HyLogic Documentation

Release 1a

Marko Manninen

Aug 03, 2017

Contents

1 HyLogic **3**

1.1 Current status . . . 3
1.2 Contents . . . 3
1.2.1 Requirements and installation . . . 4
1.3 Jupyter and Hy . . . 4
1.4 HyLogic module . . . 4
1.4.1 Propositional logic . . . 5
1.5 Symbols . . . 5
1.6 Basic axioms and theorems . . . 5
1.6.1 Propositions . . . 5
1.7 Semantical notes . . . 7
1.8 Truth-bearer in action . . . 7
1.9 Resolution of the famous paradox . . . 9
1.9.1 Formulas . . . 10
1.10 Connectives . . . 10
1.10.1 Argumentation . . . 11
1.11 Validation of the argument form . . . 12
1.12 De Morgan's laws . . . 13
1.12.1 First-order logic . . . 14
1.13 Quantifiers, predicates, variables, sets . . . 14
1.13.1 Truth tables . . . 15
1.13.2 Venn diagrams . . . 15
1.14 The MIT License . . . 16

HyLogic is a set of macros to test and automate logical expressions in Hy. Convenient propositional logic, predicate logic and first-order logic notation is supported.

Contents:

CHAPTER 1

HyLogic

Propositional, predicate, and first-order logic evaluator for Hy authored by Marko Manninen, 2017.

$$P ß Q$$
$$Q$$
$$---$$
$$\therefore P$$

Current status

Draft.

The Python Package Index installation is not provided yet, but planned in future.

Contents

- Requirements and installation
- Propositional logic
- Semantical notes
- Truth-bearer in action
- Resolution of the famous paradox
- Formulas
- Argumentation
- First-order logic
- Truth tables

- Venn diagrams
- License

Requirements and installation

Jupyter and Hy

Some work is required to get this interactive document running on your local computer (Mac, PC, Linux). First you need Jupyter Notebook and Calysto tools. Easy way to get Jupyter Notebook running is to use Anaconda package from Continuum: https://www.continuum.io/downloads. It will also install Python language interpreter to your computer.

Hy language, which by the way is a cool Lisp syntax and feature set upon Python, you can get from: https://github.com/hylang/hy. Install it and then follow Calysto Hy kernel installation instructions from their GitHub project page: https://github.com/Calysto/calysto_hy. Note, that Calysto tools does not contain Calysto Hy kernel. That's why Hy kernel for Jupyter Notebook needs to be installed also.

Hy is selected for a core Logic implementation language because its syntax resembles mathematical and logic notation in many ways, mostly the use of parentheses in the native syntax and no separate parser is needed for that purpose. Not that the parser implementation was not tried for three different languages before the selection: http://plcparser.herokuapp.com/

With macro support Hy can be extended pretty easily to satisfy notational needs for this project.

After installations you should be ready to print environment information running the following Hy code:

```
(import hy sys)
(print "Hy version: " hy.__version__)
(print "Python" sys.version)
```

```
Hy version:  0.12.1
Python 3.5.2 |Anaconda custom (64-bit)| (default, Jul  5 2016, 11:41:13) [MSC v.1900␣
↪64 bit (AMD64)]
```

HyLogic module

Finally you need to retrieve HyLogic module from GitHub: https://github.com/markomanninen/hylogic. If you run current Notebook document from the module directory, then you should be fine to import necessary macros.

```
; require macros and import functions and variables
(require (hylogic.macros (*)))
(import [hylogic.macros [*]])
; NL for newlines
(setv NL "\r\n")
```

Note: If all you need is a command line (console) interface, then you don't actually need Jupyter Notebook and Calysto. Python, Hy, and `HyLogic` module are basic prerequisites. By downloading and placing provided `Hyffix` and `HyLogic` directories to your script root, you can get `HyLogic` running even on Andoid with Termux!

Hyffix is used to support infix, prefix and affix notation plus provide operator precedence functionality.

Propositional logic

This documentation will provide a lot of introductionary material for understanding concepts of logic, not just how to use `HyLogic` module. Main motivation is to provide a computational playground for studying logic, testing logical clauses and automated deduction. See: https://en.wikipedia.org/wiki/Automated_theorem_proving and https://en.wikipedia.org/wiki/Automated_proof_checking.

Symbols

Propositional constants:

- (True / 1)
- (False / 0)

Basic axioms and theorems

- Identity P = P
- Negation ¬ = and ¬ =
- Double negation ¬¬ =
- All well defined statements are true (explicit metalogical axiom)

The last axiom may look odd at the moment, but is clarified later on the *Semantical notes* section.

Propositions

Propositional variables are created with `defproposition`, `defpropositions`, `defproposition*`, and `defpropositions*` macros. The last two macros also creates a negated propositional variable to reduce some repetitive work. In `HyLogic` a proposition consists of one mandatory and two optional parameters:

Mandatory:

1. A propositional variable that is usually denoted by a capital letter like P, Q, and R. Often small letters are used too but we have reserved small letters for predicate variable names. But it is really up to you, what letters to use. You can use multi-character symbols too. For example $VaR!$ is a proper symbol name as well. Using commonly used variable names and conventions improves readability and understandability of the logic expressions however. A propositional variable is also called a sentential variable.

Optional (statement):

2. A truth value that is either $True$ or $False$. Default is $True$. A truth value can also be defined by using the number 1 for $True$ and the number 0 for $False$ or by using constant symbols and respectively. Note that in some logic systems may be used for "unknown" rather than $False$.
3. A sentence that is a literal representation of the proposition such as the phrase "Today is Tuesday". Default is an empty string (""). Althought in `HyLogic` it doesn't really matter what is the content of the sentence, in practice we urge to find out complementizer from the written natural text. It means that in the sentence there should be a clear situational condition mentioned. If there is a state of affair in the sentence like "A is B" then it becomes a truth-maker and consequently we are able to transfer it to a proposition, which then is a truth-bearer.

So the format of the macro (`defproposition`, `defpropositions`, `defproposition*`, `defpropositions*`) to initialize a proposition in `HyLogic` is the following:

```
(macro symbol &optional [truth-value True] [sentence ""])
```

Example 1.1

Let us first define a proposition variable P by using `defproposition` macro and output it:

```
(defproposition P)
(print P)
```

```
P=True
```

Note, how truth value is set to $True$ by default. It is recommended that the truth value is set to $True$ for statements because interpretation of the arguments may get trickier if $False$ is used as we can see from the Example 1.3.

Example 1.2

With `defpropositions` marco you can create multiple proposition variables at once:

```
(defpropositions P Q R)
(print P Q R)
```

```
P=True Q=True R=True
```

It is possible to define only simple proposition variables with the mass creation `defpropositions` and `defpropositions*` macros. That is, you cannot give the truth value and the sentence on them. But on the other hand, it is possible to change the truth value and the sentence via object properties afterwards.

Next we will change the truth value of the created proposition P to $False$, plus give the literal meaning (sentence) for the proposition Q:

```
; alter the truth value of the proposition
(setv P.truth-value False)
; set the literal sentence of the proposition
(setv Q.sentence "This proposition is true")
; output modified propositions
(print P)
(print Q)
```

```
P=False
Q<This proposition is true>=True
```

Example 1.3

Let us then redefine two propositional variables P and Q and their negations P and Q by using a special `defproposition*` macro. Now we will utilize the full parameter set by also giving the specific truth value and the literal sentence:

```
(defproposition* P False "Today is Tuesday")
(defproposition* Q True "John will go to work")
```

Output propositions:

```
(print P NL ¬P)
(print Q NL ¬Q)
```

```
P<Today is Tuesday>=False
 ¬P<Today is Tuesday>=True
Q<John will go to work>=True
 ¬Q<John will go to work>=False
```

From the output we find each proposition and their negation represented in a string format that distinguishes all three aspects of the proposition, namely the symbol, the literal representation, and the truth value of the proposition.

Semantical notes

Maybe a small explanation here is in place because understanding the basic components of the propositional logic requires the understanding of the common convention on how propositional logic works and how it is represented in a written or a spoken format.

When we define the proposition P to mean "Today is Tuesday" and to be $False$, the following happens. We define that the symbol P denotes the sentence "Today is Tuesday" in natural human language. We also define that the truth value of the statement is $False$. Thus we *could* understand the proposition P to say something like "Today is not Tuesday" or maybe even something like: "Today is Tuesday", but that the statement is not true!

There is a possible pitfall in these expressions. Strictly speaking, the proposition P, as an object in `HyLogic`, is just stating that it has the sentencial property which value is "Today is Tuesday" and it has the truth value which is $False$. And **that statement is metalogically true** because the proposition object has been stored in the computer memory in that state.

The last part is important. We have to rely on the top-most metalogical axiom that all statements, as they are expressed, are invariantly true. Also, we are not trying to determine if it is really Tuesday today. In some sense, logic is not meant to find out truth from the world, but to help to follow logical steps to determine the consequence of the predefined true statements. Given the assumptions are true, and logical rules are followed, then we can count on that the consequence is true too. In our example we defined P to be $False$ and then we will deduce the rest of the argumentation according to that definition.

However, the negation of the proposition P, which is automaticly generated by the `defproposition*` macro, is P (read: not P). By literal representation it could be written: "It is not the case that Today is Tuesday". `HyLogic` module doesn't try to formulate literal representations of the negated sentences. Module just formulates negations by prepending symbol to the propositional variable and switching the truth value to its opposite. Sentence is left intact.

The truth value of the negation of P in our example is $True$ because the original P was defined to be $False$ and because we defined negation to work such way in the basic axioms and theorems.

Truth-bearer in action

Because initialized proposition P is a truth-bearer, thus it has the truth value, either $True$ or $False$, we can ask for it. There are several ways of doing it, because Hy language (backed up with Python) and HyLogic (including Hyffix) provides a large core of functions.

Let us use equality comparison function to compare if P is $True$ by Hy way:

```
(= P True)
```

```
False
```

Because a proposition object contains a truth-value property, we could achieve and compare it directly too in Hy code:

```
; is the truth-value of P True?
(if (= P.truth-value True)
    ; if it is, then
    (print "Yes, it is!")
    ; else
    (print "No, it is not!"))
```

```
No, it is not!
```

Above notation is so called prefix or Polish notation which is common in Lisp languages. `Hyffix` module, that is included in `HyLogic`, provides infix as well as more exotic affix notation support. `Hyffix` module provides `deffix` macro that you can use for infixed logical expressions. Alternatively same can be achieved by `#$` macro shorthand.

In the following example equivalence operator is used to find out if proposition P is either $True$ or $False$. Other operators, or connectives as they are called in logic, are fully listed in the *Formulas* section.

Using `deffix` macro:

```
; P is equivalent to  (True)?
(deffix P  )
```

```
False
```

Note, that we are kind of "asking" if P is $True$, but in reality we are passing P and to the equality function, which determines that P, that is $False$, is not same as $True$. Thus result is $False$.

Using `deffix` macro shorthand to achieve similar comparison, but this time we are comparing P to $False$:

```
; P is equivalent to  (False)?
#$(P  )
```

```
True
```

Note, that in `deffix` macro, = is a variable assignment function and it will change the value of propositions instead of comparison. This behaviour may change in the future...

In this example we will test, if double negation of the proposition P is equivalent to P:

```
; P is equivalent to not not P (double negation)?
#$(P  (¬(¬P)))
```

```
True
```

Double negation version of the proposition symbols are not generated by `defproposition*` macro. Just single negation variables are generated. Thus we can't use `¬¬P`, but should use either `(¬ ¬P)` or `(¬(¬P))`. One could also utilize `defproposition` macro and generate double negated propositions in the following manner:

```
(defproposition ¬¬P False "Today is Tuesday")
(print ¬¬P)
#$(P  ¬¬P)
```

```
¬¬P<Today is Tuesday>=False
```

```
True
```

As said before, the proposition variable symbol can be anything, not only single letters. In the above example negation symbol has no independent meaning or functionality. But, if negation symbol is used by alone, it actually refers to the unary boolean operation, contra to other connectives, which are n-ary binary operators.

Resolution of the famous paradox

At first glance, the separate mutable truth value may feel unnecessary and confucing. Why should we reassert, that "Today is Tuesday" is $True$, because the truthness is already stated in the sentence? Separation is really the key to prevent "real" paradoxes occur in the classical logic. Take for example the famous variation of the Liar's Paradox:

"This sentence is false"

Eubulides, who formulated it in the fourth century BC, said that if *that* sentence, or proposition, is true, then *it* can't be true, because *it* says *it* is false, hence the paradox. Paradox means that the given clause is both true and false at the same time, or neither one, or that the truthness of the statement changes mutually. So we can't decide which one it is, true or false. In reality, the paradox comes from the confusion of the references of the properties of the proposition. That's why *it* was written in italics. References are often ambiguous in a written language.

In HyLogic, the sentence itself doesn't define or hold the truth value of the proposition. The truth value and the sentence are properties of the proposition, as is the selected symbol too, for example P. For the practical purposes we have modelled that the proposition is an object with three properties. We can describe the set of proposition objects x having $Symbol$, $TruthValue$, and $Sentence$, latter two having no mutable $Relation$ with this notation:

$$\mathbb{S} = \{x | Symbol(x) Relation(TruthValue(x), Sentence(x)) \underset{endsplit}{} Thuswecandotwoclarifyingthings:$$

to say that the proposition symbolized with P with a sentence "This sentence is false" has the truth value $True$

to use only the alternative symbol, instead of the sentence, and say that the proposition P has the truth value $True$

Now, it becomes apparent that what we try to define in the Liar's Paradox is that P is $True$. There is no contradiction or paradox in this representation. Whatever is stated in the sentence of proposition P, it does not affect to the truth value of the statement. The truth value is a separate and an independent property (attribute) from the sentence. Moreover, the truth value doesn't have a mutable relation to the sentence, just to the overall state of the proposition. This can be properly presented by a model theory and is clearly emphasized in the semantic theory of truth.

It is also easy to see that as a logical statement, the sentence "This proposition is false" is not well defined because it does not really contain a clear state of affair for the proposition. To formulate it to logic language it would become "This proposition" is $True$. But then "This proposition" does not have a truth claim as we would expect. Compare to similar "Today is tuesday" is $True$, which makes clear claim on the sentence. "Today is Tuesday" tries to make the truth. "This proposition" phrase is not a truth-maker and doesn't really fit for propositional statement. Again, not that it would have any effect in `HyLogic`, where the sentence could very well be just an empty string!

Finally, according to basic axioms, all statements are true. This requirement prevents the possibility of the infinite truth value assigment, that is, the self-referencial mutability of the truth value. Together these axioms prevents paradoxes to occur in HyLogic, or classical logic more generally. Solution has familiar identities, because we designed three properties on a proposition model categorically similar to Kleene's strong three-valued logic. And we designed the lowest level of hierachy with metalogical "All statements are true" that is also similar to Russell's and Tarski's solutions

of paradoxes. Plus there is a reference to Gödel's first incompleteness theory, because the unique metalogical axiom, really just a model in a computer memory, is the necessary axiomatic system outside of the logic system itself.

Althought, this may sound promising, there is still "the million dollar" problem open:

The idea is that whatever semantic status the purported solution claims the liar sentence to have, if we are allowed freely to refer to this semantic status in the object language, we can generate a new paradox.

https://plato.stanford.edu/entries/self-reference/

Proposed logic system does not invoke self-references in theory. But the language platform, Hy in this question, does not prevent of using self-references and mutate the truth-value. It is easy to make an infinite loop and demonstrate that situation.

Let us first initialize the ostensible paradoxical proposition:

```
(defproposition P True "This proposition is False")
```

In the language level we can find a contradiction already. Sentence says that *this* proposition is $False$, but we have defined the proposition P to be $True$. Sentence does not know the truth value nor it can change the truth value. Furthermore, it has no functionality to refer to itself. Or if it would have, then where would it refer? If to the sentence itself, then what part of it? Would it say "False" is "False"? Or would the sentence refer to the proposition, or to the truth value of the proposition? The sentence is really dummy about that since it is just a description for the proposition, nothing more.

But let us assume that the sentence could change the value of the proposition and then the proposition would change the value of sentence. Following code should illustrate it:

```
(if (= P.truth-value False)
    (do
      (setv P.sentence (% "This proposition is %s" (str P.truth-value)))
      (print P)))

(if (= P.truth-value True)
    (do
      (setv P.sentence (% "This proposition is %s" (str P.truth-value)))
      (print P)))
```

```
P<This proposition is True>=True
```

```
'This proposition is True'
```

After demonstrating this, we will highly recommend to use the default truth value $True$ in propositions to simplify things. And not to change the value after initialization.

But due to automated factorization of the arguments and proofs, and demonstration purposes, we will diverge from that rule in the following examples anyway.

Formulas

Connectives

The list all connectives to operate with the propositional and the first-order logic:

```
(for [[f data] connectives]
  (print (last data) "\t" (first data) "    \t" (second data)))
```

```
¬    not             Negation
    and             Conjunction
↑    nand            Nonconjunction
    or              Disjunction
↓    nor             Nondisjunction
    xor             Exclusive or
    xnor            Nonexclusive or
    eqv             Equivalence
    neqv            Nonequivalence
←     cimp            Converse implication
    cnimp           Converse nonimplication
    mimp            Material implication
→     mnimp           Material nonimplication
```

Argumentation

Introducing `defargument`, `defpremise`, and `defconclusion` macros.

In a propositional logic, an argument is a set of premises following each other where the final premise is distinguished and called the conclusion. This is also called deductive reasoning where the more specific conclusion is reasoned from the more general premises or assumptions. In `HyLogic`, an argument is created by `defargument` macro which returns an object that can be assigned to a variable for further interaction. `Defargument` takes a serie of premises defined by `defpremise` macro plus the final conclusion defined by `defconclusion` macro. Each premise is a set of formulated propositions and axioms constructed by known inference rules.

Example 2.1

The next example is meant to demonstrate argumentation process in `HyLogic`. It is the famous modus ponens implication elimination rule. But there is a small twist that should stress the need of accuracy on small details in logical reasoning.

For the Modus Ponens argument, we will first define the implication premise "if P is True, then Q is True". Then we will define the second premise "P is True" and finally the conclusion "Q is True". Symbolically and shortly this is denoted with the expression: $P ß Q, PQ$.

```
(setv a
  (defargument
    ; if the proposition "Today is Tuesday" (P) is True
    ; then the proposition "John will go to work" (Q) must be True as well
    ; note that this does not set Q True, but just gives a rule
    (defpremise P → Q)
    ; but we stated earlier on example 1.3 that the proposition "Today is Tuesday"␣
↪(P) is False.
    ; so how should we deal with it now?
    (defpremise P)
    ; well therefore, both <John will go to work>=True
    ; and <John will go to work>=False should be concluded as a valid argument
    (defconclusion Q)))
;(print a)
```

It means that if Today is Tuesday is False OR "John will go to work" is True, then premise is True

Thus if "Today is not Tuesday" then "John will go to work" or "John will not go to work"

Moreover, because P is False, it tells nothing about Q so we can accept both True and False statements of Q

Validation of the argument form

```
(defproposition* P False "Today is Tuesday")
(defproposition* Q True "John will go to work")

(print
  (deffix P) (deffix Q) NL
  "If P, then Q =" (deffix (P → Q)))
```

```
P<Today is Tuesday>=False Q<John will go to work>=True
 If P, then Q = True
```

```
(defn gs [s] (if (= s True) s.symbol (+ "¬" s.symbol)))

(print (gs P) " → " (gs Q) "\t\t\t" (deffix P → Q))
(print (gs P) "\t\t\t\t" (deffix P = True))
;(print "(" (gs P) " → " (gs Q) ")  " (gs P) "\t\t" (deffix (P → Q)  P))
(print (gs Q) "\t\t\t\t" (deffix Q = True))
(print "((" (gs P) " → " (gs Q) ")  " (gs P) ")  " (gs Q) ")\t" (deffix ((P → Q) ␣
↪P) → Q)))
```

```
¬P  →  Q                    True
¬P                         False
Q                          True
(( ¬P  →  Q )   ¬P )   Q )        True
```

```
(defproposition* P)
(defproposition* Q)

(print (deffix (¬P  Q)  (¬P → Q)))
(print (deffix P = True))
(print (deffix ¬Q = False))
(print (deffix ((((¬P  Q)  (¬P → Q))  P) → ¬Q)))
```

```
True
True
True
False
```

```
(print P)
(print Q)
```

```
P=True
Q=True
```

Slightly more complicated argument is shown next.

```
(defproposition P True "It is raining")
(defproposition Q True "It is cold outside")
(defproposition R False "I'm indoors")

(print P NL Q NL R)
```

```
P<It is raining>=True
 Q<It is cold outside>=True
 R<I'm indoors>=False
```

```
; set up argument inference rules
(setv a
  (defargument
    ; If "it is raining and it is cold outside" then "I'm indoors"
    (defpremise (P  Q) → R)
    ; It is raining and it is cold outside
    (defpremise (P  Q))
    ; Therefore, I'm indoors
    (defconclusion R)))
```

```
(print a)
```

```
  (P  Q) → R
  (P  Q)
--------------
 R
```

```
(print
  (deffix P) (deffix Q) (deffix R) NL
  "If P and Q, then R =" (deffix (P  Q) → R))
```

```
P<It is raining>=True Q<It is cold outside>=True R<I'm indoors>=False
 If P and Q, then R = False
```

De Morgan's laws

https://en.wikipedia.org/wiki/De_Morgan%27s_laws

```
(defpropositions P Q)
```

The negation of conjunction:

```
(deffix (¬ (P  Q)) → (¬P  ¬Q))
```

```
True
```

The negation of disjunction:

```
(deffix (¬ (P  Q)) → (¬P  ¬Q))
```

```
True
```

First-order logic

Quantifiers, predicates, variables, sets

```
(tuple (range -9 0))
```

```
(-9, -8, -7, -6, -5, -4, -3, -2, -1)
```

```
;(x>0<10)(x>0)
; (all (map (fn (x) (x > 0)) (range 1 10)))
( (x) (x > 0) (range 1 10)) ; all items [1 ... 9] are greater than 0?
```

```
True
```

```
;all(map(lambda x: x > 0, range(1 10)))
(all (map (fn (x) (> x 0)) (range 1 10)))
```

```
True
```

```
( (x y) ((x > 0)  (y < 0)) (range 1 10) (range -9 1)) ; test this case
```

```
True
```

```
( (x) ( (y) ((x > 0)  (y < 0)) (range -9 0)) (range 1 10))
```

```
True
```

```
( (x) ((x > 0)  ( (y) (y < 0) (range -9 0))) (range 1 10))
;(macroexpand `( (x) ((x > 0)  ( (y) (y < 0) (range -10 -1))) (range 1 10)))
```

```
True
```

```
; Every whole number is divisible by 1 and itself.
;(x)(Div(x,x)(Div(1,x))

;(defoperator mod [x y] (% x y))
(defoperator mod0? [x y] (zero? (% x y)))
(defoperator Div [x y] (mod0? x y))

(setv domain-set [1 2 3])

( (x) ((Div x 1)  (Div x x)) domain-set)
```

```
True
```

```
(setv DX [1 1]
      DY [-1 -2])
; all[any[1-1=0 1-2=-1] any[1-1=0 1-2=-1]]
( (x) ( (y) (x + y = 0) DY) DX)
```

```
True
```

```
; all[1-1=0 1-2=-1 1-1=0 1-2=-1]
( (x y) (x + y = 0) DX DY)
```

```
False
```

```
( (x) (x < 1) (range 0 10)) ; is at least one item of [0 ... 9] smaller than 1?
```

```
True
```

```
; (x)((¬Div(x,x))(¬Div(1,x))
( (x) ( (¬ (x mod0? 1) ) (¬ (x mod0? x) ) ) domain-set)
```

```
False
```

```
; any[1-1=0 1-2=-1 1-1=0 1-2=-1]
( (x y) ((x > 0) (y < 0)) DX DY)
```

```
True
```

```
; any[all[1-1=0 1-2=-1] all[1-1=0 1-2=-1]]
( (x) ( (y) (x + y = 0) DY) DX)
```

```
False
```

Truth tables

```
(truth-tables-html 2 cimp?)
```

```
(truth-tables-html 2 eqv?)
```

```
(truth-tables-html 2 xnor?)
```

Venn diagrams

```
;(venn-diagram)
```

```
(defn odd? [x &rest y]
  (= 1 (% (+ x (sum y)) 2)))

(deffix (odd? 1 1 1))
```

```
True
```

The MIT License