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

Tijs van der Storm

CWI
Centrum Wiskunde & Informatica

university of groningen

Nomen

- Dynamically typed OO language (think: Ruby)
- Single inheritance
- Support for **method_missing** like meta programming
- *Transpiles* to Java

```
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
```

```
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
```

Motivation

- Case-study in language engineering: Rascal language workbench, debugging etc.

rascal-mpl.org

- Experimenting with language virtualization based on flexible and generic syntax
(cf. Biboudis et al., GPCE'16)

Call For Papers and Participation

The Minus-Oneth Workshop on New Object Oriented Languages (NOOL '16)

30 October, 2016, Amsterdam, Netherlands

NOOL is a new independent workshop that brings together users and implementors of object oriented systems. Through tutorials, papers, panel discussions and workshops, as well as demonstrations, exhibits and videotapes, NOOL will provide a forum for sharing experience and knowledge among experts and novices alike. We invite technical papers, case studies, and surveys in the following areas:

Theory: Including core definition of object oriented programming, semantic models and methodology.

Languages: Noo object oriented languages, extensions to conventional languages, and even nooer languages.

Implementation: Including architectural support, compilation and interpretation, and special techniques.

Tools and Environments: Including user interfaces, utilities and operating system support.

Applications: Commercial, educational, and scientific applications that exploit object oriented programming.

Related Work: The object oriented paradigm in other fields such as databases and operating systems.

Papers on other relevant topics are also welcome, as are proposals for workshops and panel discussions.

All papers will be refereed prior to selection and inclusion in the workshop proceedings. Technical papers will be selected on the basis of originality and contribution to the state of the art of design, implementation, methodology, or practice. Survey papers will be selected on the basis of how well they crystallize and integrate, in a way not previously presented, knowledge about one or more aspects of the field.

Papers must be submitted in English, and should be no longer than 2 double column pages. The cover page should include a title, an abstract of not more than 100 words, and author's name, affiliation, address and phone number. Five copies must be received by the Program Chairs at the address below, no later than 1 Sep 2016. Authors will be

Nomen:
No Other Motivation Except Nomen

Recursive bounds!

- Nomen = No Other Motivation Except Nomen
- Nomen<X extends Nomen<X>>
- Which actually turns out to be the key to the Nomen's transpilation scheme!

Tijs van der Storm
@tvdstorm

Ok, I'm going to use "transpile" in an academic context. But I won't ever use "isomorphic web development", promise!

LIKES
7

11:32 AM - 13 Sep 2016

   7  

Eric Van Wyk
@ericvanwyk

Following

@inkytonik @tvdstorm If you use 'transpile' I will stop following you on Twitter. There must be a way I can block that word from my feed...

RETWEET	LIKES
1	4

2:27 PM - 13 Sep 2016

1 4

Ron Garcia
@rg9119

Follow

@tvdstorm @inkytonik I'm with @ericvanwyk on this. "Transpiler" gives undue mystique to "compiler", which = source to source translator.

RETWEETS	LIKES
7	11

8:03 PM - 13 Sep 2016

7 11

CompilerConstruction
@IN4303

Follow

CC students: I'm adding this twitter discussion to the course material; please study and expect exam questions

> **Tijs van der Storm** @tvdstorm
> Ok, I'm going to use "transpile" in an academic context. But I won't ever use "isomorphic web development", promise!

RETWEET 1 LIKES 3

8:33 AM - 18 Sep 2016

1 3

Transpilation

- “Transparent compilation” to source code in a different language
  - Case in point: CoffeeScript
- Abstractions of the source language are “recognizable” in the generated code
- Why? Simple to implement; debugging; be idiomatic to target compiler.

How to compile a dynamically typed OO lang to Java source code

Maximal style

- Maximal: generate one interface, which supports all method patterns occurring in the source
- Have all classes implement that interface
- The declared type of all values will be that interface
- => does not support separate compilation

Minimal style

- Minimal: generate one interface *per method*
- Classes implement those method interfaces for which they have implementations
- The declared type of all values will be *Object*
- => lots of casting, class bloat

Nomen

- Recursive bounds to “incrementally” build up the *maximal* interface

- So get benefits of maximal, but without losing separate compilation.

- A module system is required, though, to partition the “payload” of the generated interface

module A

module A

interface A<O **extends** A<O>>
extends Kernel<O> {

}

module A

class Foo

  def foo:

    ...

interface A<O **extends** A<O>>
  extends Kernel<O> {

}

```
module A
class Foo
  def foo:
    ...
```

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

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

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

```
module B
import A


class Bar: Foo

  def foo:
    baz();
    new Foo()
```

module B
import A

interface B<O **extends** B<O>>
  extends A<O> {

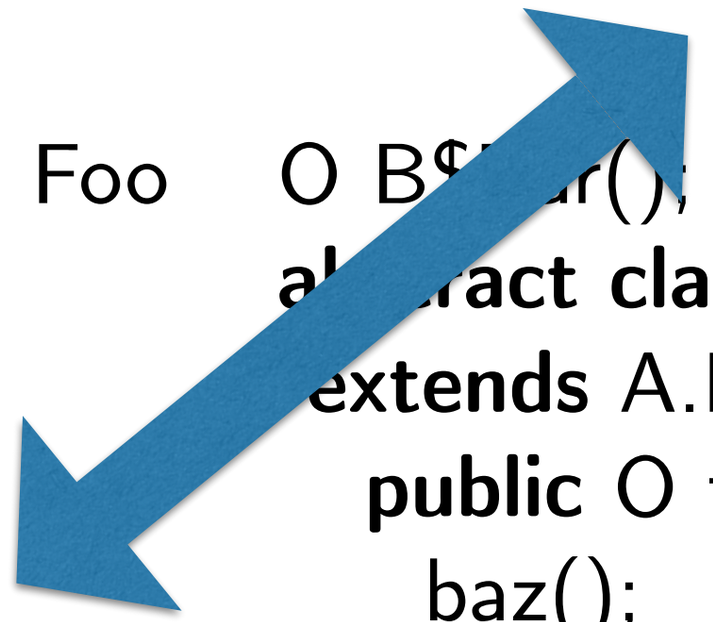
class Bar: Foo

  def foo:
    baz();
    new Foo()

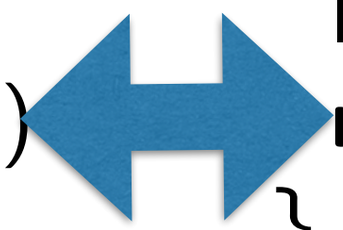
}

```
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();
                              }
                            }
                          }
```

```
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();
                              }
                            }
                          }
```



```
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();
                              }
                            }
                          }
```



- Module => generic interface
    - default methods for all method patterns in module
- Module import => extends clause

- Class => generic abstract class implementing module interface
- Declared type of values is module parameter everywhere

For a “main” module M

- Tie the knot:

```
interface Self extends M<Self>
```

- Concretize the Nomen classes, implementing Self

```
class B$Bar extends B.Bar<Self> implements Self
```

- Implement abstract constructors

```
default O B$Bar() {return (O)new B$Bar();}
```

Summary

- This is the *maximal* style: Self will support all methods
- But incrementally through the recursive bounds (hence separate compilation)
- Java's default methods will provide default behavior if a Nomen class doesn't implement the method.
- In Nomen they delegate to **method_missing**

Conclusion

- Demo