Lambda Doge: A digital canine language

Joshua Suereth

March 11, 2016

Abstract

Lambda Doge is the first programming language designed specifically for canine implementers. By applying the known simplicity of lambda calculus to the known structure of canine utterances, we are able to create a programming environment that is both elegant and turing complete.

1. Motivation

According to a 2012 report by the world health organization[1]:

> … estimates that there are around 78 million owned dogs in Africa, but that there may also be in excess of 70 million unowned, stray dogs as well.

According to a random quora infographic, there is an estimated 17 million programmers in the world[2].

If we had the opportunity to teach all stray dogs in Africa how to program, then this would have profound impacts, including:

- Less strain on the economy from freeloading pooches
- New "outsourcing tech mecha" in africa from low-cost high-quality canine coders
- Estimated 400% growth in computer book sales

So, there is a lot of reason to design a programming language for dogs.

2. Language design

Designing a language for dogs is quite hard. First we decided to take some of the best aspects of human-oriented programming languages and ensure that these elements also are there to assist dogs:

- **Statically typed**. To err is human, but typing is also unnatural for canines. We expect having a compiler with static types will dramatically help cut down on typos and errors in code.
- **Type inference.** While static typing can dramatically reduce errors, having to enter more keystrokes is certainly not dog friendly. The language should attempt to infer as much meaning from expressions as possible.
- **JVM-backend.** Java remains one of the top used/known languages for college grads. By designing a java-friendly language, we allow dogs easy access to the job market.

Let's look at a simple "Hello, world" program in lambda-doge:

```
WOW
main
MUCH PrintLn
"WUF"!
```

As can be clearly seen, the syntax and style of this program are completely natural to the language of doge, as discovered via internet memes.

3. Grammar

Every lambda doge program is built using expressions. Expressions are one of: Let-expressions, Lambda-Expressions, Apply expression, Identifier references or literals.

A lambda doge let expression is used to define a function, and looks as follows:

```
WOW <name>
SUCH <optional forced type specification>
SO <argument name list>
```

```
VERY <body-expression>
```

Functions optionally take both arguments and types for the arguments. If types are specified, they must be specified for all arguments. If argument names are specified, the values of these arguments can be referenced within the body expression. Let's compare Lambda Doge to the more human-friendly Haskell:

Lambda Doge	**Haskell**
WOW Inc SO x VERY Plus x 1 !	Inc x = Plus x 1
WOW Inc SUCH Int => Int SO x VERY Plus x 1 !	Inc :: Int -> Int Inc x = Plus x 1

Lambda doge also has lambda expressions, otherwise known as function literals. These look as follows (with comparisons to scala):

Lambda Doge	**Scala**
MANY friends VERY Happy friends !	{ friends => Happy(friends) }

The MANY keyword denotes a function literal is coming. All identifiers preceding the VERY keyword denote how many arguments the function literal takes.

We list a more complete lambda doge grammar for those interested in implementing their own.

Lambda-Doge Grammar

```
expr := let-expr | lambda-expr | ap-expr | literal | id-ref
let-expr := 'WOW' <identifier> type-list? arg-list? Ap-expr
type-list := 'SUCH' <type>
arg-list := 'SO' <identifier>*
```

```
lambda-expr := 'MANY' <identifier>* ap-expr
ap-expr := ('VERY' | 'MUCH') id-ref expr* '!'
id-ref := <identifier>
literal := <int-literal> | <boolean-literal> |
           <string-literal>
```

4. Type System

The Lambda Doge type system attempts to blend power with simplicity, enabling canines the fastest possible ramp-up time to programming as possible. The type system is composed of two types:

1. Type constructors: T[a], T
2. Type variable: a

A Type constructor can have no type arguments associated with it. In this case it is known as a simple type, and is considered "complete.". Most literals in the language are typed into simple types:

Literal	**Type**
Integer literal	Int
Boolean literal	Bool
String literal	java.lang.String

Functions are encoded in the language as curried instances of the type constructor Function[_,_]. Lambda doge provides a convenience syntax for function types, using the => literal. This allows breaking the type constructor into argument + result pairs, leading to a more haskell-looking type syntax. The following table shows examples:

Function	**Type**	**Abbreviated**
WOW Inc SO x VERY Plus x 1 !	Function[Int, Int]	Int => Int
Plus	Function[Int, Function[Int, Int]]	Int => Int => Int

Lambda Doge performs type inference by running the hindley-milner-like type inference algorithm locally against each let expression. If no type is isolated, then the type will remain as a type variable. In lambda doge, all types that start with a lower-case letter are considered 'variables' which may contain any type.

For instance, lists in Lambda doge are encoded as follows:

List function	**Type**
Nil	List[a]
Cons	a => List[a] => List[a]

So, when constructing a list, the first element defines the type of the list, e.g.

```
WOW mylist
VERY Cons 1 Nil !
```

Will infer the type of mylist to be List[Int].

One complication is how to represent Java types in Lambda Doge. While static java methods can be mapped directly to lambda doge functions, there is the issue of how to handle methods. We use an old trick of defining methods as static functions which take the instance of the class as the first argument. Here's some example mappings:

Java Method	**Java Qualifier**	**Type**
java.lang.Object#toString		java.lang.Object => String
java.lang.String#charAt		java.lang.String => Int => java.lang.Character
java.lang.Integer#MAX_VALUE	static	Int
java.lang.Integer#parseInt	static	java.lang.String => Int

Currently the type system does not take into account inheritance. We think open recursion is hard enough for humans to understand and wish to spare our canine friends. However, as time progresses, we believe there may be opportunity to explore a simple way of allowing "foreign types" to have full object oriented semantics, well preventing most of the headaches associated with such semantics and type inference.

5. Backend

The backend of lambda doge compiles all lambda doge files into Java .class files.   The mapping is done as follows:

- All .doge files are compiled to a single .class file of the same name
- All let-expressions defined within the file are encoded as static methods of the same name.
- Any let expression named "main" is encoded as a "main" method, as used by the Java Virtual Machine (JVM) to run a program. I.e. Lambda-Doge will force the type of the main let expression to conform to Java main standards.

Additionally, as closures are not natively supported (at the JVM level), Lambda Doge performs "lifting" of static method into java.util.Function<?,?> instances.

Here is an example:

```
WOW four_in
SO a b c d
VERY Plus a
MUCH Plus b
SUCH Plus c d ! ! !

WOW closure
SO captured
VERY four_in captured !
```

The first function, four_in, has a type Int => Int => Int => Int => Int, which takes four inputs before returning a value.   In Lambda Doge, functions are values and can be returned. The function called closure has a return type of Int => Int => Int => Int, which means it returns a function which will take three inputs before returning a value.

Encoding this on the JVM is tricky because the JVM makes a distinction between methods and objects. Methods are defined against objects and only methods perform behavior. Objects define data, and only objects can be passed or returned as values.   Therefore treating a function as both behavior and data is not entirely native.

However, Java 8 now defines a java.util.Function<?,?> interface, as well as a LambdaMetaFactory that can be used to construct object-instances of this interface using methods. The LambdaMetafactory can take a set of "bound" arguments, a method handle and will construct a function object which can be returned.   To make use of this, given the natural currying in Lambda-doge, we must adapt all instances of code where we must return a function object such that these instances only pass one value into a static method at a time.   To solve this, we introduce an AST transformer that will expand any partial function application into N let expressions, where N represents the number of unbound arguments. Here's an example.

Phase	**Pseudo-Ast**
Parse	let four_in(a,b,c,d) = Plus(a, Plus(b, Plus(c,d))) let closure(captured) = four_in(captured)
Lambda Lift	let four_in(a,b,c,d) = Plus(a, Plus(b, Plus(c,d))) let closure$four_in$1(a, b, c) = four_in(a,b,c) let closure$four_in$2(a, b) = four_in$1(a,b) let closure(a) = four_in$2(a)

The runtime is then able to appropriately call either the static method for the let expression, or is able to call the apply method defined on the java.util.Function instance, appropriately.

A second complication in the backend is over boolean logic. While most of lambda doge is evaluated eagerly, it is a common optimisation when dealing with boolean if statements to avoid computation for one of the branches of logic. As such, Lambda Doge provides built-in function "ifs" having type Bool => a => a => a. Here, the first argument is a boolean condition. The function returns either the second or third argument, depending on whether the boolean condition is true or false. As an optimisation, this built-in function will avoid evaluating the expression for the second and third arguments until after checking the boolean condition.

Generally the lambda-doge backend is quite simple, and most of the features map quite naturally to JVM bytecode, as the bytecode is stack based.

6. Standard Library

Currently lambda doge comes with a minimal standard library, for use in prototyping[3]. Here's a table of all functions currently defined:

Function	**Type**
tuple2	a => b => (a,b)
fst	(a, b) => a

snd	(a,b) => b
Nil	List[a]
Cons	a => List[a] => List[a]
hd	List[a] => a
tl	List[a] => List[a]
Plus	Int => Int => Int
Minus	Int => Int => Int
Multiply	Int => Int => Int
Divide	Int => Int => Int
ifs	Bool => a => a => a

7. Future Work

The language cannot be considered complete until pattern matching has been implemented in an elegant way. As you may know, dogs are excellent pattern matchers. Here is a common pattern match from a popular dog AI[4]:

```
def next_action(in: SensoryInput): Action =
 in match {
   case See(SomethingThatMoves()) => bark
   case Hungry() => bark
   case FullBladder() => whine
   case Nothing if NotSeenOwner => whine
   case Nothing => sleep
 }
```

We envision adding this to lambda doge using a haskell-style encoding of multiple let-expression definitions with the same name yielding a single physical let expression, where the arguments define the patterns, However given the nature of lambda-doge, the elegance of this approach is dubious, leading to much discussion within the canine community of the best way to bark at things which move.

Conclusions

This language presents an ideal syntax and backend to allow the canine race to enter the workforce. By focusing on Java, it allows them to immediately contribute to most financial and e-commerce institutions. WIth some work, the language could even be extended into a native backend, Allowing dogs to work on mission critical software, such as military, transportation and medical equipment. This is totally because dogs are really good programmers and not, like, some attempt to overthrow the dominant sentient race of earth or anything. Woof.

References

1. https://www.psychologytoday.com/blog/canine-corner/201209/how-many-dogs-are-there-in-the-world
2. Google for "How many programmers are there" and it totally tells you
3. Prototype implementation: https://github.com/jsuereth/lambda-doge
4. https://gist.github.com/anonymous/b3174de0aec16a8e1c7f