From Zero to Logic in Haskell

Malvin Gattinger

Originally published in: *A Programming Road to Logic, Maths, Language, and Philosophy: A Tribute to Jan van Eijck on the Occasion of His Retirement* (May 2017), edited by Stefan Minica, Christina Unger and Yanjing Wang.

My first contact with Jan and Haskell coincide. At the beginning of my second year as a Master of Logic student I took his course "Functional Specification of Algorithms".

My previous programming experience had been completely outside universities. Moreover, it was mainly in PHP which you might call the opposite of Haskell. Not just imperative and stateful as hell, but with so many weird parts that there are dedicated forums to make fun of them. Maybe this is why I usually did not think about connections between my interests in mathematics and computers.

Jan easily changed this and showed me a way to merge logic and programming. The final blow to my imperative upbringing happened towards the end of the course. Jan gave a short introduction to DEMO, the epistemic model checker. Since then my favorite example of how well Haskell accommodates logic is the comparison between mathematical definitions as we write them in a paper and their implementations. For example, consider the syntax of Public Announcement Logic (PAL):

$$\phi ::= \top \mid p \mid \neg\phi \mid \phi \wedge \phi \mid K_i\phi \mid [\phi]\phi$$

This can be easily translated to a new `data` type:

```
type Prop = Char
type Agent = String
data Form = Top | P Prop | Neg Form | Con Form Form | K Agent Form | Ann Form Form
```

The symmetry continues when we interpret the language. Here is the standard truth definition for PAL, saying when formulas are true in a pointed model:

$$\begin{array}{lcl}
\mathcal{M}, w \vDash \top & \Leftrightarrow & \text{always} \\
\mathcal{M}, w \vDash p & \Leftrightarrow & p \in V(w) \\
\mathcal{M}, w \vDash \neg\phi & \Leftrightarrow & \text{not } \mathcal{M}, w \vDash \phi \\
\mathcal{M}, w \vDash \phi \wedge \psi & \Leftrightarrow & \mathcal{M}, w \vDash \phi \text{ and } \mathcal{M}, w \vDash \psi \\
\mathcal{M}, w \vDash K_i\phi & \Leftrightarrow & \forall v \sim_i w : \mathcal{M}, v \vDash \phi \\
\mathcal{M}, w \vDash [!\phi]\psi & \Leftrightarrow & \mathcal{M}, w \vDash \phi \;\Rightarrow\; \mathcal{M}^{\phi}, w \vDash \psi
\end{array}$$

How do we write this in Haskell? First we need a definition of models.

```
type World = Int
data Model = Mo {worlds :: [World], val :: World -> [Prop], rel :: Agent -> World -> [World]}
```

Now the semantics given by $\vDash$ above can be written as a function from pointed models and formulas to booleans. The helper function `!` implements the update from $\mathcal{M}$ to $\mathcal{M}^{\phi}$.

```
eval :: (Model,World) -> Form -> Bool
eval (_,_) Top           = True
eval (m,w) (P p)         = p `elem` (val m w)
eval (m,w) (Neg phi)     = not (eval (m,w) phi)
eval (m,w) (Con phi psi) = eval (m,w) phi && eval (m,w) psi
eval (m,w) (K i phi)     = and [eval (m,v) phi | v <- rel m i w]
eval (m,w) (Ann phi psi) = eval (m,w) phi <= eval (m ! phi,w) psi
```

```
(!) :: Model -> Form -> Model
(!) m phi = m
    { worlds = filter (\w -> eval (m,w) phi) (worlds m)
    , rel = \i w -> filter (\w -> eval (m,w) phi) (rel m i w) }
```

This minimalistic toy variant of DEMO can be used as follows:

```
myModel :: Model
myModel = Mo [0,1] myval myrel where
  myval 0 = "pq"
  myval 1 = "p"
  myrel "Anne" 0 = [0,1]
  myrel "Anne" 1 = [0,1]
  myrel "Bob" 0  = [0]
  myrel "Bob" 1  = [1]
```

```
λ> eval (myModel,0) (K "Bob" (P 'q'))
True
λ> eval (myModel,0) (K "Anne" (P 'q'))
False
λ> eval (myModel,0) (Ann (P 'q') $ K "Anne" (P 'q'))
True
```

Why would you want to use anything else to implement Logics and model checkers?% [1]

```
eval :: (Model,World) -> Form -> Bool
```

$$\begin{aligned}
\mathcal{M}, w \models \top &\Leftrightarrow \text{always} \\
\mathcal{M}, w \models p &\Leftrightarrow p \in V(w) \\
\mathcal{M}, w \models \neg\phi &\Leftrightarrow \text{not } \mathcal{M}, w \models \phi \\
\mathcal{M}, w \models \phi \wedge \psi &\Leftrightarrow \mathcal{M}, w \models \phi \text{ and } \mathcal{M}, w \models \psi \\
\mathcal{M}, w \models K_i\phi &\Leftrightarrow \forall v \sim_i w : \mathcal{M}, v \models \phi \\
\mathcal{M}, w \models [!\phi]\psi &\Leftrightarrow \mathcal{M}, w \models \phi \Rightarrow \mathcal{M}^{\phi}, w \models \psi
\end{aligned}$$

Given this background, I often look at new definitions of semantics for a new logic and immediately wonder what they would look like in code. Can we easily translate all logics and their semantics to Haskell? Of course not. The language is more restrictive than mathematical notation, but this can be seen as a feature. When I started to implement the Logic of Agent Types and Questions from Liu & Wang 2013 one obstacle was this definition (adapted from page 138):

$$\mathcal{M}, w \models_\mu [!_a]\phi \Leftrightarrow \text{for all } \psi : \mathcal{M}, w \models_\mu [!_a\psi]\phi$$

The intended meaning of $[!_a]\phi$ is "No matter how agent a answers the current question μ, ϕ will be true afterwards." On the right side of the definition we quantify over all formulas to represent all possible answers. But of course we can not run through infinitely many ψ in an implementation that should ever be run (and finish). But in this case there is an easy way out: The logic only formalizes binary questions μ, so the only relevant answers are equivalents of μ and $\neg\mu$. Thus we do not actually care about all formulas, only those two, and the logic can still be implemented easily.

One of the things I learned from Jan is that in situations like this we can realize that implementation is not a one-way street: we might as well go back and change the definition that we wanted to implement (and this is actually what (Liu & Wang 2013) already do implicitly on page 138). Haskell thus prevents us from defining something in a non-computable or non-constructive way if there is no real reason to do so.

[1] The original website publication included this animation: https://malv.in/2017/Festschrift-JvE-ZeroLogicHaskell-animation.gif.