Opportunities for a Truffle-based Golo Interpreter

Julien Ponge[1], Frédéric Le Mouël[1], Nicolas Stouls[1], and Yannick Loiseau[2]

[1] Université de Lyon
INSA-Lyon, CITI-INRIA F-69621, Villeurbanne, France
`firstname.lastname@insa-lyon.fr`

[2] Clermont Université, Université Blaise Pascal, LIMOS, BP 10448,
F-63000, Clermont-Ferrand, France
`firstname.lastname@univ-bpclermont.fr`

April 2015

Abstract

Golo is a simple dynamically-typed language for the Java Virtual Machine. Initially implemented as a ahead-of-time compiler to JVM bytecode, it leverages `invokedynamic` and JSR 292 method handles to implement a reasonably efficient runtime. Truffle is emerging as a framework for building interpreters for JVM languages with self-specializing AST nodes. Combined with the Graal compiler, Truffle offers a simple path towards writing efficient interpreters while keeping the engineering efforts balanced. The Golo project is interested in experimenting with a Truffle interpreter in the future, as it would provides interesting comparison elements between invokedynamic *versus* Truffle for building a language runtime.

1 Introduction

Golo is a simple dynamically-typed language for the Java Virtual Machine [1]. Initially designed as an experiment around the capabilities of the new `invokedynamic` JVM instruction that appeared in Java SE 7 [2], it has since emerged as a language supported by a small community that goes beyond the bounds of academia. Applications have been found in *Internet of Things* (IoT) settings, and we consider Golo to be small enough to be used for language and runtime experiments by researchers, students and hobbyists. This claim is supported by examples such as ConGolo, a derivative experiment for contextual programming[1], and the community projects[2]. Golo is currently being proposed for incubation at the Eclipse Foundation in the hope of finding new opportunities and continuing the development at a vendor-neutral foundation[3].

Figure 1 provides a sample Golo program. It computes several Fibonacci numbers with the naive recursive definition of the `fib` function. It takes advantage of regular Java executors and Golo APIs for promises and futures [3] to perform the computations on 2 worker threads, and collect the results through reduction of futures.

Briefly, the main characteristics of the Golo programming language are the following:

- dynamic typing using Java types,
- higher-order functions and binding to Java single-method and functional interfaces,
- ability to *augment* existing types (including from JVM languages) with new methods,

[1]See `https://github.com/dynamid/contextual-golo-lang`.
[2]The *kiss* web framework is a good example: `https://github.com/k33g/kiss`.
[3]See `https://projects.eclipse.org/proposals/golo`.

```
module samples.Concurrency

import java.util.concurrent
import gololang.Async

local function fib = |n| {
  if n <= 1 {
    return n
  } else {
    return fib(n - 1) + fib(n - 2)
  }
}

function main = |args| {
  let executor = Executors.newFixedThreadPool(2)
  let results = [30, 34, 35, 38, 39, 40, 41, 42]:
    map(|n| -> executor: enqueue(-> fib(n)):
    map(|res| -> [n, res]))
  reduce(results, "", |acc, next| ->
      acc + next: get(0) + " -> " + next: get(1) + "\n"
  ):
    onSet(|s| -> println("Results:\n" + s)):
    onFail(|e| -> e: printStackTrace())
  executor: shutdown()
  executor: awaitTermination(120_L, TimeUnit.SECONDS())
}

# === Prints the following ===
# Results:
# 30 -> 832040
# 34 -> 5702887
# 35 -> 9227465
# 38 -> 39088169
# 39 -> 63245986
# 40 -> 102334155
# 41 -> 165580141
# 42 -> 267914296
```

Figure 1: Computing Fibonacci numbers in Golo with concurrent and asynchronous APIs.

- tuples and structures (augmenting the later is reminiscent of Go-style objects [4]),
- dynamic objects with instance-level definitions,
- Python-style decorators (i.e., higher-order function-based).

Unlike many other JVM languages such as JRuby, Jython or Nashorn, Golo is not a port of an existing language to the JVM and invokedynamic. This is interesting, as Golo was designed *around* the capabilities of invokedynamic, which gives a different perspective on the design of a invokedynamic-based runtime.

2 Ahead-of-time compilation based on JSR 292

Golo uses ahead-of-time bytecode generation rather than interpretation. The grammar of Golo is written using the $LL(k)$ JJTree / JavaCC parser generator [5], 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 several phases to check for undeclared references, expand lambda functions / closures to anonymous functions, and ultimately generated JVM bytecode with the popular ASM library [6].

Stable bytecode, adaptive runtime dispatch. The compiler generates a largely *untyped* bytecode. Most references are on the `java.lang.Object` type, with some peculiar portions of the bytecode doing cast checks (e.g., branch conditions require refining to `java.lang.Boolean` and unboxing the primitive boolean value). The generated bytecode remains stable at runtime, unlike speculations and invalidations as found in Nashorn to try to take advantage of primitive types when possible.

As most call sites (including arithmetic operators) are based on invokedynamic, the runtime adapts the dispatch

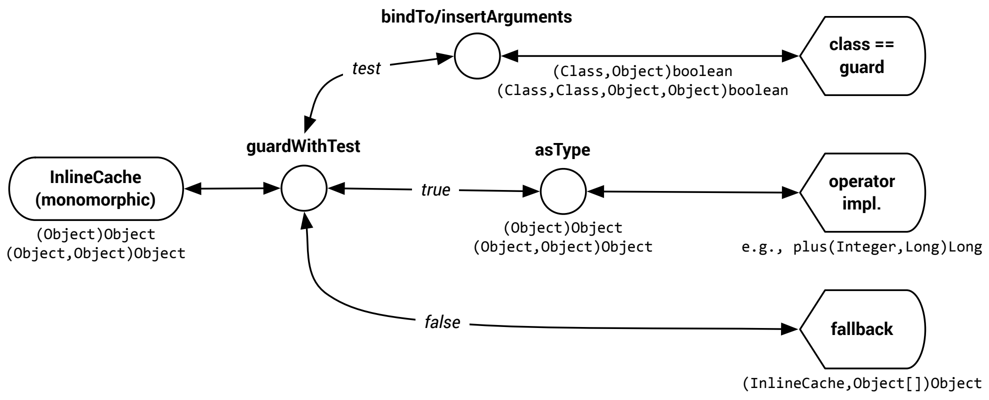

Figure 2: Operator monomorphic inline-cache based on method handles.

targets through evolving *method handle* chains, based on types observed at runtime. Figure 2 gives an example: operators use a monomorphic inline-cache construct [7]. The construction relies on a *guarded* combinator that dispatches to the right target as long as type remain stable (e.g., `plus(Integer, Long) Long` for `10 + 10_L`). The fallback branch points to a handler that dynamically finds a new target based on the observed types, and overwrites the call site method handle dispatch chain with the new one.

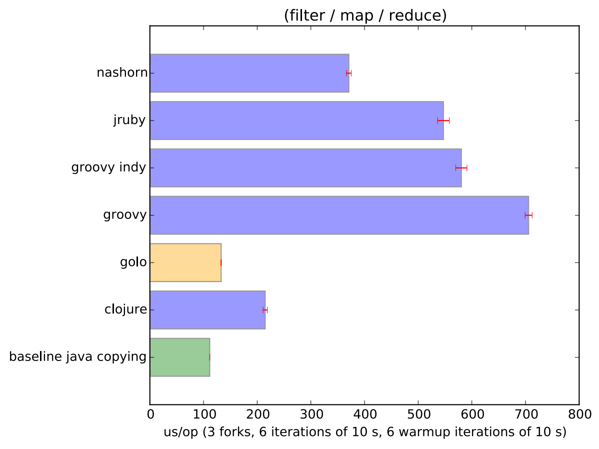

Figure 3: Filter / Map / Reduce micro-benchmark.

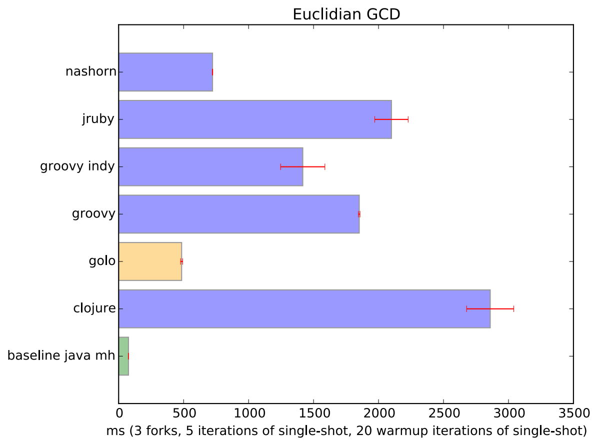

Figure 4: Greatest common divisor micro-benchmark.

Performance considerations. In general, Golo exhibits good performance on function and method dispatch[4]. Figure 3 shows the results of a micro-benchmark based on applying the usual *filter*, *map* and *reduce* operations on collections. Golo is practically as fast as a baseline in Java where the operations are implemented using collection copies[5].

[4]See `https://github.com/golo-lang/golo-jmh-benchmarks` for a collection of micro-benchmarks.

[5]This micro-benchmark was written while Golo was still compatible with Java SE 7, hence it would be interesting to compare with Java SE 8 streams.

Figure 4 shows a GCD micro-benchmark. While performance remains good compared to other dynamic languages, it highlights the performance bottleneck due to boxing of primitive types, which is also further confirmed by further nano-benchmarks that we have. We are planning to explore ways to be clever than we are at the moment with respect to arithmetic operations.

3 Perspectives with Truffle

Writing an interpreter for Golo based on Truffle [8] is interesting for comparing the effectiveness of invokedynamic *versus* Truffle to implement common language runtime patterns (e.g., arithmetic operations or inline-caches). In terms of performance, the following points are of comparison interest:

1. functions and methods dispatch,
2. arithmetic operations (Truffle node specialisation can potentially eliminate some boxings),
3. dispatch in Golo dynamic objects (Truffle proposes a *Shapes* abstraction),
4. statistical optimizations for application profiling (Truffle exposes node counters that could be use to mine application behavior and dynamically activate relevant optimizations).

We are looking forward to experimentions with Truffle in the near future, and have elements of comparisons in the challenges of designing programming language on the JVM.

References

[1] Julien Ponge, Frédéric Le Mouël, and Nicolas Stouls. Golo, a dynamic, light and efficient language for post-invokedynamic jvm. In *Proc. of PPPJ 2013*, pages 153–158, New York, NY, USA, 2013. ACM.

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

[3] B. Liskov and L. Shrira. Promises: Linguistic support for efficient asynchronous procedure calls in distributed systems. In *Proc. of PLDI'88*, pages 260–267, New York, NY, USA, 1988. ACM.

[4] Rob Pike. Go at Google. In *Proc. of Splash'12*, pages 5–6, New York, NY, USA, 2012. ACM.

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

[6] Eric Bruneton, Romain Lenglet, and Thierry Coupaye. Asm: A code manipulation tool to implement adaptable systems. In *Adaptable and extensible component systems*, 2002.

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

[8] Stefan Marr, Tobias Pape, and Wolfgang De Meuter. Are We There Yet? Simple Language-Implementation Techniques for the 21st Century. *IEEE Software*, 31(5):60–67, September 2014.