Nomen: A Dynamically Typed OO Programming Language, Transpiled to Java

Tijs van der Storm
Centrum Wiskunde & Informatica (CWI)
University of Groningen (RUG)
storm@cwi.nl

Introduction

Nomen is an experimental, dynamically typed OO programming language which compiles to Java source code. The translation to Java is transparent: a class in Nomen is a class in Java, a method in Nomen is a method in Java, etc. The generated code is thus relatively idiomatic (allowing the JVM to optimize method dispatch), and easy to map back to Nomen code during debugging.

Furthermore, the compilation scheme of Nomen supports separate compilation of Nomen modules, does not require any casts at runtime, and supports dynamic features such as Ruby's method_missing. This is achieved using a simple module system for Nomen, and a novel application of recursive F-bounds and Java 8 default methods in the generated code.

The ongoing implementation of Nomen can be found here: https://github.com/cwi-swat/nomen.

The Design of Nomen

Nomen is designed as a language for experimenting with IDE support generation using the Rascal language workbench [4, 7]. As such, it can be considered an extended case-study in language engineering, exercising techniques for specifying, testing, desugaring, compiling, and deploying language implementations, including editor services such as syntax highlighting, error marking, code completion, outlining, incremental compilation, live programming, debugging, and so on [2]. Furthermore, Nomen is designed to as a future test-bed for experimenting with language embedding approaches based on syntactic language virtualization [1, 5, 8].

Nomen is a simple, object-oriented, dynamically typed language inspired by Ruby. It features single inheritance, a module system for namespacing (a la Python), closures as objects (with instance_eval a la Ruby), and anonymous classes (a la Java). A simple example program is shown in Figure 1.

The snippet shows a module Space, containing two classes, Spacecraft and Orbiter. Both classes have an initialize method to initialize fields (prefixed with @). The describe method prints out a simple description. The puts method is inherited from the root object which all objects inherit from.

Some principles that have informed the design of Nomen:

```
module Space

class Spacecraft
  def initialize(name, launchDate):
    @name = name;
    @launchDate = launchDate;

  def describe:
    puts("Spacecraft: " + name);
    if @launchDate then
      puts("Launched at " + @launchDate);
    end

class Orbiter: Spacecraft
  def initialize(name, launchDate, altitude):
     super.initialize(name, launchDate);
     @altitude = altitude
```

Figure 1. Example Nomen Module

- *Dynamic where desired, static where needed.* Methods are alway late bound; – classes, modules, and local variables are statically resolved. Calling a method never causes a static error, but referencing undefined classes or modules does. Nomen supports method_missing to intercept method invocations of undefined methods, but does not support dynamically modifying class definitions ("monkey patching").
- *Different things have different syntax.* This applies mostly to names: static names are capitalized, local variables and method parameters start with a lowercase letter, fields start with @, and method calls always have parenthesized arguments, an explicit receiver, or a trailing statement (see below).

One important design goal has been to support a very flexible and rich method call syntax to support DSL embedding and fluent interface idioms. First of all, standard prefix and infix operators are implemented as methods. Second, method calls can be suffixed with a trailing statement which will be implicitly lifted to a closure (if it isn't already a closure), and

```
def menu(menu):
  echo(menu.title);
  ul for k in menu.kids do
    item(kid)
  end
end
```

```
def item(item)
  if item.kids then
    li menu(item)
  else
    li a(item.link) item.title
  end
end
```

Figure 2. Rendering recursive menus to HTML using statement chaining

passed in as the last argument. For instance, the statement f(x) y = 3; is equivalent to f(x, {y = 3}), where curly braces indicate explicit closure creation.

Figure 2 shows two methods exploiting this kind of chaining feature to render recursive menu structures to HTML. The code assumes that methods for rendering HTML elements (e.g., ul, li, etc.) and outputting text (echo) are in scope. Statement chaining happens when invoking ul in the menu method, and li in the item method. The statement chaining captures the nesting of HTML elements. For instance, in the **else** branch of item, the anchor a is nested within the li element and the item's title will be the content of the anchor element. This form of chaining provides a flexible syntax to express builder patterns.

Currently, however, Nomen's most distinctive feature is how it is transpiled to Java, which I describe next.

Implementation

[...] the so-called untyped (that is "dynamically typed") languages are, in fact, unityped. (Dana Scott) [3]

The compilation of dynamically typed programming languages to the JVM has been an ongoing challenge. Current implementations of dynamically typed programming languages, such as JRuby, resort to VM level techniques based on invokedynamic and method handles, or use runtime partial evaluation of interpreters in combination with specific VM support [6]. Compiling to source code has been generally problematic, since Java requires a type for invoking methods (even reflectively).

At one end, there is the strategy of generating a single, "maximal" interface declaring all method patterns occuring in the source code; this type will then be the declared type of all method parameters and return values at runtime. Unfortunately, this breaks separate compilation. At the other end of the spectrum, the compiler would generate a separate, "minimal" interface for each individual method pattern. The declared type of values will then be Object, but receivers need to be cast to the interface corresponding to the call at every call site. Furthermore, this strategy may lead to class file bloat.

Next I describe a middle ground between these two extremes: instead of *generating* a maximal interface (supporting "all possible methods") from scratch, we will construct

```
module A        interface A<O extends A<O>>
                  extends Kernel<O> {

                  default O foo() { return missing("foo"); }

class Foo         O A$Foo(); // abstract constructor
                  abstract class Foo<O extends A<O>>
                    extends Kernel.Obj<O> implements A<O> {
  def foo:          public O foo() { ... }
    ...           }
                }
```

```
module B        interface B<O extends B<O>>
import A          extends A<O> {

                  default O baz() { return missing("baz"); }

class Bar: Foo    O B$Bar(); // abstract constructor
                  abstract class Bar<O extends B<O>>
                   extends A.Foo<O> implements B<O> {
  def foo:           public O foo() {
    baz();             baz();
    new Foo()          return A$Foo();
                     }
                  }
                }
```

Figure 3. Translating Nomen modules to Java

it incrementally using recursive F-bounds. This compilation scheme is illustrated in Figure 3. The left shows two Nomen modules (A and B); the corresponding Java code is shown on the right. Every module is mapped to a generic interface, where the type parameter is bounded by itself. If a module has no imports of its own (e.g., A), the generated interface extends the built-in Kernel module containing standard classes for numbers, boolean, strings etc. Otherwise, (e.g., as in B), each import will be represented by an extends-clause. Cyclic imports are disallowed, and as such imports map directly to Java's multiple interface inheritance feature.

A module interface declares default methods for every method pattern that occurs in the module (either as a definition or in a call), delegating to the missing method (declared in Kernel). For instance, interface B provides a default implementation for baz, even it is not defined anywhere in the Nomen code.

Every Nomen class is mapped to two Java declarations. First, an abstract constructor method is generated, using a fully qualified name (e.g., A$Foo). Second, the class itself will be represented by a generic, abstract class, implementing the current module interface (e.g., Foo). The super class will be the corresponding abstract class (if any), or Kernel.Obj otherwise.

The methods in a class compile to Java methods, where every return type and argument type is the generic type parameter representing the carrier type O. Statements and expressions are compiled in a straightforward way. Object construction, however, does not map directly to object construction in Java, but delegates to the abstract constructor methods (e.g., AFoo, BBar).

Although the Java code of Figure 3 can be compiled, it is not yet executable: no objects are ever created, and there is no main method to provide an entry point. At this point the abstract classes generated from the Nomen classes are literally incomplete. They are "completed" in the context of a "main" module, which provides the entry to program execution. This is handled by additionally generating code for modules which contain top-level statements (i.e. without enclosing methods). For instance, for a main module M (importing B):

- Tie the knot: define a local **interface** Self **extends** M<Self>. As a result, Self declares all method patterns occuring in M, as well as those occurring in (transitively) imported modules.
- For every class reachable from M through the import graph, define an empty concrete class extending the corresponding abstract class, implementing Self. For instance, the interface corresponding to M will contain a **class** B$Bar **extends** B.Bar<Self> **implements** Self.
- Provide implementations for the abstract constructors returning instances of the concrete classes of the previous bullet. For instance, **default** O B$Bar() {**return** (O)**new** B$Bar();}.This is the only cast that is generated, and it's a vacuous one at that, because of Java's type erasure.

The main code itself is lifted into a synthetic Main class, which contains the standard Java static main entry point.

Defining the concrete classes over Self will effectively bind all "O" type parameters (in this module and imported ones), to Self. All constructed objects – dynamically bound through the constructor methods, but now implemented to return instances of concrete classes – will have all methods available defined or used, in this module or in any of the imported ones. Since Java inheritance predicates that class extension has precedence over inheritance of default methods, the actual methods defined in the Nomen classes will always take precedence over the stub methods defined in the module interfaces.

Conclusion and Outlook Nomen is a dynamically typed, OO programming languages, designed as an extended case study in language engineering. Its design is characterized by a static module system, flexible method invocation and statement syntax, and support for method_missing DSL embedding idioms. Nomen is transpiled to Java source code using a novel scheme based on recursive F-bounds in combination with Java 8 default methods. This scheme allows compilation of dynamically typed OO code without relying on casts, reflection, or VM level techniques, but without sacrificing separate compilation.

Nomen is by no means finished. The current prototype supports static checking of Nomen source code and incremental compilation from within an Eclipse IDE developed using the Rascal language workbench [4]. Further work in the near future includes defining the precise static semantics of Nomen and exploring the performance of Nomen's compilation scheme compared to other schemes.

Acknowledgments Thanks to the anonymous reviewers, the attendees of the IFIP Working Group on Language Design meeting in Lausanne, and James Noble for providing constructive feedback on the design and implementation of Nomen.

References

[1] A. Biboudis, P. Inostroza, and T. van der Storm. Recaf: Java dialects as libraries. In *GPCE*. ACM, 2016.

[2] S. Erdweg, T. van der Storm, M. Völter, L. Tratt, R. Bosman, W. R. Cook, A. Gerritsen, A. Hulshout, S. Kelly, A. Loh, G. Konat, P. J. Molina, M. Palatnik, R. Pohjonen, E. Schindler, K. Schindler, R. Solmi, V. Vergu, E. Visser, K. van der Vlist, G. Wachsmuth, and J. van der Woning. Evaluating and comparing language workbenches: Existing results and benchmarks for the future. *Computer Languages, Systems & Structures*, 44, Part A:24 – 47, 2015.

[3] B. Harper. Dynamic languages are static languages. Online, March 2011. `https://existentialtype.wordpress.com/2011/03/19/dynamic-languages-are-static-languages/`.

[4] P. Klint, T. van der Storm, and J. J. Vinju. RASCAL: A domain specific language for source code analysis and manipulation. In *SCAM*, pages 168–177. IEEE, 2009.

[5] A. Loh, T. van der Storm, and W. R. Cook. Managed data: modular strategies for data abstraction. In *Onward!*, pages 179–194. ACM, 2012.

[6] C. Seaton. *Specialising Dynamic Techniques for Implementing The Ruby Programming Language*. PhD thesis, University of Manchester, School of Computer Science, 2015.

[7] T. van der Storm. The Rascal Language Workbench. CWI Technical Report SEN-1111, CWI, 2011.

[8] T. Zacharopoulos, P. Inostroza, and T. van der Storm. Extensible modeling with managed data in Java. In *GPCE*. ACM, 2016.