outside of 4GL, i.e. process launch, native calls, etc
- this instrumentation will work if the call-site is NOT part of a complex expression/statement. As the instrumented code would look like this:

```
// API call to save the hint
super().
// API call to reset the hint - required, as the target may be outside of 4GL (i.e. native call)
```

> any usage in a complex expression, like i = i + super() + dynamic-function(someFuncName). can't be easily instrumented with the hint name - because each call-site needs to be instrumented individually with its specific hint ID. I think same applies to chained OO calls, like new SomeClass():someMethod()..

Beside this, all call-sites targeting 4GL code should be instrumented (and saved into hints), regardless if they are dynamic or not - because, even with OO calls, depending on the runtime type of the OO var, a method call may target different implementations.

Once we have these APIs defined, I think ~2-3 weeks to create the TRPL rules and experiment/stabilize instrumentation with a large code set. Thinking about it, the TRPL rules can be included and tested via the post-parse-fixups phase, and at least with Hotel GUI we can test it directly in FWD. This should save some time, as we will not be dependent on the anti-parser.

#5 - 08/23/2018 02:49 PM - Greg Shah

> Considering that Progress developers don't often work with patches/diffs, I think we need to plan the solution accordingly.

> Not sure what you mean here - are you referring that a 4GL dev would want to bring the instrumented code into their main/uninstrumented application?

In the past, we have provided 4GL changes as .patch files created using diff. Many 4GL developers are unsure what to do with changes provided via this mechanism so it adds confusion.

Thus: I think it is easiest if we provide entire files that have been edited.

> this instrumentation will work if the call-site is NOT part of a complex expression/statement

Is this a limitation for the automatically generated instrumentation?

> can't be easily instrumented with the hint name - because each call-site needs to be instrumented individually with its specific hint ID

For the automatic generation of the instrumented code, can't we calculate the hint ID for each location?

> Thinking about it, the TRPL rules can be included and tested via the post-parse-fixups phase, and at least with Hotel GUI we can test it directly in FWD. This should save some time, as we will not be dependent on the anti-parser.

I'm trying to get the anti-parser done very soon, so I prefer using that approach if we can.

#6 - 08/23/2018 02:58 PM - Constantin Asofiei

Greg Shah wrote:

> Thus: I think it is easiest if we provide entire files that have been edited.

Yes, the entire changed file will need to be provided; but, I would think of this process more like having a separate installation for testing/computing the callgraph purposes, and this installation would use the entire anti-parsed code. Otherwise, it can make things messy if the instrumentation 4GL changes are integrated in the next processing. The idea is, this instrumentation would use as input the legacy application (uninstrumented) and have as output an instrumented version of the application, which will be used for testing/callgraph computing.

> this instrumentation will work if the call-site is NOT part of a complex expression/statement

> Is this a limitation for the automatically generated instrumentation?

Yes. I don't see a way how to instrument the hints for each call-site in a complex expression. If there is only one, then it can be done; the problem is when there are multiple call-sites. Even if we would assume each call-site would get 'consumed' in order (by each call), this does not apply if a call targets a native API - in this case, we can't consume the hint, as we are outside of 4GL code.

> can't be easily instrumented with the hint name - because each call-site needs to be instrumented individually with its specific hint ID

> For the automatic generation of the instrumented code, can't we calculate the hint ID for each location?

Yes, the hint ID will be automatically calculated and instrumented; but the problem (as mentioned above) is having more than one call-site in a complex expression.

#7 - 08/23/2018 03:20 PM - Constantin Asofiei

Constantin Asofiei wrote:

> ... the problem is when there are multiple call-sites. Even if we would assume each call-site would get 'consumed' in order (by each call), this does not apply if a call targets a native API - in this case, we can't consume the hint, as we are outside of 4GL code.

And more, if the call site execution depends on i.e. expression evaluation, then these call-sites might not be executed at all (think logical expressions).

#8 - 08/24/2018 10:00 AM - Greg Shah

OK, you make some good points about complex expressions. The following seem to be affected (when they are in a complex expression):

- any direct user-defined function call (my-func(parm1, parm2))
- any indirect user-defined function call (DYNAMIC-FUNCTION("my-func", parm1, parm2))
- super()
- non-chained method calls (some-method(parm1))
- chained method calls (some-method(parm1):other-method(parm2))

However, I think we can solve most of the issues. The super() and DYNAMIC-FUNCTION() cases are tricky. I have a proposal for super(), which I don't know if it is a complete solution. I think I the DYNAMIC-FUNCTION() case can be handled as well.

To solve the issue we create a "wrapper" for each separate part.

For example, calls to my-func(parm1, parm2) are replaced by a call to my-func-wrapper(parm1, parm2) and the wrapper is a local function that looks like this:

```
FUNCTION my-func-wrapper RETURNS <ret_type> (<parm1_def>, <parm2_def>):
   DEF VAR rc AS <ret_type>.
   // setup hint
   rc = my-func(parm1, parm2).
   // clear hint
   return rc.
END.
```

We already know the type information at parse time for all of the above cases except for **some** DYNAMIC-FUNCTION() usage. We have the DYNAMIC-FUNCTION() typing info that works for most cases that have string literal function names except where there are conflicts that have different signatures. We've been handling the dynamic expression cases at conversion time using context type analysis which has been sufficient for the rest so far. I can see some need for hints here in cases that are still not supported, but it is probably the minority of cases so it is OK.

The DYNAMIC-FUNCTION() cases where we know the typing (or have hints) can be solved the same way, as can the chained and non-chained method calls.

This approach retains all the semantics, evaluation order, error handling and even conditional evaluation for the complex expression.

Even for the super() case, I think that super() will only be called once from a function (generally). If that is the case, we can setup a hint for the super() call before the complex expression and that hint would have the function name in it. If a function was called by that name and there is a pending super(), perhaps we can detect that case and match with it?