

METAOBJECT PROTOCOL MEETS INVOKEDYNAMIC

Attila Szegedi
Chief Architect, Adeptra Inc.

STILL CHASING THE SAME GOAL

- Interoperability between language runtimes in single JVM
- Code written in one language should be able to
 - invoke code written in another language
 - manipulate native objects of another language

NOW COMES
IN A NEW
FLAVOR

Refreshing invokedynamic aroma



CODE IS WORTH THOUSAND WORDS

obj.helloText



```
methodVisitor.visitMethodInsn(  
    INVOKEDYNAMIC,  
    "java/dyn/InvokeDynamic",  
    "dyn:getProp:helloText",  
    "(Ljava/lang/Object;)Ljava/lang/Object;"  
)
```

- “dyn” is the namespace prefix for standard operations
- “getProp” declares a property getter
- “helloText” is a fixed property name
- signature is very generic

WHY IS INVOKEDYNAMIC BETTER

- Separates the work into two phases: link-time and invocation-time
- The more you can push into link-time, the better
 - in the previous example, property id is fixed, so we can resolve it at link-time
- Additionally, invocation-time work can end up inlined

WHAT ARE YOU DOING ON MY PROPERTY?

dyn:getProp:\$name	(Object)⇒Object	property getter fixed name
dyn:getProp	(Object, Object)⇒Object	property getter variable name
dyn:setProp:\$name	(Object, Object)⇒void or (Object, Object)⇒Object for chaining/immutables	property setter fixed name
dyn:setProp	(Object, Object, Object)⇒void or (Object, Object, Object)⇒Object for chaining/immutables	property setter variable name

HANDLING CONTAINERS

<code>dyn:getLength</code>	<code>(Object)⇒int</code>	size of the container
<code>dyn:getElement</code>	<code>(Object, Object)⇒Object</code> <code>(Object, Object, Object)⇒void</code>	element getter
<code>dyn:setElement</code>	<code>(Object, Object, Object)⇒Object</code> or <code>(Object, Object, Object)⇒void</code> for chaining/immutables	element setter

- Still outstanding: iteration, keyset/values for maps

CALLING

dyn:callPropWithThis:\$name

(Object[,any])⇒any

call method of a
fixed name on an
object

dyn:callPropWithThis

(Object, Object[,any])⇒any

call method of a
variable name on
an object

dyn:call

(Object[,any])⇒any

call the object

SOME WORKSHOP DISCUSSION TOPICS

- Element/property not distinct in every language.
- Can “method missing”, prototype chaining all be considered language implementation detail? (Hopefully, yes.)
- Supporting immutable objects in mutator operations.
- Iteration, map keysets
- Supporting generic (not receiver based) dispatch

LINKING

```
import java.dyn.*;
import org.dynalang.dynalink.*;

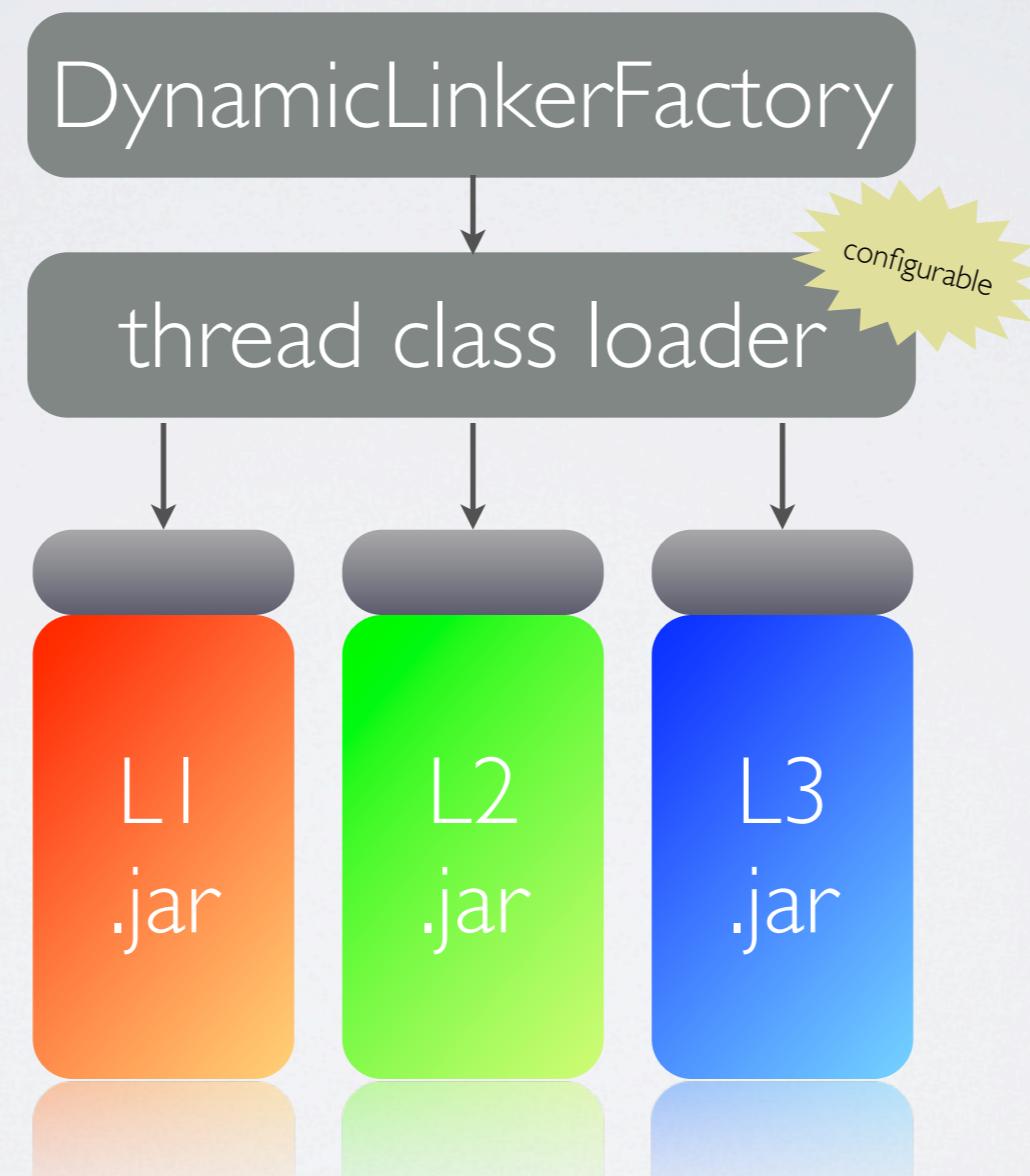
class MyLanguageRuntime {
    private static final DynamicLinker linker = new DynamicLinkerFactory().createLinker();

    public static CallSite bootstrap(Class caller, String name, MethodType type) {
        RelinkableCallSite callSite = new MonomorphicCallSite(caller, name, type);
        linker.link(callSite);
        return callSite;
    }
}

class SomeClassInMyLanguage {
    static {
        java.dyn.Linkage.registerBootstrapMethod(MyLanguageRuntime.class, "bootstrap");
    }
    ...
}
```

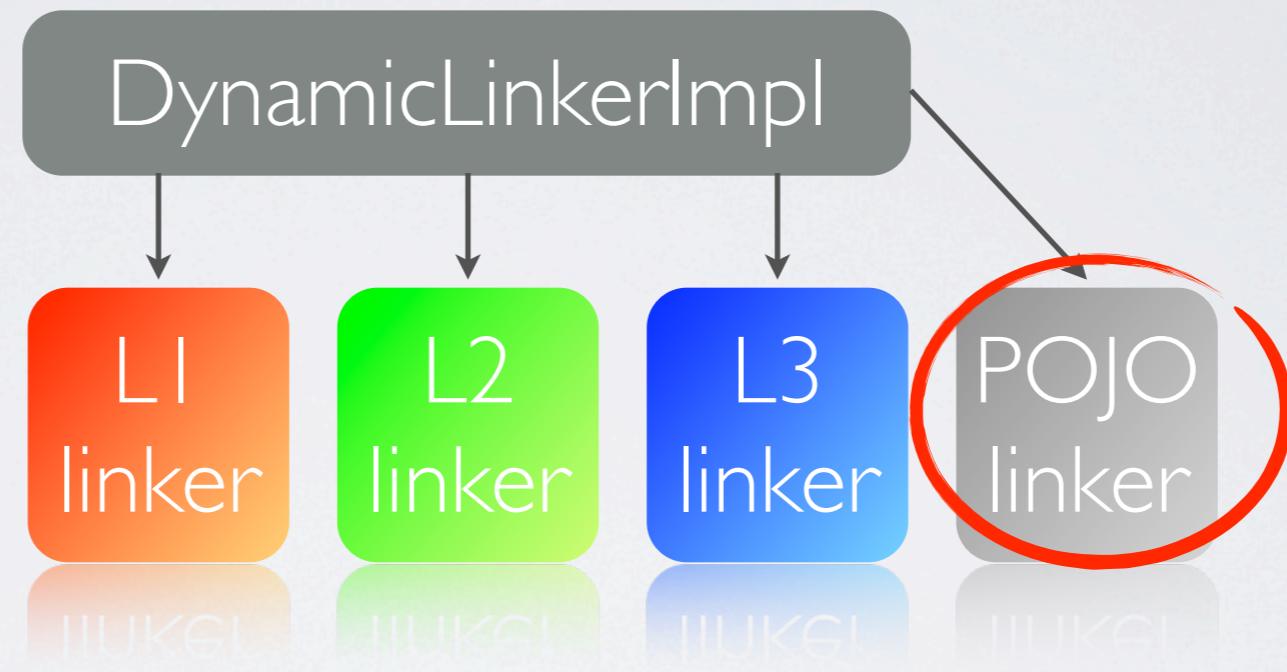
LINKER FACTORY

```
private static final DynamicLinker linker = new DynamicLinkerFactory().createLinker();
```



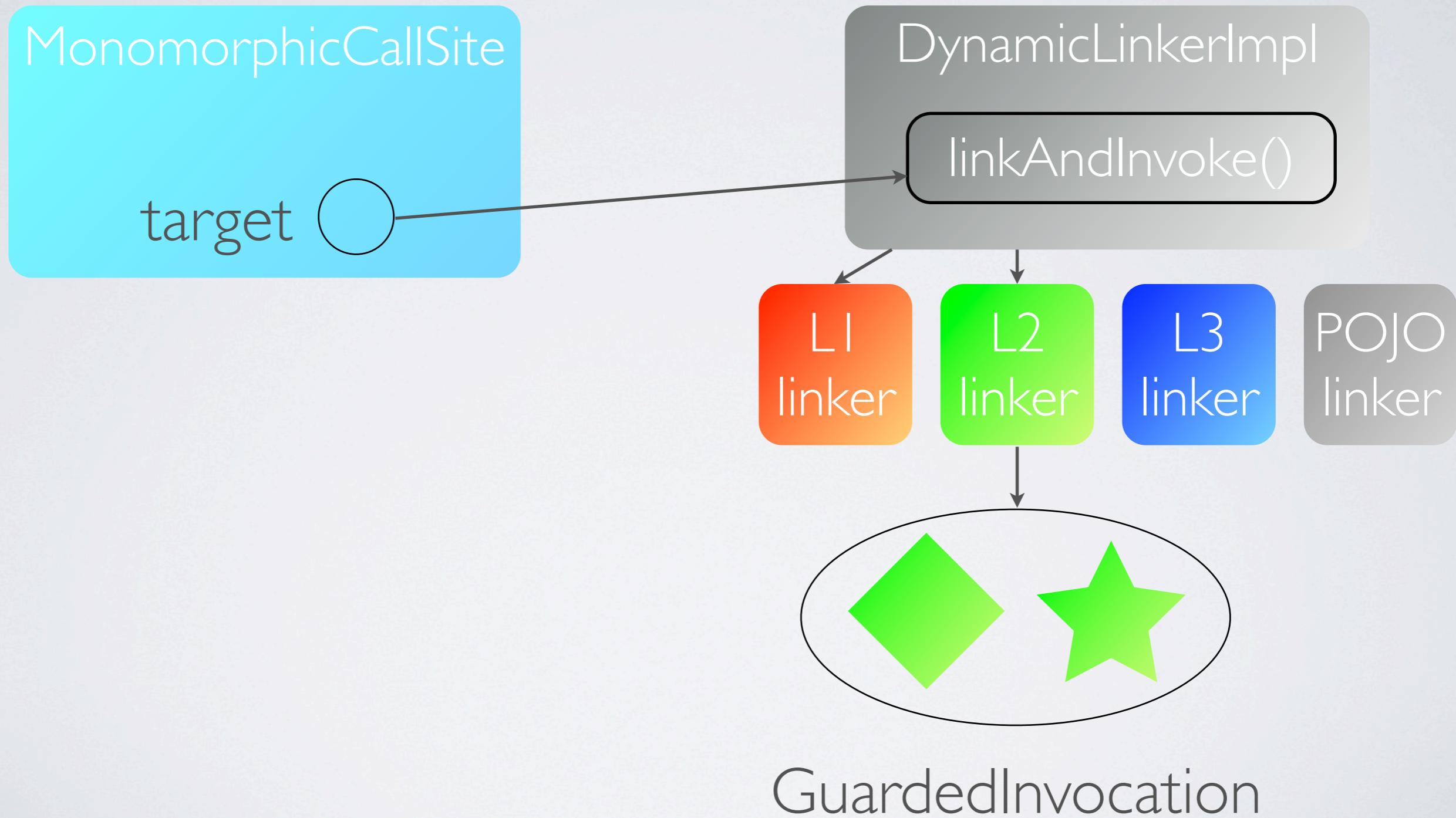
/META-INF/services/org.dynalang.dynalink.GuardingDynamicLinker

DYNAMIC LINKER

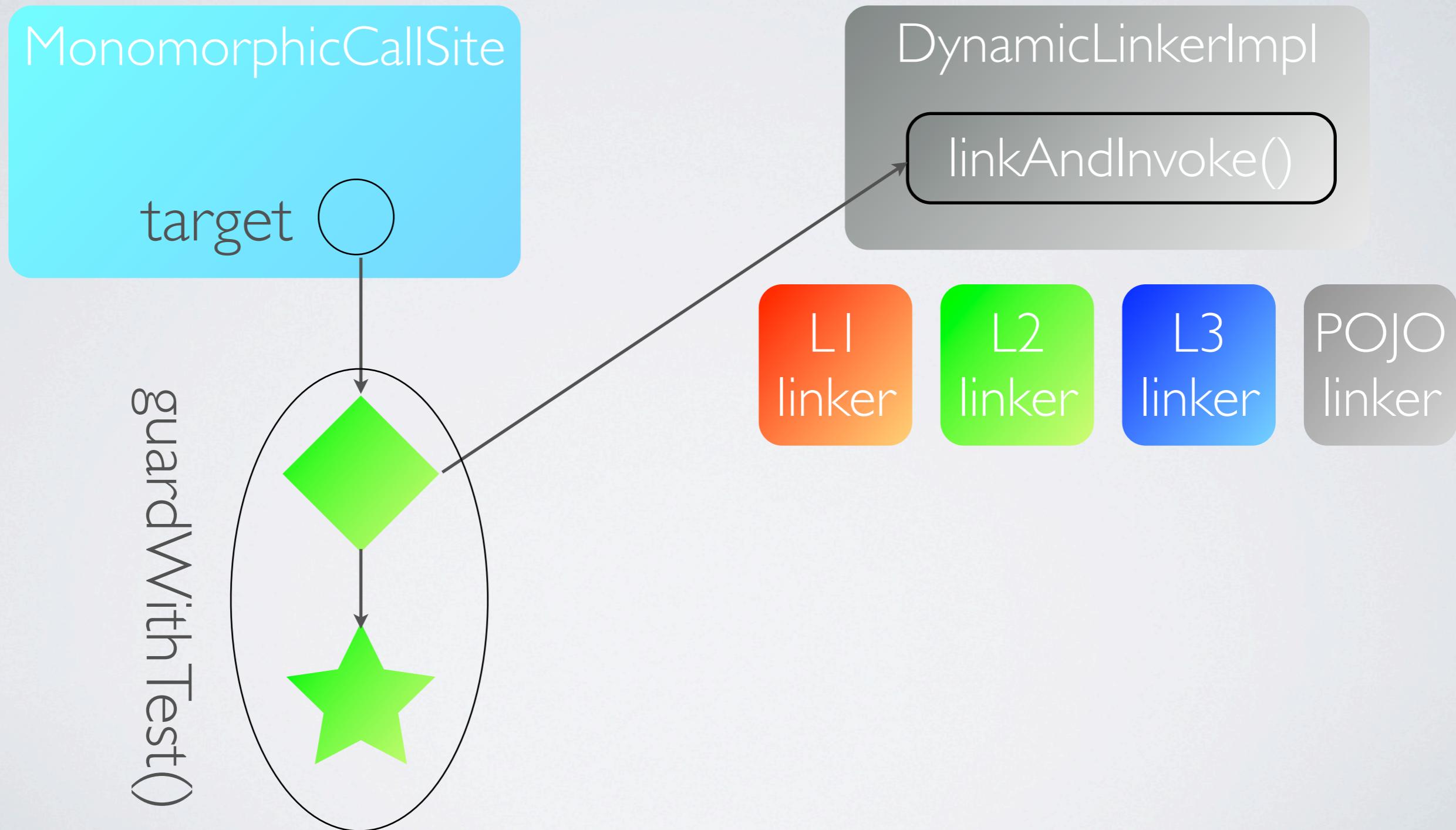


- linkers are interrogated sequentially
 - not a big deal, but can be optimized for class-based dispatch
 - you can prioritize your own language

LINKING A CALL SITE



LINKING A CALL SITE



GUARDING LINKER

```
package org.dynalang.dynalink;

public interface GuardingDynamicLinker {
    /**
     * Creates a guarded invocation appropriate for a particular invocation
     * with the specified arguments at a call site.
     * @param callSiteDescriptor the descriptor of the call site
     * @param linkerServices linker services
     * @param arguments the arguments for the invocation
     * @return a guarded invocation with a method handle suitable for the
     * arguments, as well as a guard condition that if fails should trigger
     * relinking. Must return null if it can't resolve the invocation. If the
     * returned invocation is unconditional (which is actually quite rare), the
     * guard in the return value can be null.
     * @throws Exception if the operation fails for whatever reason
    */
    public GuardedInvocation getGuardedInvocation(
        CallSiteDescriptor callSiteDescriptor,
        LinkerServices linkerServices, Object... arguments)
    throws Exception;
}
```

GUARDED INVOCATION

```
package org.dynalang.dynalink;

public class GuardedInvocation
{
    private final MethodHandle invocation;
    private final MethodHandle guard;
    ...
    public GuardedInvocation(MethodHandle invocation, MethodHandle guard) {
        ...
    }
}
```

PRIORITIZING YOUR OWN LINKER

```
import java.dyn.*;  
import org.dynalang.dynalink.*;
```

```
class MyLanguageRuntime {  
    private static final DynamicLinker linker = createLinker();
```

```
private static DynamicLinker createLinker() {  
    DynamicLinkerFactory factory = new DynamicLinkerFactory();  
    factory.setPrioritizedLinker(new MyLanguageLinker());  
    return factory.createLinker();  
}
```

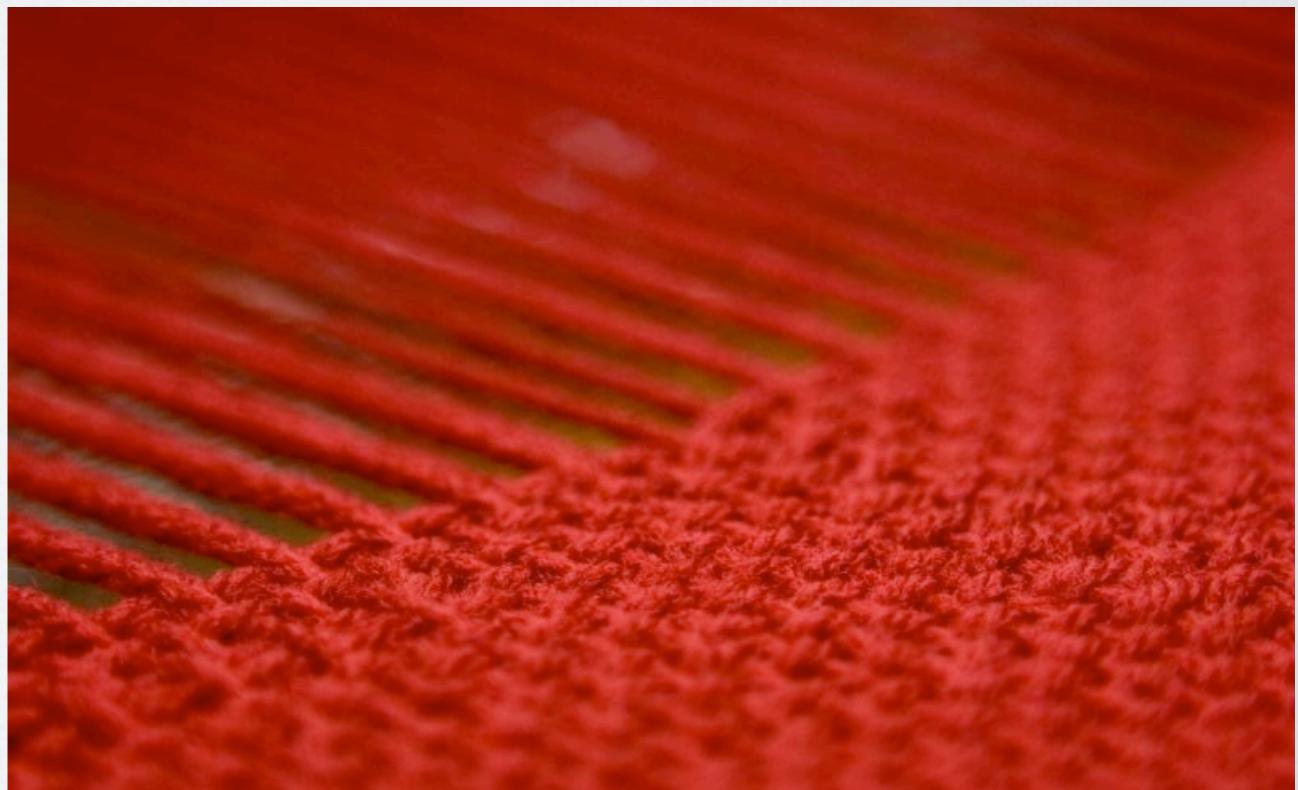
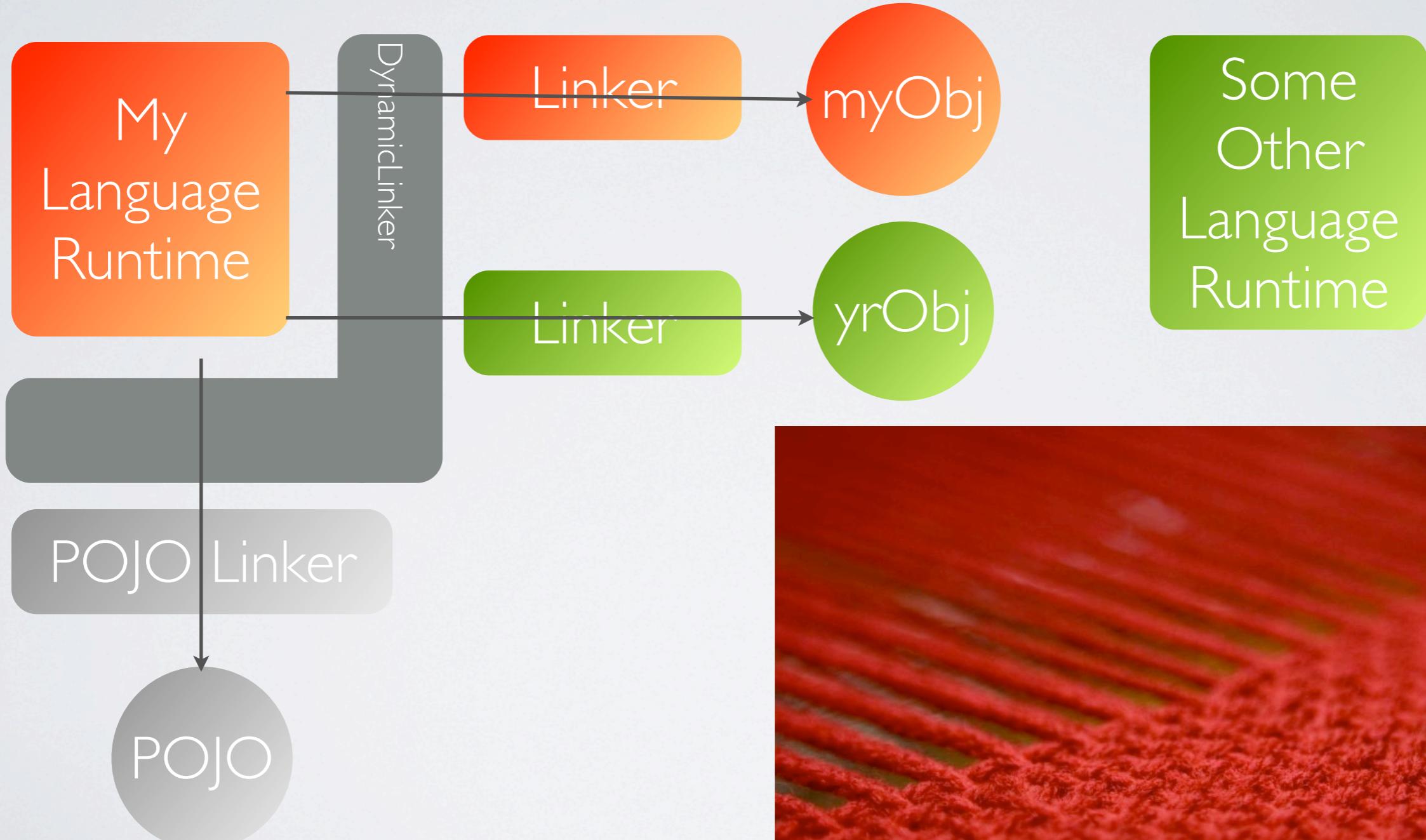
```
public static CallSite bootstrap(Class caller, String name, MethodType type) {  
    RelinkableCallSite callSite = new MonomorphicCallSite(caller, name, type);  
    linker.link(callSite);  
    return callSite;  
}
```

L2
linker

L1
linker

L3
linker

UNIFIED API FOR OBJECT HANDLING



WHAT'S A LANGUAGE IMPLEMENTER TO DO?

- **ADOPT** it (vertical integration), and
- **ENABLE** it (horizontal integration).
- Start using the linker framework to adopt it.
 - Immediate benefit: **POJO Linker** 
 - Also, can shield you from JSR-292 changes
 - Implement a linker to enable it, then...
 - ... access your own objects through it
 - eat
 - dog food



FREE!

POJO Linker

- JavaBeans property getters and setters
- Method invocation
 - Handles overloaded methods, variable arity methods, ...
 - ... and combinations of the two (you can ask me about it later)...
 - ... and even situations not covered by JLS but present in dynamic binding
- Container operations for arrays, collections, and maps



FREE!

POJO Linker

- Does all convertArguments(), filterArguments(), collectArguments(), insertArguments(), dropArguments(), guardWithTest() etc. gymnastics for you behind the “*dyn:someOperation*” facade
- Not currently handled:
 - enums, public fields (easy to add)
 - non-public access, static methods, constructors
 - were solved in last year’s MOP, but I’m not happy with it (another workshop discussion point).



FREE!

POJO Linker

- It'll continue to be optimized
- plenty of caching opportunities
 - caching is hard - when is it safe to strongly reference?
- opportunistically broad guards with deoptimization on horizon
- since you program for the interface, you'll also get



FREE!

Performance enhancements

LANGUAGE-SPECIFIC TYPE CONVERSIONS

```
package org.dynalang.dynalink;

public interface GuardingTypeConverterFactory
{
    /**
     * Returns a method handle that receives an Object of the specified source
     * type and returns an Object converted to the specified target type. The
     * type of the invocation is targetType(sourceType), while the type of the
     * guard is boolean(sourceType).
     * @param sourceType source type
     * @param targetType the target type.
     * @return a guarded invocation that can take an object (if it passes
     * guard) and returns another object that is its representation coerced
     * into the target type. In case the factory is certain it is unable to
     * handle a conversion, it can return null.
     */
    public GuardedInvocation convertToType(Class<?> sourceType, Class<?> targetType);
}
```

LANGUAGE-SPECIFIC TYPE CONVERTERS

- Converter chains constructed using `guardWithTest()`, injected using `filterArguments()`
- But only when
 - needed to negotiate call site type with method type,
 - and built-in Java conversion isn't applicable
- Converter factories can return null to declare they can't participate in the conversion (shorter chains, faster conversion)

TYPE CONVERTERS EXAMPLE

- Call site: foo(IRubyObject, Object, String)
- Method: foo(Number, Number, Object)

IRubyObject -> Number	guardWithTest(rubyTest(), rubyConverter(), identity)
Object -> Number	guardWithTest(rubyTest(), rubyConverter(), guardWithTest(rhinoTest(), rhinoConverter(), identity))
String -> Object	no conversion needed

SO, CAN THIS REALLY BE DONE?

- I think yes. No insurmountable obstacle on the horizon
- .Net has DLR, after all
 - we are not using that approach, although it could be investigated.
- Community adoption will make it or break it

WHERE IS IT?

<https://dynalang.svn.sourceforge.net/svnroot/dynalang/trunk/Invoke>

- README contains the currently understood dyn: ops
- Lots of JUnit tests (and expanding)
 - 74% block coverage (measured with EMMA)
- Usable with both native implementation (MLVM) and Rémi's backport (JUnit executed with both)
 - Although I do run into bugs and incompleteness in

QUESTIONS

ACKNOWLEDGMENTS

The following photos were used in this presentation, in compliance with their Creative Commons licenses:

green jelly photo: <http://www.flickr.com/photos/reggaedori/3351891756/>

container loading photo: <http://www.flickr.com/photos/elbfoto/2201493832/>

little red phone photo: <http://www.flickr.com/photos/betsyjean79/1353043704/>

weaving photo: <http://www.flickr.com/photos/lollyknit/2239377452/>

CONTACT

<mailto:szegedia@gmail.com>

<http://twitter.com/szegedi>