Lambda Calculus with Subtyping and Dynamic Semantics Or: Towards a Formal Understanding of Duck Typing

Amery Chang, ac2925

May 18, 2023

Abstract

We extend the Simply Typed Lambda Calculus (STLC) with a few types, most notably records, as well as subtyping semantics. We also added dynamic semantics in order to include the notion of making dynamic type updates to our language. Given these features, we create a model of duck typing in a lightly extended STLC that is useful to compare with structural subtyping.

1 Introduction

This project is a response to the blog post "Is go duck-typed" [Smi20] and a Hacker News discussion thread about it [mbe20]. Smith makes the claim that Golang has features which can be seen as duck typing, citing the example of Go's interfaces.

In figures 1 and 2, Smith claims that by implementing the *Animal* interface's *NumberOfLegs()* method, the types *Spider*, *Dog*, and *Stool* are "each duck-typed as *Animal*". They go on to make the distinction between an "explicit" interface- which they explain to be one where the compiler can "make assumptions about the underlying memory layout so long as the specification of the interface is enforced by the language", and another "waiting" approach, where we pass some type as a parameter to a function and see if the parameter is used in an acceptable way.

In other words, Smith seems to draw a hard line between two methods for implementing interfaces: an explicit form, where the specifications of the interface are addressable in memory (they cite this as a performance improvement and "the most common reason" for using this type of interface), and a form where the validity of using some type as a parameter is run without previous checking. Smith claims that the latter is "usually considered to be duck typing".

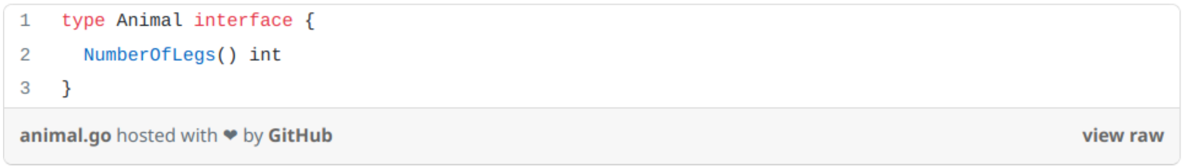
In a Hacker News discussion thread [mbe20] about Smith's article, user *mbell* simply responds that "This is called Structural Typing, and is in contrast to Nominal Typing."

This paper will discuss our implementation of several extensions to the lambda calculus in order to identify whether Smith or mbell are correct. Our primary intent is to implement something that resembles duck typing in order to make an informed judgment of whether Smith's assessment is correct.

2 Definitions

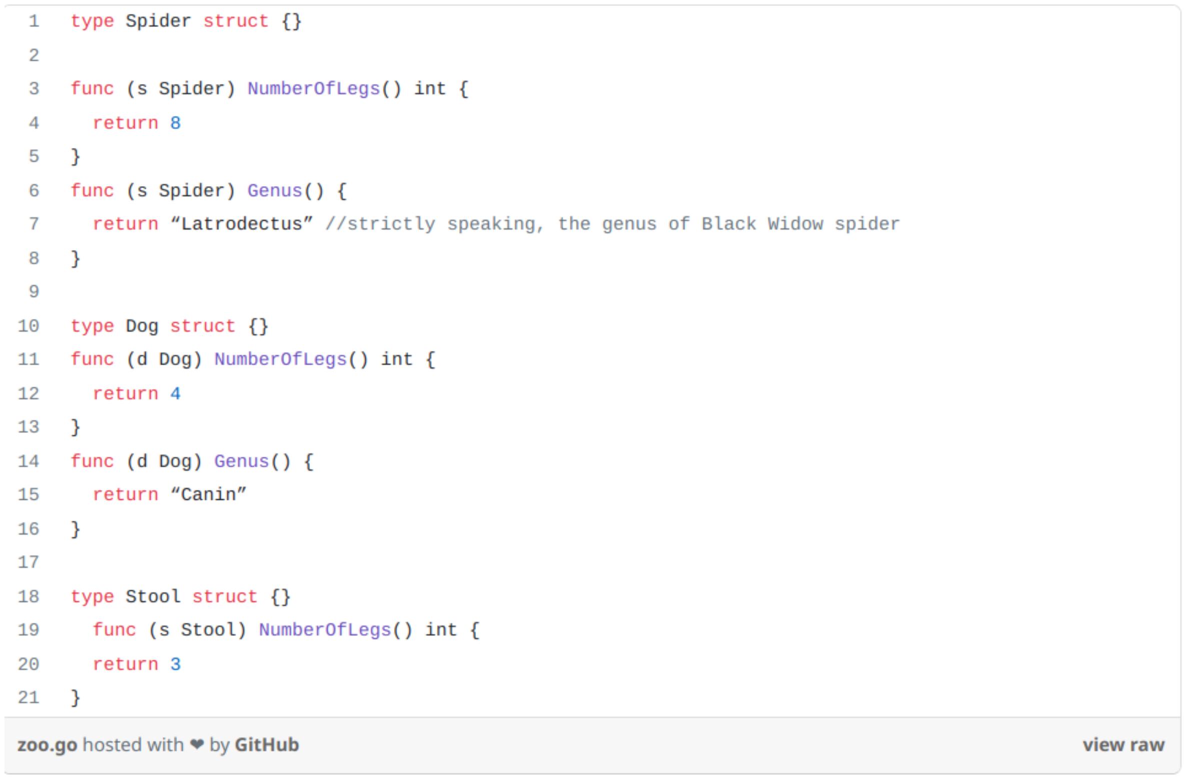
2.1 Structural Subtyping

Subtyping, or subtype polymorphism, is described by the following property: for types S and T, S is a subtype of T if any term of type S can be used in place of a term of T. This is often called the *principle of safe substitution*. An inference rule for this, called the rule of *subsumption*, is given by



```
type Animal interface {
  NumberOfLegs() int
}
```

Figure 1: Example of Go interface, [Smi20].



```go
type Spider struct {}

func (s Spider) NumberOfLegs() int {
  return 8
}
func (s Spider) Genus() {
  return “Latrodectus” //strictly speaking, the genus of Black Widow spider
}

type Dog struct {}
func (d Dog) NumberOfLegs() int {
  return 4
}
func (d Dog) Genus() {
  return “Canin”
}

type Stool struct {}
  func (s Stool) NumberOfLegs() int {
  return 3
}
```

Figure 2: Implementations of Animal interface, [Smi20].

$$\frac{\Gamma \vdash t : S \qquad S <: T}{\Gamma \vdash t : T} \text{ T-SUB}$$

which says that for any value t of type S, if S is *subsumed* by T, then t is also of type T.

To actually implement the properties by which one type would be subsumed by another, there are two common approaches:

1. Structural subtyping: subtyping is defined based on the structure of the types
2. Nominal subtyping: type definitions are named, and subtyping relationships are defined explicitly between names. Note that the requirement for this is that if the program declares that S extends T, S must actually implement the properties of T (this will be checked at compile time).

We focus on structural subtyping in our project. Nominal subtyping has several practical advantages, including the type names themselves are useful (including for defining recursive types), and the near triviality of identifying a subtype relationship [Pie02]. Pierce also notes a ”more contentious” advantage of nominal typing- that they prevent ”spurious subsumption”, when a type S is falsely identified as a subtype of T by being structurally compatible with T. This latter point echoes Smith's example where, by virtue of implementing the *NumberOfLegs()* method in the *Animal* interface, the *Stool* type is subsumed by the *Animal* type even though this is nonsensical in the real world.

2.2 Duck typing

We have not identified research or academic (i.e. textbook) literature that precisely defines duck typing. Smith notes that ”the Python community popularized the phrase 'duck typing'...” and also notes its prevalence in the Ruby language. The Golang specification does not make any explicit reference to duck typing [Goo].

Colloquially, duck typing is described by the application of the *duck test*: “If it looks like a duck and quacks like a duck, it must be a duck.” [VRD09]. The CPython glossary specifies that duck typing "does not look at an object's type to determine if it has the right interface; instead, the method or attribute is simply called or used". There also appears to be general agreement that duck typing is something that occurs only during program execution, and not compile time; a Python protocol even describes structural subtyping as "static duck typing" [VRD].

In the Hacker News thread, user *lkitching* claims "The term duck typing when employed by dynamic languages like Ruby is misleading since it's a consequence of not having a type system at all." [lki20]. If we accept the Python definition of duck typing, and apply it to our *T-Sub* rule, then it seems likely that *lkitching*'s comment is correct. Python's implementation of duck typing seems to not suffice the *T-Sub* rule. We require $\Gamma \vdash t : S$, but the duck typing resists checking the type of t altogether, so we cannot claim this judgment to be true. We can therefore argue that Python's duck typing is not a subtyping relation at all. However, it's not very clear how literally the Python description should be taken; does the typing truly not happen at all, or does it happen in an ad-hoc fashion at runtime? If the purpose of typechecking (with subtyping) is to identify whether one type is equivalent to another type, then the Python description taken literally is clearly not subtyping, since it seems to eschew typing completely. However, if a value of some time is to be used, i.e. scanned for some necessary properties as needed, is this not effectively a form of type checking? Besides, what does Python even mean by "look at its type"? Is that possibly a rejection of nominal, but not structural, subtyping?

3 Implementation

Here, we describe the implementation of a calculus that implements functionality to explore some of the topics discussed so far.

3.1 Terms and Representation

Our primary terms are nothing unusual- we represent the terms of the untyped lambda calculus: *variables*, *abstractions*, and *applications*. To avoid name capture, we implement a mixed naming convention for DeBruijn indices where only bound variables are renamed with an index [McG] [MM]. This helps us avoid name capture while keeping a strong distinction between free and bound variables in lambda bodies, and in turn simplifies accurate printing (since it's easy to identify free variables here, it's easy to know what which variables need a *pickFresh* when printing).

The parsing and printing is an adapted from Matt Wetmore's tutorial for using the Parsec library. [Wet]

Following conventions used in Pierce, we define one type called *Context*, which is an association list of *String* and *Binding*. The Binding can represent either a *NameBind*, the binding of a bound variable to a DeBruijn index (which is the index of the variable in the list), or a *VarBind*, the binding of a value to a type.

3.2 Types

The Lambda Calculus is extended with a few simple additions. To most trivially add subtyping relations, we include the notion of a top class, *TyTop*, which subsumes all other classes (akin to Java's *Object* class).

The addition of a pair type *TyPair* is simply a precursor to adding the record type *TyRecord*. The rules for subtyping on records are defined in Pierce and, in short, amount to the matching of named fields and their types. The subtyping relation of $S : TyRecordf1 <: T : TyRecordf2$ will be true if S has at least the same named fields as T (*S-RCDWIDTH)*, and the types of fields in S are subtypes of their analogous fields in T (*S-RCDDEPTH*).

Pierce makes brief mention of a *dynamic* type, which is described as an "infinite disjoint union" of pairs of (value, type annotation). We simply represent this *TyDyn* as a map of $(String, Type)$, similar to *TyRecord*. We add a function to add fields to this type ad-hoc, which we will perform "at run time" by simply calling that function in a main method. Then, we will perform a function that checks whether the *TyDyn* is a subtype of another (most likely a *TyRecord* or another *TyDyn*).

While this record matching is tedious (and indeed, this model fails to make use of more efficient techniques such as coercion, which is performed at runtime), record matching in this fashion is an integral aspect of determining whether one type is equivalent to another.

4 Conclusion

With a few simple extensions to the lambda calculus, we were able to create semantics for structural subtyping, primarily on records. We also introduce a dynamic type, which gives our calculus an ad-hoc notion of type checking that resembles type checking being performed at run time.

We still do not have a particularly strong notion of duck typing, but this project has led to a few conclusions:

- It seems to be agreed that for something to be ”duck typing” it must occur at run time. Therefore, simulating this requires some notion of dynamic typing in a language.
- Duck typing is, arguably, not really typing at all. Python flat out claims that it is not typing, and Python’s definition of duck typing seems to be incompatible with the subsumption rule, which requires that the type of the value that we are typechecking be identifiable.
- Nonetheless, duck typing requires that something *like* structural subtyping be performed. In order to use the property of a value that’s needed for some operation, the language must identify the properties of that value (e.g. does that value implement a particular function that needs to be called at run time).

To respond to ”Is Go duck-typed?”, we do not believe that Smith has sufficiently demonstrated that Go is duck typed. While Smith does imply that duck typing is necessarily performed at run time (this is essentially their point about ”waiting”), their examples do not show that the language is accessing fields of values only as needed, nor do they identify a source which confirms that that is happening.

Further, we believe that Smith’s claim that ”The most common reason for requiring that an interface be defined is performance” is questionable for two reasons. First, in defining the subtyping relation rules for records, Pierce uses the record permutation rule *S-RCDPERM* to note that the order of fields in records need not be the identical for two records to have a subtype relation. Smith describing that a common interface would perform faster record matching due to memory layout is just an implementation detail. Morever, it seems reasonble to assume that this would not always result in better performance- if we had record types with lots of fields, it may in fact be much more performant to use ”duck typing” to only access the fields we need.

References

[Goo] Google. The go programming language specification.

[lki20] lkitching. ”the fact that one is checked statically and the other...”, 03 2020.

[mbe20] mbell. ”this is called structural typing[0] and is in contrast to nominal...”, 03 2020.

[McG] Callan McGill. ”locally nameless”.

[MM] C. McBride and J. McKinna. ”functional pearl: i am not a number–i am a free variable”.

[Pie02] Benjamin C. Pierce. *Types and Programming Languages.* The MIT Press, Cambridge, 2002.

[Smi20] Ian Smith. Is go duck-typed?, 03 2020.

[VRD] Guido Van Rossum and Fred L. Drake. Pep 544 – protocols: Structural subtyping (static duck typing).

[VRD09] Guido Van Rossum and Fred L. Drake. Python 3 glossary, 2009.

[Wet] Matt Wetmore. ”parsing combinators with parser combinators”.