Golo, a Dynamic, Light and Efficient Language for Post-Invokedynamic JVM

Julien Ponge, Frédéric Le Mouël, Nicolas Stouls

Golo, a Dynamic, Light and Efficient Language for Post-Invokedynamic JVM

Julien Ponge Frédéric Le Mouël Nicolas Stouls
Université de Lyon
INSA-Lyon, CITI-INRIA F-69621, Villeurbanne, France
firstname.lastname@insa-lyon.fr

Abstract

This paper introduces *Golo*, a simple dynamic programming language for the Java Virtual Machine (JVM) that has been designed to leverage the capabilities of the new Java 7 `invokedynamic` instruction and API (JSR 292). Golo has its own language constructs being designed with `invokedynamic` in mind, whereas existing dynamic languages for the JVM such as Groovy, JRuby or Nashorn have to adapt language constructions which are sometimes hard to optimize. Coupled with a minimal runtime that directly uses the Java SE API, Golo is an interesting language for rapid prototyping, polyglot application embedding, research (e.g., runtime extensions, language prototyping) and teaching (e.g., programming, dynamic language runtime implementation). We show that the language design around `invokedynamic` allows for a very concise runtime code base with performance figures that compare favorably against Java and other dynamic JVM languages. We also discuss its future directions, either as part of Golo or through language and runtime research extensions.

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: Language Classifications—Multiparadigm languages; D.3.4 [*Programming Languages*]: Processors—Compilers

General Terms Languages

Keywords Golo, invokedynamic, Java, JVM

1. Introduction

The JVM ecosystem goes way beyond the sole Java programming language and a myriad of frameworks and libraries. The JVM itself provides a proven adaptive, managed and efficient runtime environment for a wide range of programming languages. While the JVM had an initial bias that favored statically-compiled languages, the interest in dynamically-typed languages prompted developments as part of JSR 292 to better support such languages starting from Java 7 [17, 18]. The runtime of existing dynamic languages are evolving to take advantage of this, leading to both better performance and simplification of runtime implementation [3, 4, 6].

PPPJ'13, September 11–13, 2013, Stuttgart, Germany.

http://dx.doi.org/10.1145/2500828.2500844

While working on middleware for dynamic applications, the authors often felt the need for modifying existing languages. Most well-known JVM languages have significant code base sizes, which make them arguably hard to modify or extend. This is one of the reasons for introducing *Golo*, a simple dynamic programming language for the JVM. Golo has simple semantics and a minimal runtime. It takes advantage of the Java SE API, or any API from another JVM language. It builds on JSR 292, making it one of the first language to have been designed around it.

We believe that Golo makes for an appealing language for research purposes, as it is easy to derive dialects from it, or make experiments with new runtime features. We believe that it has some educational value, both as a programming language and as a reference of how to build a dynamic language using `invokedynamic` on the JVM. We are also convinced that the simple design in Golo makes it useful in real-world polyglot applications, as the early community interest shows.

This paper starts with an informal tour of the Golo programming language, showing examples of the current constructions that it provides. We then focus on the implementation of the Golo runtime on the JVM, taking advantage of JSR 292 as a building block for most dispatch and dynamic selection operations. We then proceed to some early evaluation of the runtime performance based on micro-benchmarks, showing that it compares favorably against Java and existing dynamic languages for the JVM. We finally provide related work and give perspectives.

2. A tour of Golo

This section is an informal tour of Golo, presenting its main constructions and features. A complete guide to the language is made available on the Golo website [2].

2.1 Basics

Golo arranges source code definitions in *module* files that primarily define *functions*. A classic *"hello world"* example looks as follows:

```
module hello.world

function main = |args| {
  println("Hello, world!")
  let a = 1
  var b = "Hello"
  b = b + 123 + a     # b is a mutable reference
}
```

As one may guess, the `main` function serves as the program entry point, and parameters are given between *pipe* symbols (`'|'`). The `println` function is a predefined function that delegates to Java `System.out.println()` under the hood. Golo is a dynamically-typed language, where all values are objects, according to the JVM

bottom type `java.lang.Object`. Constant object references are introduced by the `let` clause, while mutable references defined by `var`. Note that (1 line) comments begin with a *sharp* symbol (`'#'`).

Visibility. By default, functions are visible to other modules. A function visibility can be restricted by prefixing its declaration by `local`, meaning that it can only be called from functions declared within the same module.

Imports. The `main` function above can be called from another module function by prefixing the call with the module name, as in: `hello.world.main(someArray)`. Golo supports *import* statements to facilitate symbol resolution at runtime. With: `import hello.world`, the previous call reduces to: `main(someArray)`.

Import statements are being resolved in order of their declaration in a module. Simply put, when a function is being called and its name cannot be resolved, each *import* definition is tried in declaration order to match it against a target.

Operators. Golo provides operators in the tradition of many C-style languages. Arithmetic operations are made using `+, -, *, /, %` operators. Value comparisons use `<, <=, ==, !=, >, >=` while reference can be compared using `is, isnt`. It must be noted that `==` in Golo is `Object#equals(Object)` in Java, and we believe that the distinction between `==` and `is` makes sense as Java developers frequently confuse reference and value equality. Boolean expressions can be composed using the logical `or, and, not` operators. Finally, we provide a `orIfNull` binary operator which is especially useful for dealing with `null` values and provide a default value to an expression, as illustrated by:

```
println(null orIfNull "N/A") # => "N/A"
```

2.2 Creating Java objects from Golo

Golo is not just made to run on top of the JVM: it embraces the Java Standard API. When a (Java) class provides a public *static* method or field, it can be called from Golo as a function, with the class qualified name acting as a prefix unless a corresponding *import* is being used. Object instances are created by calling a class constructor as if being a Golo function. Calling instance methods is being done using the `':'` operator, where the left-hand side is the receiver object or expression. Here is an example:

```
let list = java.util.LinkedList()
list: add(1)
list: add(2)
list: add(Integer.MAX_VALUE())
println(list)
```

It creates an instance of `java.util.LinkedList`, calls its `add` method several times, the last one with the value of the public static field `MAX_VALUE` of the `java.lang.Integer` class.

Golo also provides `null`-safe method invocations with the `'?:'` operator, also called *"elvis operator"* in languages such as Groovy or Kotlin. Invoking a method on a `null` reference raises a `NullPointerException`, but the `?:` operator traps those and simply returns `null`. Such invocations are especially useful when combined with the `orIfNull` operator, as it provides a convenient way to traverse an object graph with potentially `null` nodes, and return a default value as in:

```
let person = dao: findByName("Mr Bean")
let city = person?: address()?: city() orIfNull "n/a"
```

This single expression consists of chained method calls, and evaluates to either the city of some entity object retrieved from some data store, or `"N/A"` if either of the address or entity is `null`. This is arguably cleaner than:

```
let person = dao: findByName("Mr Bean")
var city = "n/a"
if person isnt null {
  let address = person: address()
  if address isnt null {
    city = address: city() orIfNull "n/a"
  }
}
```

2.3 Control flow

As suggested by the last example, Golo supports conditional branching using `if, else` statements. Loops can be expressed using `while`, `for` and `foreach` constructions. `for`-loops consist of strictly 3 parts: initialization, termination condition and iteration expression.

```
let a = Array(1, 2, 3)
for (var i = 0, i < a: length(), i = i + 1) {
  println(">>> " + a: get(i))
}
```

`foreach`-loops can yield to simpler expressions on JVM arrays and objects implementing `java.lang.Iterable`:

```
let a = Array(1, 2, 3)
foreach i in a {
  println(">>> " + i)
}
```

Conditional expressions consisting of many different cases can be better expressed using `case` constructs rather than successive `if, else if, ..., else` statements:

```
case {
  when a() { ... }
  when b() { ... }
  ...
  otherwise { ... }
}
```

Finally, Golo provides a `match` statement that evaluates several conditions and returns a value:

```
function what_could_it_be = |item| {
  return match {
    when item: contains("@") then "an email?"
    when item: startsWith("+33") then "a phone number?"
    when item: startsWith("http://") then "a URL?"
    otherwise "I have no clue..."
  }
}
```

Both `case` and `match` statements compile equivalently to a cascade of `if` statements.

2.4 Exceptions

Golo uses Java exceptions, but makes no distinction between *checked* and *unchecked* exceptions. Exception-handling statements mimic those of Java: `try / catch / finally`, `try / catch` and `try / finally`. Similarly, exceptions are being thrown using the `throw` keyword. Here is a `try / catch` construct example:

```
try {
  doSomethingDangerous()
} catch (e) {
  logger: info(e)
  throw e
}
```

The type of the exception can be checked using the `oftype` operator, as in:

```
try { ... }
catch (e) {
  case {
    when e oftype IOException.class { ... }
    when e oftype SecurityException.class { ... }
    otherwise { ... }
  }
}
```

2.5 Class augmentations

Golo provides *augmentations* to existing Java classes. This is similar to *extension methods* in C#. It acts as "syntactic sugar" to add methods to existing classes. An `augment` declaration is defined at the module level, as the following example shows:

```
module ConcatenableStrings

augment java.lang.String {
  function concatWith = |this, args...| {
    var result = this
    foreach arg in args {
      result = result + arg
    }
    return result
  }
}
```

An augmentation applies to a fully-qualified Java class, and consists of several functions, each acting as a "method" to be added. Each augmentation function must have a first argument that acts as a receiver object. The previous example adds a `concatWith` method to strings. Note that `args` is a variable argument parameter, and is dealt with as an array. The following would print `"abcd"` to the standard console:

```
println("a": concatWith("b", "c", "d"))
```

Augmentations are visible from code within the same module. They can be used from other modules too. This requires an import, so a module gets all possible augmentations from each module it imports. As a matter of style, it is suggested that reusable augmentations get isolated in dedicated modules, so as to avoid accidental availability of an augmentation. Finally, augmentations do not override existing methods: a method such as `java.lang.Object#toString()` cannot be overridden by augmentations.

2.6 Functions as first-class objects

Closures. Like many modern languages, functions are handled as first-class objects. They can be referenced and declared just like normal objects, and passed as parameters to other functions and methods. They capture their lexical context (*closures*), so that references to objects can be passed to a locally-declared function:

```
let a = 3
let f = |x| -> x + a
let g = |h, x| -> ">>> " + h(x)
println(f(1))     # => "4"
println(g(f, 1))  # => ">>> 4"
```

Note that Golo provides a short notation for functions consisting of a single expression, using `->` between arguments and an expression.

Functions as objects. A closure reference is an instance of `MethodHandle` from `java.lang.invoke`, meaning that we can take advantage of the *combinators* API that it provides. As an example, the following is a *partial application* that takes advantage of the `MethodHandle#bindTo` combinator:

```
let adder = |a, b| -> a + b
let add2 = adder: bindTo(2)
println(add2(1)) # prints '3'
```

Leveraging augmentations. Golo comes with a set of standard augmentations to enhance Java collections, method handles, iterable objects and more. While it would be too lengthy to list them, the following example illustrates them:

```
# Prints "[3, 5, 7]"
let incr = |x| -> x + 1
println(LinkedList():
  append(1, 2, 3, 4, 5, 6):
  map(incr: andThen(incr)):
  filter(|x| -> (x % 2) == 1))
```

`append` is an augmentation on lists that allows adding several values at once to a collection. Indeed, the `java.util.List` API offers of a `void add(Object)` method that requires several, non-chained invocations. `map` and `filter` implement the common eponymous functional operations. Finally, `andThen` allows to compose function references.

2.7 Dynamic objects

Golo provides *dynamic objects*, that is, objects whose properties and methods can be defined on a per-instance basis, like in Groovy, Python or Ruby [3–5]. They can be used as in:

```
let mrbean = DynamicObject():
  name("Mr Bean"):
  email("mrbean@gmail.com"):
  define("toString", |this| -> this: name() + " <" +
         this: email() + ">")

println(mrbean: name())     # 'getter'
mrbean: name("Sir Mr Bean") # 'setter'
println(mrbean: toString()) # 'method'
```

Values can be get and set, providing methods with the corresponding name. They can either be set by name, or by calling the `define` reserved method. When the value to be set is a function, then it defines a method, which in turn can be used. Note that such functions shall have at least a first argument to act as a receiver.

Dynamic objects provide further operations. Properties can be unset, updated and frozen, in which case a dynamic object becomes immutable[1]. A dynamic object can be cloned using the `copy()` reserved method. Finally, a dynamic object can be mixed into another one, as in" `obj: mixin(other)`. This copies every property of `other` into those of `obj`, overwriting existing entries.

3. Implementing Golo on the JVM

This section discusses how the Golo runtime has been designed and implemented on top of the JVM. After an overview, we discuss how bytecode is being generated ahead-of-time. Finally, we show how JSR 292 allows for a thin and efficient runtime design.

[1] At least at the level of its own properties. This does not prevent a property value to be an instance of a mutable class.

3.1 Overview

The Golo runtime and compiler fit into a single uncompressed Jar archive of 225KB as of version `0-preview4`. The grammar of Golo is written using the $LL(k)$ *JJTree* / *JavaCC* parser generator [12], mainly due to its simplicity and lack of a runtime dependency, as it generates all the Java code required for a working parser. The front-end generates an abstract syntax directly from *JJTree*, which is then transformed into an intermediate representation based on a Golo-specific object model, comprising classes to model reference lookups, functions, common statements and so on.

The intermediate representation is visited by a limited number of phases. The first one expands closures into synthetic functions, so that a function `f` declared as:

```
let a = 3
let f = |x| -> x + a
```

yields a synthetic local function `$f = |$a, x| ...` at the module level. The second phase visits the intermediate representation to verify and assign local references. It checks that symbolic references are meaningful within the context in which they are being used. It assigns each object reference an index number. It also reports undeclared references as well as assignments to constants references (`let`). The third and last phase generates JVM bytecode from the intermediate representation. Golo uses ahead-of-time compilation. Code generation is done in a single pass using the ASM library [8].

3.2 Bytecode generation

The Golo compiler generates one JVM class per Golo module, and the class name simply matches those of the module. A module `foo.Bar` generates a class `Bar` in package `foo`. All functions are compiled to static methods, and type signatures are generic, that is, they use `java.lang.Object`. Functions are public static methods unless they are declared as `local`. Variable arguments are honored in the bytecode flags: a function `f` with parameters `|head, tail...|` compiles to a static method `Object f(Object head, Object[] tail)`. Closures are compiled to private, static methods whose names are manglings of `"__$$_closure_"` and incremented integers.

Each class augmentation is compiled to a separate class. Given an augmentation on `java.lang.String` defined in the `better.Strings` module, a class `better.Strings$java$lang$String` is being generated. Augmentation functions are being generated to this class like other functions.

Generated classes embed meta-data in the form of public static methods. `$imports()` returns an array of strings that corresponds to the module imports, in order. This is being used by the Golo runtime to dynamically resolve names against imports. Similarly, a `$augmentations()` method returns an array of strings with the augmentations that the module provides. Again, this is being used at runtime to help in missing method resolution. Augmentation-generated classes only have an `$imports()` method to provide meta-data.

3.3 `invokedynamic`-based call sites

The bytecode produced by Golo is fairly traditional for a language on the JVM, except that it makes extensive use of the `invokedynamic` instruction for every function and method invocation. `invokedynamic` instructions are given a generic signature for the corresponding call site: a function call such as `foo(bar, baz)` yields an `invokedynamic` instruction with the symbolic name `foo` and a `(Object, Object) Object` signature.

`invokedynamic` instructions are linked to a *bootstrap* method whose role is to initialize the corresponding call site the first time it is being executed. Golo provides the following specialized call sites, for which we also describe the arguments.

1. Functions support: lookup, name and type.
2. Operator support: lookup, name, type and arity (1 or 2).
3. Method support: lookup, name, type and an integer flag for whether calls shall be `null`-safe or not.
4. Class reference: lookup, name and type.
5. Closure reference: lookup, name, type, declaration module name, arity and an integer flag for variable arguments.
6. Closure invocation support: lookup, name, type.

Most call sites returned by these bootstrap methods are instances or derivatives of `MutableCallSite`. There are two exceptions: class and closure reference call sites return instances of `ConstantCallSite`, as they remain constant once their initial lookup has been made.

3.4 Dispatching calls

Function. The dynamic nature of Golo takes advantage of the call site mutability. The case of function calls is fairly simple, as the first invocation triggers a lookup, first in the current module, then by iterating through the imported modules. When a target function is found, the call site is updated with it and it remains constant through the execution. Not finding it triggers an exception.

Operators. Operators are implemented using specialized static methods, e.g., an addition method for `(Integer, Integer)`, `(Integer, Long)` and so on. Operator call sites construct inline caches [11] using the `guardWithTest` method handle combinator of the `java.lang.invoke.MethodHandles` class. To do that, they have a guard condition that looks at the arguments, a method handle to the specialized static method, and a fallback method handle that performs another lookup and call site mutation. Consider the case of additions: when a call site sees arguments of classes `Integer` and `Integer`, it loads a method handle to the static method doing additions on two integers. The call site uses this method handle for as long as calls are being performed on two integers. If, say, a call happens on one integer and a double, the guard check fails, causing a call site invalidation.

Methods. Method invocations also employ an inline-cache construction. As receiver objects are being discovered at a call site, method handles to their target methods are being selected and cached, forming polymorphic inline-cache trees [11] of `guardWithTest` combinators. The corresponding guards test on the receiver class. The fallback handles perform new lookups as new receiver classes are being discovered, rewriting call site targets with a new `guardWithTest` where the fallback is the previous target, effectively assembling them in a tree. Because the fallback method handle has to be generic for different types of call site signatures, it uses the `asCollector` combinator to wrap arguments into an array, and `bindTo` to pass the call site as a parameter through partial application, so that it can be updated with a new target once it has been found and cached.

Megamorphic call sites. Method dispatch arrange trees of `guardWithTest` combinators up to a depth of 5. Once this limit has been reached, a method call site turns into a *megamorphic* state and changes to a new dispatch strategy. Instead of composing trees on cache invalidation, it caches method handles in a `HashMap`. Method dispatch is then performed by a composition of `exactInvoker` and `foldArguments` combinators. The later performs a lookup into the cache `HashMap`, giving a method handle to the former. The lookup populates the cache as new method handles need to be discovered. This ensures an amortized dispatch for megamorphic call sites.

Dynamic objects. Method dispatch call sites handle dynamic objects as a special case. It still employs inline-caches, with a `guardWithTest` based on the receiver instance, as caching can only occur on a per-instance basis. Each dynamic object maintains a set of `SwitchPoint` instances that are given to the consuming call sites. These switch points provide a method handle to the dynamic object method in use, be it to access a value property or to dispatch to a function. Shall the dynamic object property be changed, all switch points become immediately invalidated. Each fallback method handle points to the consuming callsite general-purpose fallback, prompting a new lookup to be made on subsequent dispatches.

`null`-safe method invocations. Method invocations using the `'?:'` operator use the strategies above for polymorphic, megamorphic and dynamic objects. It adds a frontal `catchException` combinator on `NullPointerException`, with a handler method handle that simply discards all arguments and return null. The advantage of this solution is that non-`null` method invocations execute in a *fast path*, while occasional `null` invocations will actually produce a `NullPointerException` that will be trapped. Unless `null` receiver objects are frequent, our experiments show that this yields better performance and a simpler design than introducing a further check in a `guardWithTest` guard.

Closures invocations. Closures can be referenced by both `let` and `var` references. Their invocation is done in 2-steps: a first call site provides the target method handle which is being loaded from the current reference value. The next call sites is an inline-cache, again built using `guardWithTest` where the guard is on the method handle instance and the target an invoker for the call site type. This provides an efficient dispatch mechanism, as the first call site looks into a local variable, while the next one is an inline-cache.

4. Early evaluation

Benchmarking a language on the JVM remains difficult. The adaptive nature of the runtime makes it hard to predict which optimizations will be effective, yielding results with deviations over runs. Also, a benchmark rarely matches the conditions of a real application, including the impact of input-output operations, workload patterns and code base sizes. Nevertheless, micro-benchmarks are useful to evaluate specific portions of a language runtime.

The Golo source code [2] contains a `benchmark` folder. It is organized as a Maven project where benchmarks are written using JUnitBenchmarks. Each test fixture is run in a fresh JVM instance, with sufficient warm-up and execution rounds to have low standard deviation in the measures (worst case observed: 0.15s, most cases are below 0.01s). Table 1 shows the micro-benchmark results, run on a 2.66 GHz Intel Core 2 Duo MacBook Pro with 8 GB of RAM memory with Mac OS X 10.8.3. The Java Runtime Environment is the Oracle JDK build 1.7.0_17-b02, running HotSpot 64 bits build 23.7-b01. We obtained similar performance figures using custom builds of OpenJDK 8. Complete results including measured standard deviations can be consulted at `https://gist.github.com/jponge/5965512`.

The Fibonacci test measures the time to compute $fib(40)$ using the naive recursive definition of the function. The next step measures the time to execute a chain of filter, map and reduce operations over a large list of integers. The monomorphic, trimorphic and megamorphic stress method call dispatches in presence of respectively a single, three and many (10) receivers types. Finally, the last test measures the performance of calling closures. Different languages are being used depending on the test cases: Java (version 7), Groovy (version 2.1.3), JRuby (version 1.7.3) and Clojure (version 1.5.1). We did our best to stick to the idioms of each language while providing comparable code constructs. Especially, we did not use typing or type hints when offered by the language, as Golo can only be fairly compared to dynamically-typed code. Finally, we ensured that each language runtime would use `invokedynamic` support, if available, which is the case of Groovy and JRuby.

Fibonacci 40	Java	1.00 s
	Java (boxing)	1.99 s
	Golo	2.92 s
	Clojure	8.66 s
	JRuby	16.42 s
	Groovy	33.56 s
Filter, map, reduce	Golo	0.16 s
	Groovy	0.71 s
	Clojure	0.73 s
	Clojure (lazy collections)	1.12 s
	JRuby	1.33 s
Monomorphic dispatch	Java	0.30 s
	Golo	0.31 s
	Golo (`?:`)	0.31 s
	Golo (`?:` $\approx 50\%$ `null` receivers)	0.39 s
	Groovy	0.54 s
	JRuby	1.25 s
Trimorphic dispatch	Java	0.19 s
	Golo	0.20 s
	Golo (`?:` $\approx 21\%$ `null` receivers)	0.31 s
	Groovy	0.71 s
	JRuby	1.23 s
Megamorphic dispatch	Java	0.10 s
	Golo	0.21 s
	Groovy	1.05 s
	JRuby	1.21 s
Calling closures	Golo	0.18 s
	Java (anonymous inner-classes)	0.24 s
	Groovy	1.16 s
	JRuby	3.76 s

Table 1. Micro-benchmark results (times are in seconds).

The results show that the Golo runtime performs well on those benchmarks. This can be explained due to the straightforward usage of `invokedynamic` and the design of the language around it. Porting an existing language to the JVM, or coping with a long development history when adapting it to `invokedynamic` necessarily makes the effort more difficult. Golo is efficient on method and closure dispatch. The result gap for Golo is bigger on $fib(40)$ than some other tests when compared to Java. Indeed, Golo suffers from primitive boxing and unboxing. The evaluation of arithmetic expressions is near-sighted, and boxing / unboxing happens for every operator. Expressions such as $fib(n-1) + fib(n-2)$ could be better handled to reduce boxing effects.

Finally, we could expand the micro-benchmark sets to other dynamic languages. It would be especially interesting to evaluate Nashorn [6], but since it is still in development and depends on Java 8 which has yet to be finalized, it is better waiting for stabilization.

5. Related work

The Java Virtual Machine has an open specification [14] with an ecosystem of languages that goes well beyond Java. Extensive research efforts have been put into the design of efficient adaptive runtime strategies for the JVM to make it suitable for *client* and long-running *server* applications [9, 13, 15].

It is commonly reported that over 200 languages have been designed for the JVM, or ported to it, but more realistically only few of them have been popular. Dynamic languages are popular choices on the JVM, including Groovy [3], JRuby [4], Jython [5] or Clojure [1].

The JVM had an initial bias towards statically typed languages [14], which made it hard for language implementers to design

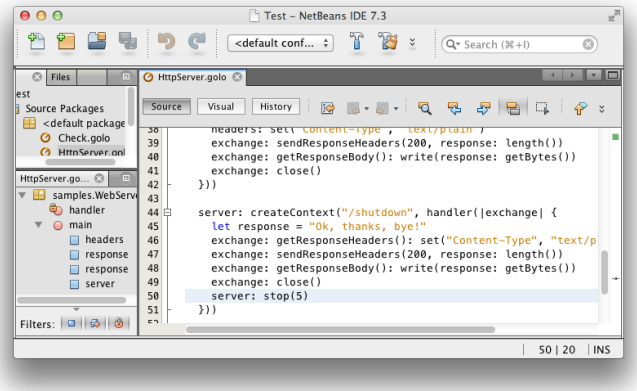

Figure 1. The Golo Netbeans IDE.

runtimes that would be subject to efficient JIT optimizations. The situation changed with the JSR 292 and the release of Java 7, introducing a new *opcode* called `invokedynamic` and a support API [17, 18].

As Golo shows, `invokedynamic` allows the design of simple language runtimes while providing hints for the JVM to perform efficient optimizations. Some of the existing dynamic languages for the JVM are already porting their runtimes to take advantage of `invokedynamic` [3, 4]. Clean-room implementations of runtimes for existing languages are starting to appear such as Nashorn for JavaScript [6].

It is especially interesting to note that `invokedynamic` allows implementing adaptive runtime techniques that the JVM itself employs, while staying in a high-level language (Java). Golo makes heavy usage of inline-caches, which date back to the works on SmallTalk and Self [10, 11].

Last but not least, `invokedynamic` is interesting for other purposes than dynamic languages. They are poised to serve in the implementations of *lambdas* for Java 8. Derivative usage are starting to appear in [7] or the dynamic software modification agent JooFlux [16].

6. Conclusion and perspectives

This paper introduced Golo, a new dynamic programming language for the JVM. With simple semantics and a lightweight runtime based on JSR 292, it is an interesting language for research on languages, middleware and runtimes. It can also serve in education and be useful to the polyglot application programmer. It can call Java, be called from Java, or more generally from any JVM language. Last but not least, its runtime performance compares favorably against Java and the other dynamic languages for the JVM.

Community. Having been released to the public at the end of March 2013, Golo remains a young language with a nascent community. Still, it has already received external contributions. Some are being listed at `http://k33g.github.io/nano.golo/`, which is a website maintained by a Golo enthusiast. Contributions include experiments of Golo as a domain-specific language for testing, web frameworks or IDE-support. Figure 1 shows a community-contributed support of Golo in the Netbeans IDE, with code completion, error reporting, structure overview and run integration. A similar project is in the works for the Eclipse platform.

Future works. The Golo language will progressively get new features. Of particular interest at the language level, we would like to support class definitions, value class definitions and collection literals. We also plan to add support for API documentation blocks written in Markdown format. This requires the implementation of a text template system that we would like to design after *ERB* in Ruby [4]. Golo already provides a *workers* API for concurrent message-passing, and we would like to investigate how it can be further improved to facilitate concurrent programming.

As developers discover Golo, we expect it to be tested on larger code bases. This will be interesting to better reflect on the runtime performance than just look at micro-benchmarks. Also, the impact of a large number of `invokedynamic` call sites has to be investigated with respect to performance and memory, too.

The authors will experiment with research works where Golo will serve as a basis for language extensions and a simple runtime to modify to support, say, isolation, security or ambient computing. Finally, we invite the larger researchers and practitioners community to experiment with Golo, report suggestions, signal potential issues, and contribute to its development.

Acknowledgments

This work is partially supported by the *ARC6* program of *Région Rhône-Alpes*. We would like to thank: the initial testers before Golo was made public, Philippe Charrière for his enthusiasm, Serli and Ninja Squad for their support, and Rémi Forax for the thoughtful technical discussions.

References

[1] URL `http://clojure.org/`.

[2] URL `http://golo-lang.org/`.

[3] URL `http://groovy.codehaus.org/`.

[4] URL `http://jruby.org/`.

[5] URL `http://www.jython.org/`.

[6] URL `http://openjdk.java.net/projects/nashorn/`.

[7] M. Appeltauer, M. Haupt, and R. Hirschfeld. Layered method dispatch with invokedynamic: an implementation study. In *Proc. of COP'10*. ACM, 2010.

[8] E. Bruneton, R. Lenglet, and T. Coupaye. Asm: A code manipulation tool to implement adaptable systems. In *Adaptable and extensible component systems*, 2002.

[9] C. Häubl and H. Mössenböck. Trace-based compilation for the Java HotSpot virtual machine. In *Proc. of PPPJ'11*, pages 129–138. ACM, 2011.

[10] U. Hölzle and D. Ungar. Optimizing dynamically-dispatched calls with run-time type feedback. In *Proc. of PLDI'94*, pages 326–336. ACM, 1994.

[11] U. Hölzle, C. Chambers, and D. Ungar. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In *Proc. of ECOOP'91*, pages 21–38. Springer-Verlag, 1991.

[12] V. Kodaganallur. Incorporating language processing into java applications: A javacc tutorial. *IEEE Software*, 21(4):70–77, 2004.

[13] T. Kotzmann, C. Wimmer, H. Mössenböck, T. Rodriguez, K. Russell, and D. Cox. Design of the Java HotSpot client compiler for Java 6. *ACM Trans. Archit. Code Optim.*, 5(1):7:1–7:32, May 2008.

[14] T. Lindholm and F. Yellin. *Java Virtual Machine Specification*. Addison-Wesley, 2nd edition, 1999.

[15] M. Paleczny, C. Vick, and C. Click. The Java HotspotTM server compiler. In *Proc. of JVM'01*. USENIX, 2001.

[16] J. Ponge and F. Le Mouël. JooFlux: Hijacking Java 7 InvokeDynamic To Support Live Code Modifications. Research report, INSA-Lyon, CITI-INRIA, Oct. 2012.

[17] J. R. Rose. Bytecodes meet combinators: invokedynamic on the JVM. In *Proc. of VMIL'09*. ACM, 2009.

[18] C. Thalinger and J. Rose. Optimizing invokedynamic. In *Proc. of PPPJ'10*, pages 1–9. ACM, 2010.