Naive Zen Logic

by Sven Nilsen, 2018

Zen rationality is an extension of instrumental rationality with the ability for higher order reasoning about goals. In order to capture some formal features of zen rationality in logical form, it might be best to start out with a simple syntax of agents that reason about beliefs believed to be believed by smarter versions of themselves.

Naive Zen Logic consists of one indeterministic unary operator `.` and one time-asymmetric binary operator `?`. The syntax is designed for easy typing on a standard computer keyboard:

.x	smarter than `x`
x ? y	`x` is believed by `y`

For example:

(“1+1=2” ? .me) ? me

I believe that a smarter me will believe “1+1=2”.

A self-consistent agent is an agent that believes something when it believes it believes it:

```
consistent := \(x : agent) = ∀ a { ((a ? x) ? x) → (a ? x) }
```

This can also be written using path semantical notation:

```
consistent := \(x : agent) = ∀ a : [? x] (? x) { a ? x }
```

The `?` operator uses quotation instead of normal function evaluation. There is only evaluation at top level, in the language where the Naive Zen Logic is embedded. This language might be an agent that thinks, but it can also be a way of showing how this kind of reasoning without specifying an agent.

In path semantical notation, `()` is used to terminate and `[]` is used to chain operators:

a : (? x)	a ? x		`a` is believed by `x`
a : [? x] (? x)	a ? x : (? x)	(a ? x) ? x	`x` believes `a` is believed by `x`

A zen-consistent agent has the following property (in addition to being self-consistent):

```
zen_consistent := \(x : consistent) = ∀ a { ((a ? .x) ? x) → (a ? x) }
```

In path semantical notation:

```
zen_consistent := \(x : consistent) = ∀ a : [? .x] (? x) { a ? x }
```

Zen-consistency can be thought of as the ability to imagine what a smarter version of yourself would believe. The more detailed and accurate this imagination is, the more likely is it that something can be learned without actually being that smart. Everything you believe that a smarter version of yourself believe, is something that you should believe too.

Since a smarter version of yourself would be better at imagining a smarter version of yourself (it would not necessary need to imagine a smarter you, it could just observe itself), it would have more reason to believe what it believes the smarter version of yourself would believe, than you do. Therefore, if you believe that a smarter version of yourself believes it believes something, then you should too:

```
(((a ? .me) ? .me) ? me) → (a ? me)
((a ? .me) ? me) → (a ? me)
(a ? .me) → (a ? me)
true
```

In path semantical notation, for left side of implication `→`:

```
a : [? .me] [? .me] (? me)
a : [? .me] (? me)
a : (? me)
```

This also works recursively: A smarter version of yourself that imagines a smarter version of itself (not you) would believe what it believes that second smarter version believes. Therefore, you should believe what you imagine a smarter version of yourself believes a smarter version of itself believes.

```
a : [? ..me] [? .me] (? me)
a : [? .me] (? me)          Apply zen-consistency for smarter myself
a : (? me)                  Apply zen-consistency for myself
```

This assumes that the smarter version of yourself that you imagine is zen-consistent.

Naive Zen Logic leads to an infinite recursion of agents imagining smarter versions of themselves. However, what does “smarter” mean? Naive Zen Logic can also define this (assuming rational agents):

```
smarter := \(y : zen_consistent , x : agent) =
        ∀ a : (? x) { a ? y } ∧ ∃ a : (? y) ∧ ¬(? x) { ¬(a ? x) ? y }
```

Here, the language reasoner believes `y` is smarter than `x` in some aspect if `y` believes everything that `x` believes but also something that `x` does not, `y` knows that `x` does not, yet `y` still behaves as if `y` believes it while being zen-consistent. It does not mean that `y` is aware of being smarter, only that `y` believes it without reflecting over it.

Since `smarter` can be used to define a sub-type, one can use the `.` notation:

```
y : (smarter x)          y : .x          `y` is smarter than `x`
```

When `y : .y` it fails type checking, because `y` can not believe something itself does not believe. Likewise, `y = .y` would mean `y` is equal to something smarter than itself, which can not be true.