Higher Order Operator Overloading and Natural Lambda Properties

by Sven Nilsen, 2019

In this paper I show that there is an isomorphism between structural properties with Higher Order Operator Overloading and a natural interpretation of lambda properties.

A Cartesian product[1], or a tuple, is written:

(a, b) : A × B

To access the variable `a` and `b`, two functions `fst` and `snd` can be used:

fst((a, b)) = a
snd((a, b)) = b

The functions `fst` and `snd` are structural properties, because they access parts of the Cartesian product that can be used to reconstruct the tuple:

(fst((a, b)), snd((a, b))) = (a, b)

Many programming languages support naming of structural properties, used the following way:

foo.bar : U The property `bar` of the object `foo`

foo : T
bar : T → U

Under Higher Order Operator Overloading[2] (HOOO), the object `foo` might be a function:

foo.bar := \(x : X) = foo(x).bar

foo.bar : X → U
foo : X → T
bar : T → U

This is isomorphic to some type `T(X)` which has the property `bar : X → U`. Instead of having a lambda returning an object of type `T`, one has an object which contains functions. All these functions are interpreted to be functions of the same source of data:

foo := { bar: \(x : X) = …, baz: \(x : X) = …, … }
foo(x) := { bar: foo.bar(x), baz: foo.baz(x), … }

Due to this isomorphism, this gives a natural interpretation of lambda properties.

For example, a lambda that checks for equality of two booleans:

```
\(a : bool, b : bool) = a == b                    (first lambda)
```

Is logically equivalent to the following (`¬` means NOT and `⊻` means XOR):

```
\(a : bool, b : bool) = ¬(a ⊻ b)                  (second lambda)
```

A lambda is logically equivalent to another if it produces same results for same inputs.

According to Leibniz' law[3], two objects are the same if and only if every property are the same. This means that it is not logically possible to distinguish between the two lambdas above under computation.

However, using natural lambda properties, is it possible to distinguish between these two lambdas? Even computation is not assumed, how can one prove that the two lambdas are not the same objects?

There is an isomorphism between lambdas returning data structures, and data structures containing lambdas. So, for the first lambda, one can perform the transformation:

```
\(a : bool, b : bool) = a == b
        <=>
eq { left: \(a : bool, b : bool) = a, right: \(a : bool, b : bool) = b }
```

`left` and `right` are arbitrary choices that depends on the underlying language representation.
The `eq` object is a node in the Abstract Syntax Tree[4] of this language that performs the computation:

```
eq(a, b) = (a == b)
```

All lambda terms can be transformed into a data structure that contains sub-lambda terms. Since the same transformation can be made on sub-lambda terms, this can be done recursively. This is why an Abstract Syntax Tree can be used to represent lambdas.

This transformation is logically equivalent to the first lambda.
However, by treating `eq` as an object, the natural lambda property `right` becomes:

```
( \(a : bool, b : bool) = a == b ).right <=> \(a : bool, b : bool) = b
```

In the second lambda, there is no property `right`:

```
\(a : bool, b : bool) = ¬(a ⊻ b)
        <=>
not { arg: \(a : bool, b : bool) =  a ⊻ b }

( \(a : bool, b : bool) = ¬(a ⊻ b) ).right        ERROR: The node `not` has no property `right`.
```

Therefore, the two lambdas are not identical when using natural lambda properties.

Natural lambda properties makes it possible to reflect upon the Abstract Syntax Tree at runtime through a designed interface without needing a new type besides lambdas as first class citizens.

References:

[1] “Cartesian product”
Wikipedia
https://en.wikipedia.org/wiki/Cartesian_product

[2] “Higher Order Operator Overloading”
Sven Nilsen, 2018
https://github.com/advancedresearch/path_semantics/blob/master/papers-wip/higher-order-operator-overloading.pdf

[3] “The Identity of Indiscernibles”
Stanford Encyclopedia of Philosophy
https://plato.stanford.edu/entries/identity-indiscernible/

[4] “Abstract syntax tree”
Wikipedia
https://en.wikipedia.org/wiki/Abstract_syntax_tree