

GEP 10 - Static compilation

Metadata

Number:	GEP-10
Title:	Static compilation
Version:	3
Type:	Feature
Status:	Draft
Leader:	Cédric Champeau
Created:	2011-11-23
Last modification:	2012-06-21

Abstract: Static compilation

This GEP introduces an experimental new feature in the language known as *static compilation*. Static compilation can be used when the dynamic features of Groovy are not necessary and that the performance of the dynamic runtime is too low. The reader must understand that :

- we do not plan to change the semantics of regular Groovy
- we want static compilation to be triggered explicitly
- we want the semantics of static groovy to be as close as possible as the semantics of dynamic Groovy

Rationale: Static Type Checking vs Static compilation

Static compilation heavily relies on another feature called [Static type checking](#) but it is important to understand that they are separate steps. You can statically type check your code, and still have the dynamic runtime at work. Static type checking adds type inference to the compilation process. This means that, for example, type arguments of method calls are inferred, as well as return types of closures or methods. Those inferred types are used by the checker to verify the type flow of your program. When it does so, once must be aware that the **static checker cannot behave like dynamic Groovy**. This means that even if a program passes static type checking, it *may* behave differently at runtime.

Method selection

Method selection in dynamic Groovy

Groovy supports multimethods. Especially, in dynamic Groovy, target methods are chosen at runtime, in opposite to Java which chooses target method at compile time. Let's illustrate the difference with an example :

```
public void foo(String arg) {
    System.out.println("String"); }
public void foo(Object o) {
    System.out.println("Object"); }
```

```
Object o = "The type of o at runtime is
String";
foo(o);
```

In Java, this code will output "Object", because the target method is chosen according to the type of the arguments at compile time. The declared type of "o" is Object, so the Object version of the method is chosen. If you want to call the String version, you have to define "o" as a "String", or cast o to string in the method call arguments. Now the same example in dynamic Groovy :

```
void foo(String arg) { println 'String' }
void foo(Object o) { println 'Object' }

def o = 'The type of o at runtime is String'
foo(o)
```

If you run this snippet, you will see that Groovy prints "String". The reason is that Groovy resolves type arguments at runtime and chooses the most specialized version of a method signature when multiple versions are possible. To deal with multiple arguments, Groovy computes a distance between the actual type arguments and the declared argument types of a method. The method with the best score is chosen.

The method selection process is complex, and makes Groovy quite powerful. It is also possible, for example, to define metaclasses which will choose different methods, or change the return type of a method dynamically. This behaviour is responsible for a large part of the poor performance of Groovy invocation times as compared to Java. However, performance has greatly improved with the introduction of call site caching in Groovy. Even with those optimizations, Groovy is far from the performance of a pure static language.

InvokeDynamic

InvokeDynamic support is under development and should be introduced in the upcoming Groovy 2.0. What InvokeDynamic allows us to do is to, basically, replace call site caching with a native "dynamic method dispatch" system directly implemented by the JVM. It is still uncertain what performance improvement we will reach. At best,

we could be close to the performance of a statically typed language. However, as some features are still difficult to implement with InvokeDynamic, most likely the performance gain will not be so high. It is important to talk about it though, because :

- invokedynamic allows JIT optimizations
- replaces the current dynamic method dispatch without changing the semantics of the language
- if performance is good, could make static compilation unnecessary

However, InvokeDynamic has a major drawback : it will only be available for people using a Java 7+ JVM. If we want to deliver Java-like performance for Groovy programs to our users, can we afford leaving most of them without such? Most probably, Java 7 won't be mainstream before two or three years.

[Rémi Forax](#), however, created a backport of invokedynamic for older versions of the JVM. Such a backport relies on bytecode transformation to replace invokedynamic instructions with "emulated instructions". This "emulated" mode is great for us Groovy developers, because it would allow us to write code once and run it on any JVM, but for users, the performance would most probably be bad. To be sure of that, an experiment with backported InDy will be made once invokedynamic support is implemented in Groovy core. The most probable situation, though, is that performance of such code will remain far from what a static language could provide.

Static Groovy

Type inference based dispatch

This GEP is there to experiment a static compilation mode in Groovy. With what we have explained before, you should already understand that statically typed code implies a different behaviour from dynamic code. If you expect the statically checked and statically compiled code to behave exactly like dynamic Groovy, you should already stop there, or wait for invoke dynamic support to expect improved performance. If you perfectly understand that statically compiled code means different semantics, then you can continue reading this GEP, and help us choose the best implementation path. In fact, there are several options that we will explain here.

The current implementation relies on the static type checker, which performs type inference. This means that with the previous example :

```
void foo(String arg) { println 'String' }
void foo(Object o) { println 'Object' }

def o = 'The type of o at runtime is String'
foo(o)
```

The compiler is able to infer that when the *foo* method is called, the actual type argument **will** be a string. If we compile it statically, the behaviour of this statically compiled program at runtime will be the same as dynamic Groovy. With such an implementation, we expect most programs to behave statically like they would in dynamic Groovy. However, this will never be always true. This is why we say this behaviour is "as close as possible" as the one of dynamic Groovy.

The drawback of this implementation is that the developer cannot easily know what method the compiler chooses. For example, let's take this example, extracted from a discussion on the mailing list:

```

void foo(String msg) { println msg }
void foo(Object msg) { println 'Object' }

def doIt = {x ->
    Object o = x
    foo(o)
}

def getXXX() { "return String" }

def o2=getXXX()
doIt o2 // "String method" or "Object
method"????

```

The static type checker infers the type of *o2* from the return type of *getXXX*, so knows that *doIt* is called with a *String*, so you *could* suspect the program to choose the *foo(String)* method. However, *doIt* is a closure, which can therefore be reused in many places, and the type of its "x" argument is unknown. The type checker will not generate distinct closure classes for the different call sites where it is used. This means that when you are in the closure, the type of 'x' is the one declared in the closure arguments. This means that without type information on 'x', x is supposed an Object, and the *foo* method which will be statically chosen will be the one with the Object argument.

While this can be surprising, this is not really difficult to understand. To behave correctly, you must either add explicit type arguments to the closure, which is always preferred. In a word, in a statically checked world, it is preferred to limit the places where types will be inferred so that code is understood properly. Even if you don't do it, fixing code is easy, so we think this is not a major issue.

Java-like method dispatch

The alternative implementation is not to rely on inferred types, but rather behave exactly like Java does. The main advantage of this is that the user doesn't have 3 distinct method dispatch modes to understand, like in the previous solution (Java, dynamic Groovy, inferred static Groovy). The major drawback is that the semantics of static Groovy are not close to the ones of dynamic Groovy. For this reason, this is not the preferred experimental implementation. If you think this version should be preferred, do not hesitate to send an email to the mailing list so that we can discuss. You can even fork the current implementation to provide your own.

Testing

Static compilation is now part of the Groovy 2.0.0 release. You can download the latest Groovy 2 releases and test it.

@CompileStatic

Static compilation is for the moment only supported at the method level (do not try to add it to a class) and supports direct method calls as long as you do not:

- use methods from *DefaultGroovyMethods*
- use "indirect" methods like `list << obj` where the AST representation is not a method call but a binary expression

For example, you can try the following snippet :

```
@groovy.transform.CompileStatic
int fib(int i) {
    i < 2 ? 1 : fib(i - 2) + fib(i - 1)
}
```

This code should already run as fast as Java.

The "arrow" operator for direct method calls

Some users have suggested another idea, related to static compilation, which is the "arrow operator". Basically, the idea is to introduce another way of calling methods in Groovy :

```
obj.method() // dynamic dispatch
obj->method() // static dispatch
```

While this idea sounds interesting, especially when you want to mix dynamic and static code in a single method, we think it has many drawbacks. First, it introduces a grammar change, something we would like to avoid as much as possible. Second, the idea behind this operator is to perform direct method calls when you **know** the type of an object. But, without type inference, you have two problems :

- even if the type of 'obj' is specified, you cannot be sure that at runtime, the type will be the same
- you still have to infer the argument types too, which leaves us with the same problem as before: relying on type inference, or relying on Java-like behaviour where the method is chosen based on the declared types. If we do so, then we would introduce possible incompatibility with the static mode... So we would have to choose between this mode and the full static compilation mode.

Imagine the following code :

```

void write(PrintWriter out) {
    out->write('Hello')
    out->write(template())
}

def template() { new MarkupBuilder().html {
p('Hello') } }

```

While the first call can be resolved easily, this is not the same with the second one. You would have to rewrite your code probably this way to have this work :

```

void write(PrintWriter out) {
    out->println('Hello')
    String content = template() // dynamic
call, implicit coercion to string
    out->println(content) // static method
dispatch based on declared types only
}

def template() { new MarkupBuilder().html {
p('Hello') } }

```

Which is not necessary if you use the @CompileStatic annotation :

```
@CompileStatic
void write(PrintWriter out) {
    out.println('Hello')
    out.println(template())
}

def template() { new MarkupBuilder().html {
p('Hello') } }
```

References

Mailing-list discussions

- [\[groovy-dev\] Static compilation for Groovy](#) : An interesting discussion where dynamic lovers explain why they do not want static compilation

JIRA issues

- [GROOVY-5138: GEP 10 - Static compilation](#)

Useful links

- Blackdrag's blog [post about flow typing](#)
- Cedric Champeau's blog [post about Grumpy update](#)