Base Language - Feature #1970

improve ControlFlowOps method resolution by reflection

01/21/2013 03:35 AM - Constantin Asofiei

Status:	WIP	**Start date:**	
Priority:	Normal	**Due date:**	
Assignee:	Constantin Asofiei	**% Done:**	80%
Category:		**Estimated time:**	10.00 hours
Target version:			
billable:	No	**version:**	
vendor_id:	GCD		

Description

Related issues:

Related to Base Language - Feature #1920: implement persistent procedures	**Closed**
Related to Base Language - Support #4032: block/transaction processing contex...	**New**

History

#1 - 01/21/2013 03:36 AM - Constantin Asofiei

Cache the Method objects obtained by Class.getMethod (and other Class APIs) by class/method name/signature, to improve method resolution. This applies for the c'tor lookup too.

#2 - 04/12/2013 04:39 AM - Constantin Asofiei

- Estimated time set to 10.00

#3 - 03/09/2020 04:55 PM - Greg Shah

I've been thinking about how we could cache method instances in ControlFlowOps recently. I started to document my idea and then I found this task, so I'm leaving my thoughts here.

I want to go much further than just caching method objects. I want to avoid all of the dynamic processing done during a RUN statement/function/method invocation.

The idea is that most call sites invoke the same method each time (even if it is from a different instance of the business logic class). Of course, it is possible for the invoked code to change the target from call to call, but it is not as common.

It follows that caching this method in a manner that links the method to the call site might have significant performance benefits. My idea is that we would emit a static object instance for each call site (private static final Object _CALL_SITE_TOKEN_xyz = new Object();). This can be passed to CFS from a specific call site (and only ever from that site) so it can be used as the key in a caching strategy.

If the same target is used by a given call site, then it will result in the same called code being invoked unless the following happens:

- PROPATH is modified
- the super procedure stack is changed

These "events" would cause cache invalidation.

I am also thinking that in addition to eliminating the reflection lookup process for 2nd and subsequent calls, we can potentially cache other related data with that same key (if this is helpful).

I'd like to see if we can also avoid all the parameter validation that happens on each call. It seems to me that the validation is normally related to the data types and modes passed from the call site. But these don't vary at all. Both the types and modes are fixed for a given call site. Why not handle the validation on the first invocation and save the result in the cache? In the common case, the result was success and we bypass all that validation work. In the failing case, we raise the correct error as needed.

Finally, we will want to do some kind of simple check to at least confirm that the target has not changed from a previous call. If the target changes, then we would want to remove the cache entry (or replace it) but we definitely wouldn't use the cached result. I suspect this is the minority of cases.

#4 - 03/09/2020 04:56 PM - Greg Shah

- Related to Support #4032: block/transaction processing context-local and other performance improvements added

#5 - 09/02/2020 03:13 PM - Greg Shah

I've been reviewing the CFS code and debugging through it with testcases to understand the implications of this idea. Based on this analysis, there is a substantial amount of processing that can be potentially bypassed with either a "site specific" approach or with much more aggressive caching in general. By aggressive, I mean caching everything needed to invoke the target immediately, without any usage of a Resolver, without any additional argument/parameter mode validation, without any lookups in SourceNameMapper and so forth.

I remain convinced that the call-site specific approach is a very workable idea. I think it allows us to detect invalidation events that affect that call site such that we can know at the time the site is executed whether the cached value is valid or not. I do have some open questions about whether there are any kinds of arguments which still need runtime validation. Static call sites do not vary their argument types, so it seems to me that only the POLY cases and possibly fixd/indeterminate extent cases may need runtime checking.

With this in mind, I tried a simple test to estimate the value of the idea.

unchanging_invocation_targets.p

```
def var start as int64.
def var i as int.
def var num-ep as int.
def var num-ip as int.

start = etime.
do i = 1 to 10000:
   run increment-arg.p (input-output num-ep).
   run ip.
end.

message "Completed in " + string(etime - start) + " milliseconds.".

procedure ip:
   num-ip = num-ip + 1.
end.
```

increment-arg.p

```
def input-output parameter num as int.

num = num + 1.
```

The idea is that these cases really have fixed call targets, so site-specific caching would always work.

By default, the conversion (branch 3821c) is as follows:

```
package com.goldencode.hotel;

import com.goldencode.p2j.util.*;
import com.goldencode.p2j.ui.*;

import static com.goldencode.p2j.util.BlockManager.*;
import static com.goldencode.p2j.util.InternalEntry.Type;
import static com.goldencode.p2j.util.MathOps.*;
import static com.goldencode.p2j.util.TextOps.*;
import static com.goldencode.p2j.util.character.*;
import static com.goldencode.p2j.ui.LogicalTerminal.*;

/**
 * Business logic (converted to Java from the 4GL source code
 * in unchanging_invocation_targets.p).
 */
public class UnchangingInvocationTargets
{
   @LegacySignature(type = Type.VARIABLE, name = "num-ip")
   integer numIp = UndoableFactory.integer();

   /**
```

```
    * External procedure (converted to Java from the 4GL source code
    * in unchanging_invocation_targets.p).
    */
   @LegacySignature(type = Type.MAIN, name = "unchanging_invocation_targets.p")
   public void execute()
   {
      int64 start = UndoableFactory.int64();
      integer i = UndoableFactory.integer();
      integer numEp = UndoableFactory.integer();

      externalProcedure(UnchangingInvocationTargets.this, new Block((Body) () ->
      {
         start.assign(date.elapsed());

         ToClause toClause0 = new ToClause(i, 1, 10000);

         doTo("loopLabel0", toClause0, new Block((Body) () ->
         {
            ControlFlowOps.invokeWithMode("increment-arg.p", "U", numEp);
            ControlFlowOps.invoke("ip");
         }));

         message(concat("Completed in ", valueOf(minus(date.elapsed(), start)), new character(" milliseconds."
)));
      }));
   }

   @LegacySignature(type = Type.PROCEDURE, name = "ip")
   public void ip()
   {
      internalProcedure(new Block((Body) () ->
      {
         numIp.assign(plus(numIp, 1));
      }));
   }
}
```

I then made the following manual edits:

```
package com.goldencode.hotel;

import com.goldencode.p2j.util.*;
import com.goldencode.p2j.ui.*;

import static com.goldencode.p2j.util.BlockManager.*;
import static com.goldencode.p2j.util.InternalEntry.Type;
import static com.goldencode.p2j.util.MathOps.*;
import static com.goldencode.p2j.util.TextOps.*;
import static com.goldencode.p2j.util.character.*;
import static com.goldencode.p2j.ui.LogicalTerminal.*;

/**
 * Business logic (converted to Java from the 4GL source code
 * in unchanging_invocation_targets.p).
 */
public class UnchangingInvocationTargets
{
   @LegacySignature(type = Type.VARIABLE, name = "num-ip")
   integer numIp = UndoableFactory.integer();

   long cumulativeEp = 0;
   long cumulativeIp = 0;

   /**
    * External procedure (converted to Java from the 4GL source code
    * in unchanging_invocation_targets.p).
    */
   @LegacySignature(type = Type.MAIN, name = "unchanging_invocation_targets.p")
   public void execute()
   {
      int64 start = UndoableFactory.int64();
      integer i = UndoableFactory.integer();
```

```
      integer numEp = UndoableFactory.integer();

      externalProcedure(UnchangingInvocationTargets.this, new Block((Body) () ->
      {
         start.assign(date.elapsed());

         ToClause toClause0 = new ToClause(i, 1, 10000);

         doTo("loopLabel0", toClause0, new Block((Body) () ->
         {
            long startEp = System.currentTimeMillis();

            /*
            ControlFlowOps.invokeWithMode("increment-arg.p", "U", numEp);
            */
            IncrementArg ia = new IncrementArg();
            ia.execute(numEp);

            cumulativeEp += System.currentTimeMillis() - startEp;

            long startIp = System.currentTimeMillis();

            /*
            ControlFlowOps.invoke("ip");
            /*
            ip();

            cumulativeIp += System.currentTimeMillis() - startIp;
         }));

         String details = String.format(" (ext ms %d, int ms %d)", cumulativeEp, cumulativeIp);

         /*
         message(concat("Completed in ", valueOf(minus(date.elapsed(), start)), new character(" milliseconds."
)));
         */
         message(concat("Completed in ", valueOf(minus(date.elapsed(), start)), new character(" milliseconds"
+ details + ".")));
      }));
   }

   @LegacySignature(type = Type.PROCEDURE, name = "ip")
   public void ip()
   {
      internalProcedure(new Block((Body) () ->
      {
         numIp.assign(plus(numIp, 1));
      }));
   }
}
```

By switching out the CFS usage for the direct calls, we can see the cumulative timimg of those parts AND the specific difference that this kind of caching could have on the CFS usage. In fairness, this is the best case scenario. But it also represents a common case to which we should be able to get pretty close.

I see some interesting results. I did multiple runs, which allows the JVM native compilation to warm up as well as any caching to get properly preloaded.

Run #	CFS Total (ms)	Direct Total (ms)	CFS External Cumulative (ms)	Direct External Cumulative (ms)	CFS Internal Cumulative (ms)	Direct Internal Cumulative (ms)
1	823	886	514	271	242	**551**
2	600	752	367	182	195	**544**
3	204	391	132	59	56	**324**
4	118	302	80	31	27	**266**
5	109	320	70	49	30	**263**
6	154	305	117	45	34	**254**

The direct external procedure call is significantly fast than CFS but strangely, the direct internal procedure call is slower in all cases! This makes very little sense, since the CFS code is doing non-trivial processing AND THEN IS ALSO invoking the ip() method on the Method instance via reflection. I don't see how this can possibly be faster than a direct call (which uses a hard coded invokevirtual bytecode in the class file).

I suspect that we must be doing something expensive inside the BlockManager.internalProcedure() when we find the direct call is occurring. Perhaps we are doing some kind lookup of the this-procedure handle when it is not set?

Constantin: Do you have any ideas on this? I'd like to know for 2 reasons. First, it will tell us more about what must be cached to get this working. Second, it will give us a proper estimate of the benefit of this approach.

#6 - 09/02/2020 03:51 PM - Constantin Asofiei

Greg Shah wrote:

> I suspect that we must be doing something expensive inside the BlockManager.internalProcedure() when we find the direct call is occurring. Perhaps we are doing some kind lookup of the this-procedure handle when it is not set?
> Constantin: Do you have any ideas on this? I'd like to know for 2 reasons. First, it will tell us more about what must be cached to get this working. Second, it will give us a proper estimate of the benefit of this approach.

See BlockManager.checkJavaCall - this code is executed only if we are in a Java-direct call, like direct function execution, which can avoid ControlFlowOps. I suspect the resolveClosestMethod and findRootEnclosingInstance are the most expensive ones.

I don't know why internalProcedure doesn't have the this reference and legacy name emitted as arguments (this would have avoided the findRootEnclosingInstance and resolveClosestMethod calls).

#7 - 09/03/2020 12:02 PM - Greg Shah

> don't know why internalProcedure doesn't have the this reference and legacy name emitted as arguments (this would have avoided the findRootEnclosingInstance and resolveClosestMethod calls).

Yep, that was it.

Revised results:

Run #	CFS Total (ms)	Direct Total (ms)	CFS External Cumulative (ms)	Direct External Cumulative (ms)	CFS Internal Cumulative (ms)	Direct Internal Cumulative (ms)
1	868	538	528	303	265	182
2	443	354	278	197	136	121
3	311	246	198	128	88	64
4	157	69	106	45	38	19
5	103	67	74	41	25	22

6	140	70	99	41	38	21

Overall, this is a 21% to 57% performance improvement. This is the "headroom" for improvement in the CFS layer. Considering how commonly used this is, I think it is well worth the effort.

#8 - 09/03/2020 12:04 PM - Greg Shah

By the way, it seems to me that in the case where the target name is unique across the project, that we could hard code the invocation using the "direct" method. I'm trying to remember why we don't do that. Do you recall?

#9 - 09/03/2020 12:12 PM - Constantin Asofiei

Greg Shah wrote:

> By the way, it seems to me that in the case where the target name is unique across the project, that we could hard code the invocation using the "direct" method. I'm trying to remember why we don't do that. Do you recall?

For external programs, the initialization may fail (think missing shared variables, problems with frame definitions, etc). We can solve this with a hint, if we know that the external program initialization may never fail and the program is always in the propath (unique name doesn't mean the program can't be omitted from the propath).
For internal procedures, we can't guarantee the target, because of the super-procedure complications. But it may work if the internal procedure name is unique across the entire source code, and the RUN statement is in the defining external program. But we may have the same problems with frame definitions as with external programs.

The idea with the shared vars/frame failures is that FWD does this initialization before we execute our 'top-level block'. So, a IncrementArg ia = new IncrementArg(); will fail with no way to catch that error and process it. See ControlFlowOps.ExternalProgramResolver.

#10 - 09/03/2020 01:52 PM - Constantin Asofiei

Some issues I see:

1. RUN ON SERVER shouldn't be touched by this optimization - the resolution is done on the remote appserver.
2. RUN .. hard-coded-external-program-name and RUN hard-coded-internal-procedure can't be treated as if the target is always an external program or internal procedure. We need to treat them as a RUN VALUE(...) IN <handle>.
3. RUN VALUE(<value>) doesn't know what the target will be - an external program or internal procedure. If the <value> is not the same as the cached one, we can't use the cache.
4. RUN ... IN <handle>. This is the complex part. The target can be resolved from multiple places:

- the <handle> (which defaults to THIS-PROCEDURE), if IN <handle> is missing.
- super-procedures for <handle>
- session's super-procedures

Also::

- a PROCEDURE or FUNCTION can be defined IN SUPER with no actual body in this current external program.
- a FUNCTION can be defined as FUNCTION ... [MAP TO <other-name>] IN <handle>. In this case, there is a recursive resolution of the actual target.
- DLL/native targets should not be cached (at least in the first phase, as we don't have a Java method to call).

So, the cache should be:

- invalidated if PROPATH has changed
- invalidated if session's or any external program's super-procedures have been changed.
- the cached method should be recomputed if the actual external program handle for RUN ... IN, DYNAMIC-FUNCTION ... IN is not the same as the cached one
- the cached method should be recomputed if the target for the RUN statement is not the same as the last cached one.
- for FUNCTION ... IN <handle> definitions, considering that the target resolution may 'bounce' through multiple external programs until the actual implementation is found, we can cache:
  - the entire chain of handle vars (and not external programs!) used to reach it
  - the entire chain of external programs used to reach it.
    If the chains are not the same external programs, then we can't use the cache (at least is easy to decide early on).
    (Side note: I'm wrapping my brain but I can't figure out if is possible in 4GL for an original FUNCTION ... IN <handle> to 'bounce' through a FUNCTION ... IN SUPER definition - or if this works in FWD or not. In any case, we invalidate the cache if super-procedures have changed.).

About the arguments:

- I'm not sure if POLY arguments need to be treated specially - we should cache the 'call site signature', and if this is not the same as the cached one, then do not use the cached method.
- buffer/temp-table/dataset - I don't recall how these are validated at the call.
- even if the argument is not POLY, consider that 4GL does an automatic conversion from i.e. character to integer. So, the cache should keep a 'convertor' function for each argument, which has an incompatible datatype than the target (and apply this before the call, and raise an ERROR if conversion fails). See newpar usage in ControlFlowOps.InternalEntryCaller.valid.
- for output/input-output parameters, in some cases we have to create an additional 'back converter', see line 8660:

```
FieldAssigner.update(((BaseDataType) param[i]),
                     ((BaseDataType) newPars[i]),
                     bwCtor);
```

  ControlFlowOps.InternalEntryCaller.valid should be reviewed carefully, and even if the target has not changed for a call-site, perform all the additional conversion/registration for the arguments.
- same for extent arguments - they may need special validation, but considering that we can cache the call site signature, any changes between calls will trigger a recalculation of the target.

#11 - 09/03/2020 03:04 PM - Constantin Asofiei

An addition note: a RUN ... IN can target a remote procedure/port-handle - see this code in ControlFlowOps.invokeImpl:6286:

```
         WrappedResource res = h.getResource();

         // get the server from the handle
         if (ProcedureManager.isProxy(h))
         {
            server = new handle(((ProxyProcedureWrapper) res).getServer());
         }
         else if (res instanceof PortTypeWrapper)
         {
            server = new handle(((PortTypeWrapper) res).getServer());
         }
         else if (res instanceof DeferredProgramWrapper)
         {
            deferred = (DeferredProgram) ((DeferredProgramWrapper) res).get();
            server = deferred.getServer() == null ? null : new handle(deferred.getServer());
         }
```

In all these cases, the target is remote, so we can't cache.

#12 - 09/10/2020 10:30 AM - Greg Shah

I'm planning to separate RUN SUPER from SUPER() (they both convert as runSuper() today):

```
function bogus-func returns int ():
   /* compile error */
   /* Cannot specify RUN SUPER outside of internal procedure. (6435) */
   run super.
   return 0.
end.

procedure bogus-proc:
   /* compile error */
   /* Cannot invoke a SUPER user-defined function outside of a user-defined function definition. (6446) */
   super().
end.
```

The actual implementation is different inside anyway so there is no implementation advantage that I can see. Also, it will allow the elimination of the call to TransactionManager.nearestExecutingBlock() ro determine if this is a function or procedure.

Do you see any flaw in my logic?

#13 - 09/10/2020 10:32 AM - Constantin Asofiei

There is already a runSuper(Class<?>,...) which is emitted only for functions. I don't recall if runSuper(String iename) is emitted for functions - did you find this case?

I have no problem separating the calls in different Java methods.

#14 - 11/09/2020 03:52 PM - Greg Shah

Can InvokeConfig be used as a full replacement for all forms of CFS.invoke*() usage?

The more I look at CFS, the more I don't want to rework all of the invoke*() methods. I think the result would be much cleaner if we converted to a chained approach similar to how we use InvokeConfig today. I would want it to include the invocation itself and it would be expanded to include the caching support.

Constantin: What do you think?

#15 - 11/09/2020 04:02 PM - Constantin Asofiei

Greg Shah wrote:

> Can InvokeConfig be used as a full replacement for all forms of CFS.invoke*() usage?

Yes, InvokeConfig is a builder, and was added as an early step to refactor ControlFlowOps conversion emitted for RUN, etc.

There is a ControlFlowOps.invoke(InvokeConfig) which takes the instance and executes the call. You may want to add a InvokeConfig.execute() and emit it something line:

```
new InvokeConfig(target).setPersistent(true).setProcedureHandle(phandle).execute()
```

for a RUN target PERSISTENT SET phandle.

#16 - 11/09/2020 04:05 PM - Greg Shah

Is there anything significant that is missing from InvokeConfig support today?

All forms of function are supported too?

#17 - 11/09/2020 04:10 PM - Constantin Asofiei

Greg Shah wrote:

> Is there anything significant that is missing from InvokeConfig support today?
>
> All forms of function are supported too?

RUN SUPER and SUPER() are not supported at this time by ControlFlowOps.invoke(InvokeConfig). Otherwise, all clauses for RUN, DYNAMIC-FUNCTION or standalone function calls are supported (but were not extensively tested).

#18 - 11/11/2020 03:37 PM - Greg Shah

What is the purpose of the and !upPath("KW_ON/KW_PERSIST") exclusion in convert/control_flow.rules line 935 which has type == prog.kw_run and !upPath("KW_ON/KW_PERSIST")?

#19 - 11/11/2020 03:43 PM - Greg Shah

Nevermind, I guess that it is meant to exclude the persistent trigger form of the ON statement.

#20 - 08/01/2022 07:38 AM - Greg Shah

In regard to my idea to use a call-site specific "token" (e.g. private static final Object _CALL_SITE_TOKEN_xyz = new Object();) in the converted code, it seems to me that we could make this an InvokeConfig instance (e.g. private static final InvokeConfig _CALL_SITE_xyz = new InvokeConfig("some_hard_coded_target_name").setPersistent(true);). Any truly hard coded configuration like a string literal target name or setPersistent(true) could also be set once. Instead of looking up something cached, we could cache it in the call-site's token itself.

The tricky part is that at the call site, we would have some configuration/data that is dynamic or otherwise needs to be per-instance like the handle variable for a persistent procedure or parameters which can be passed in. We would need to be able to handle these parts separate from the cached data which would be shared for all instances (across sessions if we use static).

#21 - 09/01/2022 11:01 AM - Constantin Asofiei

Greg Shah wrote:

> The tricky part is that at the call site, we would have some configuration/data that is dynamic or otherwise needs to be per-instance like the handle variable for a persistent procedure or parameters which can be passed in. We would need to be able to handle these parts separate from the cached data which would be shared for all instances (across sessions if we use static).

This static InvokeConfig I would consider it just a template, with non-dynamic state (like persistent, argument modes, the target if is a string constant, and anything else). When actual invoke takes place, I would emit a new InvokeConfig(_CALL_SITE_TOKEN_).setInHandle(...).setArguments(...).execute() which will copy the template state into the instance, and also check if it can used the cached target (for direct invocation) for this call-site.

#22 - 09/02/2022 04:20 AM - Greg Shah

This makes sense.

#23 - 09/26/2022 01:34 PM - Constantin Asofiei

The conversion is in testing and for this test:

```
def var h as handle.
def var ch as char.

function func0 returns int(input i1 as int) in h.

run external-program.p.
run value(ch).
run value(ch) in this-procedure (1, 2, 3, 4).
dynamic-function("func0" in this-procedure, 1,2,3,4).
func0(1).
```

looks like this:

```
   private static final InvokeConfig RUN_CALL_SITE_1 = new InvokeConfig().setTarget("external-program.p");

   private static final InvokeConfig RUN_CALL_SITE_2 = new InvokeConfig();

   private static final InvokeConfig RUN_CALL_SITE_3 = new InvokeConfig().setModes("IIII");

   private static final InvokeConfig DYN_FUNC_CALL_SITE_1 = new InvokeConfig().dynamicFunction().setModes("III
I").setTarget("func0");

   private static final InvokeConfig FUNC_CALL_SITE_1 = new InvokeConfig().function().setTarget("func0").setMo
des("I");

   @LegacySignature(type = Type.VARIABLE, name = "h")
   handle h = UndoableFactory.handle();

   @LegacySignature(type = Type.VARIABLE, name = "ch")
   character ch = UndoableFactory.character();

   @LegacySignature(type = Type.MAIN, name = "ex.p")
   public void execute()
   {
      externalProcedure(Ex.this, new Block((Body) () ->
      {
         ProcedureManager.registerFunctionHandle("func0", h);
         new InvokeConfig(RUN_CALL_SITE_1).run();
         new InvokeConfig(RUN_CALL_SITE_2).setTarget((ch).toStringMessage()).run();
         new InvokeConfig(RUN_CALL_SITE_3).setTarget((ch).toStringMessage()).setInHandle(thisProcedure()).setA
rguments(new integer(1), new integer(2), new integer(3), new integer(4)).run();
         new InvokeConfig(DYN_FUNC_CALL_SITE_1).setInHandle(thisProcedure()).setArguments(1, 2, 3, 4).execute(
);
         new InvokeConfig(FUNC_CALL_SITE_1).setArguments(1).execute();
      }));
   }
```

The runtime for RUN internal-proc is almost done, need to add RUN external procedure, ASYNC/AS-THREAD and FUNCTION/DYNAMIC-FUNCTION cases. Dynamic OO will not be added at this time.

#24 - 09/26/2022 02:10 PM - Greg Shah

It looks good!

#25 - 09/27/2022 02:57 PM - Constantin Asofiei

Unfortunately this approach of putting the static parts at the CALL_SITE makes debugging very hard - either way, I didn't find any usage for the callsite's state at runtime, it gets copied to the actual InvokeConfig anyway... maybe the previous idea to make the CALL_SITE a plain 'Object' instance is better.

#26 - 09/27/2022 04:45 PM - Greg Shah

> Unfortunately this approach of putting the static parts at the CALL_SITE makes debugging very hard

The resulting converted code reads better, in my opinion. Is the issue with debugging that it is hard to see where the initial values are set?

#27 - 09/27/2022 04:59 PM - Greg Shah

I'd also go with an invocation syntax more like this:

```
RUN_CALL_SITE_1.clone().run();
RUN_CALL_SITE_2.clone().setTarget(ch).run();  // I got rid of the toStringMessage() unwrapping here
```

#28 - 09/28/2022 08:56 AM - Constantin Asofiei

Greg, I'm leaving caching support only for internal procedures or function calls. Others, like ASYNC, ON SERVER, AS-THREAD, SINGLETON/SINGLE-RUN, do not make sense at this time. I think external programs calls might be able to be cached, but I want to stabilize current version first.

I'll also 'flatten' the stacktrace on the ControlFlowOps.invoke(InvokeConfig), to directly call the (almost) final API, without bouncing through 4 or 5 other Java method calls.

In regard to your clone().run() comment, I'm working on this.

#29 - 09/29/2022 06:51 AM - Constantin Asofiei

ControlFlowOps is being used in FWD internal classes, for example invoking a callback procedure for SAX parsing or DATASET:FILL. These callsites can be cached using the resource, and invalidation must be done if the callback procedure changes (remove the cache for this resource).

I'll work on this after I finish the external program case. There is also the RUN SUPER or SUPER() case which is not supported in conversion yet...

#30 - 09/29/2022 08:25 AM - Greg Shah

> There is also the RUN SUPER or SUPER case which is not supported in conversion yet...

Agreed. These seem important too, especially for ADM/ADM2.

#31 - 09/29/2022 01:25 PM - Constantin Asofiei

For external procedures, the only thing I can cache is the target Java classname - to avoid PROPATH resolution on each and every invocation. Everything else, will go through the same paths, but explicitly targeting an external program (bypassing internal procedure checks, etc).

For RUN SUPER and SUPER - FWD was emitting the target legacy name, too - this is not needed, the runtime knows what converted internal entry it runs on top of the stack. I've removed this.

About the internal usage of ControlFlowOps - I've refactored the calls which made sense to use the cache support; the CALL_SITE is built based on the callback type, and the cache is being kept on the resource (and destroyed once the resource gets deleted, same as with external programs).

What's left:

- setArguments - I'm removing this and passing the arguments directly to the 'execute', 'run', 'runSuper'.
- flattening the ControlFlowOps API calls from the ControlFlowOps.invoke(InvokeConfig) method.

Otherwise, the runtime I've added until know looks good, but I need to reconvert everything again and re-test.

#32 - 09/29/2022 01:55 PM - Greg Shah

> For external procedures, the only thing I can cache is the target Java classname - to avoid PROPATH resolution on each and every invocation. Everything else, will go through the same paths, but explicitly targeting an external program (bypassing internal procedure checks, etc).

Can't we also avoid re-checking the data types of any arguments? It seems that the first time any call site is executed, we can validate the types and then never need to do that again. At least most (if not all) types seem like they can't possibly change at a given call site. I'm trying to think if there is a way that an OO parameter or other data type can be different from call to call. Maybe if a ternary is used, it is possible. Is there some other scenario? DYNAMIC-FUNCTION() will be wrapped so that is probably safe.

> For RUN SUPER and SUPER - FWD was emitting the target legacy name, too - this is not needed, the runtime knows what converted internal entry it runs on top of the stack. I've removed this.

Nice!

#33 - 09/30/2022 01:18 PM - Constantin Asofiei

- % Done changed from 0 to 80

- File ca_upd_20220930a_3821c_14261.zip added

- Status changed from New to WIP

- Assignee set to Constantin Asofiei

Greg, please review these changes. The runtime for the caching is this:

- use the caching mechanism for event procedure callbacks (for CALL, BUFFER, DATASET, QUERY, SOCKET, SOCKET-SERVER, SAX-READER).
- RUN SUPER, SUPER function, RUN statement, DYNAMIC-FUNCTION, FUNCTION ... IN SUPER/handle invocations are being converted to call-site and InvokeConfig support.
- SINGLE-RUN, SINGLETON, ASYNC, AS-THREAD, OO calls (like dynamic method call or NEW), FUNCTION ... IN SUPER/handle invocations, library calls, are not being cached.
- the cache is used for:
  - external programs (via RUN or RUN ... ON SESSION): the Java classname is cached, to avoid PROPATH resolution; regarding your question about the argument validation - the secondary cache related to this was already added some long time ago.
  - internal procedures and functions, if the target, IN handle, argument types, etc have not changed.
- the cache is invalidated on:
  - propath change
  - super-procedure change (either SESSION or external program). I've been thinking for a way to not be this aggressive and not clear the entire cache when any kind of super-procedure gets changed, but I couldn't find a good way for this.
- when the resource/external program gets deleted, its associated cache gets cleaned up

I have not yet 'flattened the stacktrace',

#34 - 10/02/2022 11:19 AM - Constantin Asofiei

Latest update is attached - for external programs, a separate call-site cache is being kept, as these are dependent on PROPATH and not SUPER-PROCEDURES.

#35 - 10/02/2022 12:11 PM - Greg Shah

ca_upd_20220930a_3821c_14261.zip is the latest?

#36 - 10/02/2022 12:17 PM - Constantin Asofiei

- File ca_upd_20221002a_3821c_14261.zip added

Greg Shah wrote:

> ca_upd_20220930a_3821c_14261.zip is the latest?

No, see attached.

#37 - 10/05/2022 04:39 AM - Constantin Asofiei

ca_upd_20221002a_3821c_14261.zip was committed to 3821c/14268

#38 - 10/10/2022 07:51 AM - Constantin Asofiei

3821c/14283 fixed problems in 3821c/14268:

- for typed FUNCTION calls, convert the returned value to the expected type.
- fixed caching issues when the call does not complete due to errors or for recursive calls.
- InvokeConfig.runSuper() and InvokeConfig.execute() calls always mark the configuration as FUNCTION.

Files

ca_upd_20220930a_3821c_14261.zip	572 KB	09/30/2022	Constantin Asofiei
ca_upd_20221002a_3821c_14261.zip	573 KB	10/02/2022	Constantin Asofiei