Logic, recursion, and programming

Dr. Giuseppe Maggiore
Dr. Giulia Costantini
Francesco Di Giacomo
Gerard van Kruiningen

November 26, 2015

Contents

Chapter 1

Introduction

For anyone who has been through the contemporary school system, many concepts have been studied so long ago and to such degrees of depth and repetition that they now feel natural. For example, numbers are something that we typically assume to be given for granted, and not something that requires further definition. Indeed, who in their right mind would consider the concept of natural numbers and their addition to need explaining?

A similar feeling is shared in the computer sciences: variables, `if`, `while`, functions, etc. are all concepts that have been seen so long ago and used so frequently that they are often given for granted.

In this book we will challenge this assumption from the perspective of logic applied within the realm of programming. We will define a logic programming language (which is also implemented, so all samples *can actually be run on a computer*). We will then use this language to define, from scratch, multiple basic concepts. We start from numbers, rebuilt from scratch, and then move on to basic computational concepts that we assemble into always more complex forms. The end result of this voyage will be a full-blown interpreter that understands simple imperative programs.

One side effect of such a way of thinking is a stronger understanding of the shared foundations of mathematics and informatics. There is a chance that **the use of logic when decomposing problems will help in writing software that is better thought out and over which it is far easier to reason and ensure correctness or other**

important properties.

1.0.1 Structure of the book

In the rest of the text we will: *i*) introduce informal logical reasoning in Section 1.1; *ii*) introduce our logic programming language in Section 1.2 and its internal mechanisms; *iii*) give a first example of logic in action to define basic boolean operators in Section 1.3;

1.1 An informal introduction to logic

The basic rules of logic are very simple. Logic is entirely defined as a way to manipulate symbols. These symbols are usually common signs such as numbers, letter, Greek letters, etc., but it is not forbidden to use any symbol we find useful or interesting. In this section, precisely with the goal of illustrating the *independence of logic from the symbols chosen*, we shall manipulate smileys and other icons. **The first step is thus choosing the symbols.** Logic is really the description of most (complex) dynamic processes that search for an answer in a complex space. This search may be a purely mathematical search, or it could be a program running. The dynamic process is defined in terms of a series of rules which are activated whenever we find a matching input. Thus, **the second step is choosing the rules that define our dynamic processes**.

After defining symbols and rules, things become interesting and we can try to use our system to find answers and process information. We usually start with a given proposition, which is the input of the whole process. A *proposition* is just a series of symbols, such as for example: `3 + 2 = 5` `of`   . We apply the rules to the proposition until we reach the desired answer, and we have also answered all of the intermediate questions that came to existance during the dynamic process.

A concrete example Let us consider a full example. Consider the symbols of our language to be:

- A smiley 

- A candle 🕯
- A tree 🌳
- A coffee cup ☕

We consider a proposition to be true if and only if we can process it until we reach a ☕. [1] Our rules are:

- **(G0)** A ☕ means we are done
- **(R1)** Two 🙂 followed by a 🌳, and then further followed by `r`, means that we will have to process `r` to find the answer
- **(R2)** Two 🕯 followed by a 🌳, and then further followed by `r`, means that we will have to process `r` to find the answer

Consider now the input proposition of
🙂 🙂 🌳 🕯 🕯 🌳 🙂 🙂 🌳 ☕

We begin by using **R2**, therefore obtaining:
🕯 🕯 🌳 🙂 🙂 🌳 ☕

We then use rule **R1**, therefore obtaining:
🙂 🙂 🌳 ☕

We then use rule **R2**, therefore obtaining:
☕

Now according to **G0** we are done. The proof is successfull, therefore we can conclude that 🙂 🙂 🌳 🕯 🕯 🌳 🙂 🙂 🌳 ☕ was **true within our system of rules**.

Consider now the new input proposition of
🙂 🙂 🌳 🙂 🙂 🌳 🕯 🌳 ☕

[1]Given the extreme importance of coffee in the diet of mathematicians and computer scientists, equating coffee with truth does not really seem that illogical a step

We begin by using **R2**, therefore obtaining:
🙂 🙂 🌳 🕯 🌳 ☕

We then use rule **R1**, therefore obtaining:
🕯 🌳 ☕

Unfortunately now we cannot apply rule **R1** again, because we have no 🙂 at the head of our proposition; we cannot apply rule **R2** because we have no 🕯 at the head of our proposition; and we can certainly not apply rule **G0** because there is no lonely ☕. If we cannot apply any of our rules, the process is *stuck*. This means that we cannot prove 🙂 🙂 🌳 🙂 🙂 🌳 🕯 🌳 ☕ with our rules, thus 🙂 🙂 🌳 🙂 🙂 🌳 🕯 🌳 ☕ was **not true within our system of rules**.

1.2 Inference systems

In this section we present a slightly more formal treatment of inference systems. Moreover, we give a clearer syntax for expressing operators and rules. The formalism which we will use, which is also a programming language, is *Meta-Casanova*. Meta-Casanova falls under the broader category of *logic programming languages*, but given its structure and some of its applications it can also be considered a *meta-compiler*. The latest release of the language can be downloaded from: `https://github.com/vs-team/metacompiler`.

1.2.1 Syntax for symbol definitions

Defining symbols in Meta-Casanova requires drawing a distinction: some symbols are considered *data*, while other symbols are considered *functions*. `Data` symbols are used to store and structure information, whereas `function` symbols are used to transform information.

Data symbols Data symbols are used to assemble *propositions* (or *expressions*), which are nested sequences of symbols.

The most basic definition of data looks like the following:

```
Data "NAME" : TYPE
```

where `NAME` is a sequence of alphanumeric characters and arithmetic operators that will now be bound to a new symbol, which will be useable whenever an expression cathegorized as `TYPE` is expected.

Examples of such a definition could be boolean truth-value symbols:

```
Data "TRUE" : Expr
Data "FALSE" : Expr
```

Some data symbols can be used to assemble complex expressions. The syntax for such a definition then becomes:

```
Data P_1 -> ... -> P_{N-1} -> "NAME" -> P_N -> ... P_{N+M} -> : TYPE
```

where `NAME` is the name of the symbol, `TYPE` indicates in what contexts this symbol can be used, $\mathtt{P}_1$ through $\mathtt{P}_{N-1}$ are the parameters that are expected left of the symbol, and $\mathtt{P}_N$ through $\mathtt{P}_{N+M}$ are the parameters that are expected right of the symbol.

For example, we might wish to have an expression that represents boolean disjunction (the so-called *or* operator):

```
Data Expr -> "|" -> Expr : Expr
```

With the definitions above we can write a proposition such as `TRUE | TRUE`, which would be perfectly valid. Associativity is by default to the right, thus `TRUE | TRUE | TRUE` is equivalent to `TRUE | (TRUE | TRUE)`. We can modify the default associativity by specifying explicitly whether `Associativity` is `Left` or `Right`:

```
Data Expr -> "|" -> Expr : Expr    Associativity Left
```

By default all data symbols have the same priority. Suppose that we want to define another way to combine boolean expressions. For example, we might wish to have an expression that represents boolean conjunction (the so-called *and* operator):

```
Data Expr -> "&" -> Expr : Expr
```

In this case there is some ambiguity as to the meaning of a composite expression that involves both and and or, such as `TRUE & FALSE | TRUE`: does this mean `TRUE & (FALSE | TRUE)` or `(TRUE & FALSE) | TRUE`? This ambiguity can be quite dangerous, because the two expressions have significantly different meanings.

In this case we can augment our definition by explicitly specifying a `Priority` of the operators, for example by saying:

```
Data Expr -> "|" -> Expr : Expr           Priority 10
Data Expr -> "&" -> Expr : Expr           Priority 20
```

So far we have defined `TRUE` and `FALSE` as `Expr`, thereby saying that concrete truth values are boolean expressions. On one hand, this is needed to be able to build a proposition such as `TRUE | FALSE`; on the other hand, this sounds a bit imprecise, because an expression is usually understood to be composite and we often call symbols such as `TRUE` and `FALSE` *values* with the caveat that they can be used wherever expressions are expected. This means that ideally we would like to define a data symbol like `TRUE` with one type, but then also making it so that propositions of this type can be used whenever proposition of another type is expected. This is a form of *subtyping*. Our language supports it. First we define the symbols with the new type `Value`:

```
Data "TRUE" : Value
Data "FALSE" : Value
```

If we stopped here, then `TRUE | FALSE` would not be legal anymore because `|` now expects `Expr`'s but we are giving it `Value`'s. To fix this, we connect `Value` to `Expr`:

```
Value is Expr
```

this means that any proposition of type `Value` (that is only `TRUE` and `FALSE`) will be allowed whenever a proposition of type `Expr` is expected (that is left and right of `|` and `&`).

Function symbols Function symbols are used to define transformations from propositions (combinations of data symbols) into new propositions. Functions are defined in two steps: first we **declare** the function, specifying its input parameters (which can come both left and right), and then we **define** the function, specifying its behaviour with a series of recursive definitions.

Functions are declared with the following syntax:

```
Func P1 -> ... P(N-1) -> "NAME" -> PN -> ... P(N+M) -> : TYPE => RETURN
```

which defines a function named `NAME`, with parameters $\mathtt{P}_1$ through $\mathtt{P}_{N-1}$ which are applied to the left of the function and parameters

$\texttt{P}_N$ through $\texttt{P}_{N+M}$. The function with its parameters applied is a proposition of type `TYPE`. Evaluation of the function will yield a result of type `RETURN`.

For example, we could define an `eval` function that takes as input a boolean expression of type `Expr`, and returns as input a boolean value of type `Value`:

```
Func "eval" -> Expr : Expr => Value
```

It will now be possible to write `eval (TRUE | TRUE)`. Of course so far we have only defined the acceptable *shape* of propositions, but we have not yet said anything about how computations happen.

1.2.2 Syntax and semantics of rules

Rules define how propositions that begin with a *functions* process the *data* that they have as parameters. The rules of an inference system are syntactically quite simple. The simplicity comes from the fact that a rule syntax is only made up of two operators and two additional concepts. The two operators are:

- the **horizontal bar** `------------`
- the **arrow** `=>`

Both horizontal bar and implication arrow can be read out loud as "therefore", as they both capture a form of implication. The horizontal bar separates the main implication from its premises, and thus operates at a higher level of precedence and abstraction with respect to the arrow. A single inference rule will consist of a series of **premises** which are separated from the **main proposition** by the horizontal bar:

```
PREMISE 1
PREMISE 2
...
PREMISE N
-----------------
MAIN PROPOSITION
```

Both individual premises and the main proposition are defined as the **input** sequence of symbols, separated from the **output sequence of symbols** by the implication arrow:

```
INPUT SYMBOLS => OUTPUT SYMBOLS
```

This means that a rule will look like:

```
I₁ => O₁
I₂ => O₂
⋮
Iₙ => Oₙ
-----------------
INPUT => OUTPUT
```

`INPUT` is the input of the rule. It is always defined as a function with a specified shape and value of its parameters. When the rule is fired, then its premises are evaluated from top to bottom; this means that first we evaluate $\texttt{I}_1$ to obtain $\texttt{O}_1$, then we move on to $\texttt{I}_2$ and $\texttt{O}_2$, until we reach the last premise. If all premises are successfully evaluated, then `OUTPUT` is the result of the rule, which is returned as the final result.

The simplest rules feature no premises, and directly specify how input and output are connected without any actual computation. An example of this is:

```
------------------
eval TRUE => TRUE
```

The rule above tells us that whenever we encounter `eval TRUE`, then we can directly return `TRUE`. Another example along the same lines would be:

```
--------------------
eval FALSE => FALSE
```

Sometimes a rule is more complex, in that we cannot directly determine its result from the input bu rather need some further computation. This is quite often the case with complex proposition which need to be further processed through the rule premises in order to determine the final result. For example, we could have:

```
eval a => TRUE
-------------------
eval (a|b) => TRUE
```

which processes an `or`-expression. Given an `or` of two boolean expressions `a` and `b`, we cannot directly determine if the whole `or` is true or false. What we do is specify, in the first premise, that we want to evaluate the first sub-expression by writing `eval a`. We also say that the expected output **must be** `TRUE`. Should `eval a` return something else than `TRUE`, then we would stop evaluation of this rule. If the output of the first premise indeed appears to be `TRUE`, then we will return `TRUE` as the output of the rule itself.

A question that should arise at this point is: what happens if `eval a` returns `FALSE`? Clearly we are not stuck, simply the rule above cannot provide us with the answer we are looking for. We will often need multiple rules to specify the whole behaviour of a complex function such as `eval`. In reference to our example, we give another rule that covers the opposite case:

```
eval a => FALSE
eval b => y
--------------------
eval (a|b) => y
```

The first premise evaluates the first sub-expression by writing `eval a`, this time with an expected output of `FALSE`. This rule thus succeeds whenever the other fails. Should `eval a` return something else than `FALSE`, then we would stop evaluation of this rule. If the output of the first premise indeed appears to be `FALSE`, then we will proceed to the evaluation of the second premise, `eval b`. It does not matter what for result we get from the second premise, so we simply say that this result will be called `y`, which we then further return as the output of the rule itself.

Program evaluation An aspect that is still unclear in our system is how rules are selected. Rules specify a sort of library of possible evaluations, but we still miss an important ingredient: the initial input. The initial input will determine the selection of rules, further recursively through the evaluation of premises, until an output is found or no further progress can be made.

Consider the boolean expressions example we have seen so far, which we recap here fully:

```
Func "eval" -> Expr : Expr => Value
Data Expr -> "|" -> Expr : Expr

Data "TRUE" : Value
```

```
Data "FALSE" : Value

Value is Expr


------------------
eval TRUE => TRUE

------------------
eval FALSE => FALSE


eval a => TRUE
-------------------
eval (a|b) => TRUE

eval a => FALSE
eval b => y
-------------------
eval (a|b) => y
```

Suppose we give our system an initial input of `eval (FALSE | TRUE)`, which becomes our *current proposition*. The first step, given any input, is to find **matching rules**. The process of matching looks for rules which input can be mapped to our current proposition. The candidate rule inputs are, in order of declaration:

```
eval TRUE
eval FALSE
eval (a|b)
eval (a|b)
```

we begin with the first input, and write it side-by-side with our current proposition:

```
eval      TRUE
eval (FALSE | TRUE)
```

Whereas `eval` matches, the parameters given to eval are clearly different and unrelated, so the first rule cannot (yet) be used.

We continue with the second input, and write it side-by-side with our current proposition:

```
eval      FALSE
eval (FALSE | TRUE)
```

Once again, `eval` matches, but its parameters do not.

We move on to the third rule:

```
eval (  a   |  b  )
eval (FALSE | TRUE)
```

Now things look promising. `eval` matches, and its first parameter is `|` in both cases. Since the rule specifies nothing further for the parameters of the `|`, simply giving them generic names like `a` and `b`, we can say that **the current proposition matches the rule, provided that `a=FALSE` and `b=TRUE`**. `a=FALSE` and `b=TRUE` is called a **binding**, in that it binds the variables of the rule input, which so far were unspecified, to parts of the current proposition. We can more easily represent this whole process graphically by putting the current proposition underneath the rule, with a question mark next to it to denote the fact that we are still working on it:

```
eval a => TRUE
----------------------------
eval (  a   |  b  ) => TRUE
eval (FALSE | TRUE)?
```

Since we now know that `a=FALSE` and `b=TRUE`, we can replace any occurrence of `a` and `b` as follows:

```
eval FALSE => TRUE
----------------------------
eval (FALSE | TRUE) => TRUE
eval (FALSE | TRUE)?
```

The process of replacing variables with their values as dictated by a binding is called **instancing**.

Now of course we need to make sure that all premises are valid, so we start with the first premise `eval FALSE => TRUE`. We take its input `eval FALSE`, and consider it to be the current proposition. We can represent this graphically by re-writing the current proposition above the premise we are evaluating:

```
eval FALSE?
eval FALSE => TRUE
----------------------------
eval (FALSE | TRUE) => TRUE
eval (FALSE | TRUE)?
```

We now repeat the matching process. We will represent this graphically by writing the rule we are considering above the current proposition to see if it matches. We start with the first rule:

```
-----------------
eval TRUE  => TRUE
eval FALSE?
eval FALSE => TRUE
```

```
----------------------------
eval (FALSE | TRUE) => TRUE
eval (FALSE | TRUE)?
```

Clearly this is no match, so we move on to the second rule:

```
------------------
eval FALSE => FALSE
eval FALSE?
eval FALSE => TRUE
----------------------------
eval (FALSE | TRUE) => TRUE
eval (FALSE | TRUE)?
```

We have a very evident match. Since the second rule immediately tells us what the result was (`FALSE`), we can take this result and write it next to the current proposition. We also delete the uppermost rule, because its evaluation is completed with result `FALSE`:

```
eval FALSE => FALSE
eval FALSE => TRUE
----------------------------
eval (FALSE | TRUE) => TRUE
eval (FALSE | TRUE)?
```

Now we have to match the actual output of the premise to the expected output, that is we have to see if we can match `FALSE` and `TRUE`. Unfortunately this is not the case, so we have to abort evaluation of the whole rule and go back to the original current proposition of `eval (FALSE | TRUE)`. Now we know from before that:

- the first rule does not match
- the second rule does not match
- the third rule matches, but gets stuck at the first premise

so we can proceed with the fourth rule:

```
eval a => FALSE
eval b => y
----------------------------
eval (  a   |  b  ) => TRUE
eval (FALSE | TRUE)?
```

We know the binding of `a=FALSE` and `b=TRUE`, so we obtain a workable instance by replacing `a` and `b` in the rule:

```
eval FALSE => FALSE
eval TRUE => y
----------------------------
eval (FALSE | TRUE) => y
eval (FALSE | TRUE)?
```

We evaluate the first premise, so we duplicate its input to signal that we now have a new current proposition:

```
eval FALSE?
eval FALSE => FALSE
eval TRUE => y
----------------------------
eval (FALSE | TRUE) => y
eval (FALSE | TRUE)?
```

we try to match the first rule to the current proposition, but with no success:

```
------------------
eval TRUE => TRUE
eval FALSE?
eval FALSE => FALSE
eval TRUE => y
----------------------------
eval (FALSE | TRUE) => y
eval (FALSE | TRUE)?
```

we then try to match the second rule to the current proposition, this time succesfully:

```
------------------
eval FALSE => FALSE
eval FALSE?
eval FALSE => FALSE
eval TRUE => y
----------------------------
eval (FALSE | TRUE) => y
eval (FALSE | TRUE)?
```

the current proposition results in `FALSE`, so we delete the upper rule (it is done evaluating and we do not need it anymore) and propagate its result down:

```
eval FALSE => FALSE
eval FALSE => FALSE
eval TRUE => y
----------------------------
eval (FALSE | TRUE) => y
eval (FALSE | TRUE)?
```

we must now match the actual result of the premise with the actual result of the premise. Both are `FALSE`, so matching is trivial. We delete the upper lines because they are done evaluating, and have no further use:

```
eval TRUE => y
----------------------------
eval (FALSE | TRUE) => y
eval (FALSE | TRUE)?
```

We must now evaluate the last premise, `eval TRUE => y`:

```
eval TRUE?
eval TRUE => y
----------------------------
eval (FALSE | TRUE) => y
eval (FALSE | TRUE)?
```

We try the first rule, which succesfully matches and immediately returns `TRUE`:

```
------------------
eval TRUE => TRUE
eval TRUE?
eval TRUE => y
----------------------------
eval (FALSE | TRUE) => y
eval (FALSE | TRUE)?
```

We propagate the result down next to the premise input:

```
eval TRUE => TRUE
eval TRUE =>  y
----------------------------
eval (FALSE | TRUE) => y
eval (FALSE | TRUE)?
```

Now we wish to propagate the result down to the premise, and to do this we need to perform another matching. Fortunately this matching succeeds, with binding `y=TRUE`. Now that we have a value for `y`, we can replace all its occurrences with the value:

```
eval TRUE => TRUE
eval TRUE => TRUE
----------------------------
eval (FALSE | TRUE) => TRUE
eval (FALSE | TRUE)?
```

We delete the premises above, since they have no further use, and voilá: we know the result of the evaluation of the initial proposition:

```
---------------------------
eval (FALSE | TRUE) => TRUE
eval (FALSE | TRUE)?
```

This result is also what we expected, to our satisfaction.

1.3 A first example: boolean expressions

In this chapter we will focus again on boolean expressions. Be careful: instead of just repeating the same sample of the previous chapter, what we will be really doing is focus on design strategies and concepts. We will thus begin with a few notes on general design of rules within our system, with a focus on how to translate such designs into an actual implementation. We will then analyse runtime properties, in particular performance, and discuss a first batch of possible improvements.

1.4 Implementation

We will now build a full logic program with our language. The first such program that we will see is, quite fittingly, [2] a program for the definition and evaluation of boolean expressions.

Symbols The basic symbols that we need represents the boolean values of `TRUE` and `FALSE`, and they both have type `Value`:

```
Data "TRUE" : Value
Data "FALSE" : Value
```

To make things more articulated, we define operators to build boolean expressions. Boolean expressions are (recursively) defined as:

1. boolean values;
2. negation of a boolean expression (commonly known as "not");

[2] Boolean logic is consider a form of logic itself, which fits snugly within our own implementation of an interpretation of logic. Recursion can be quite fun, as we will see in one of the last chapters.

3. disjunction of boolean expressions (commonly known as "or");

4. conjunction of boolean expressions (commonly known as "and").

The first item is simply defined by connecting the type `Value` to the type `Expr` of boolean expressions. We specify this as follows:

```
Value is Expr
```

We will use the typical naming convention of programming languages for boolean operators:

- `|` for `or`
- `&` for `and`
- `!` for `not`

With this convention, composite boolean expressions (which are all of type `Expr`) are defined as:

```
Data Expr -> "|" -> Expr : Expr                    Priority 10
Data Expr -> "&" -> Expr : Expr                    Priority 20
Data "!" -> Expr : Expr                            Priority 30
```

The definition of symbols with a priority is quite important. The common intuition of composite boolean expressions dictates that `|` is comparable to arithmetic addition, `&` is comparable to arithmetic multiplication, and `!` is comparable to arithmetic negation. For this reason, we expect that an expression such as `TRUE | !FALSE & FALSE` will be implied as equivalent to `(TRUE | ((!FALSE) & FALSE))`. This means that through our priorities then:

- `!` is always the most deeply nested
- `&` is the second deepest
- `|` comes last

In order to determine the `Value` of an `Expr`, we define the `eval` function. This function is what will give the usual behaviour of boolean expressions to our specific implementation:

```
Func "eval" -> Expr : Expr => Value
```

Designing rules So far our definitions only specify **how to build valid expressions**. As much as our symbols can be used to build propositions that may look like boolean expressions, in reality no such thing is implemented yet. For example, nobody (besides common sense, which is quite a weak defense) stops us from letting `&` behave like an `or` instead of an `and`. We shall now carefully define computations around our symbols, through the definition of rules. The meaning of our our symbols will only be determined by how the behave in our computations, not just by their names. A good implementation obviously keeps the intuition evoked by the names in sync with the actual behaviour, but a bad implementation may obviously exist.

The first, obvious question that we must answer is *how do we design rules?*. The basic idea is that multiple rules will refer to a single function, and that each rule will cover some different aspects of the function.

The first division in different rules is based on the different forms of the input of the function. Our `eval` function, for example, will need to be able to process input that looks like all possible valid boolean expressions. Thus, `eval` will require:

- some rules for basic values such as `TRUE` and `FALSE`
- some rules for negation `!`
- some rules for disjunction `|`
- some rules for conjunction `&`

After identifying the possible inputs, we have to populate the premises of our rules. To do so, we always start with a description[3] of what the rule will do in natural language. We will then translate this description into premises and an output. Sometimes, rules will contain decisions that are reminiscent of conditionals. In this case, we will need multiple rules to cover that case.

Of course, functions in our logical framework are practically always **recursive**, given the complete lack of iterative operations such as `while` and `for`. Recursion means that:

[3]This is often called pseudo-code

- we start with a proposition, which is recognised as the input
- in the premises we process sub-propositions which are always smaller than the input proposition
- when the input proposition is tiny enough in size we can output the answer directly (the so-called **base case**)
- We reassemble partial results from the premises into the final result

Rules implementation The only rules that we can define involve the `eval` function, because there are no other functions that we have defined. The first two rules trivially specify that when we reach the evaluation of `TRUE` or `FALSE`, then we do not need to further proceed:

```
------------------
eval TRUE => TRUE

--------------------
eval FALSE => FALSE
```

These rules, which process the tiniest form of input, are our **base case**.

If we reach the evaluation of the negation of some expression `a`, then we will evaluate `a`. If the evaluation of `a` returns `TRUE`, then the evaluation of the negation of `a` returns `FALSE`:

```
eval a => TRUE
-----------------
eval !a => FALSE
```

Notice that, in order to evaluate `!a`, what we have done is evaluate `a` (which is smaller than `!a` for the simple fact that it misses the `!` at the beginning), and then we reassemble the final output based on the output of the first premise.

Similarly, if we reach the evaluation of the negation of some expression `a`, and evaluation of `a` returns `FALSE`, then the evaluation of the negation of `a` returns `TRUE`:

```
eval a => FALSE
----------------
eval !a => TRUE
```

Note that making choices is simply defined by having different rules. We will need to use this strategy quite often, in that our language features no explicit conditional operators such as `if-then-else`, and so multiple rules with different patterns are our only way to define conditionals and choices.

When evaluating the disjunction of two expressions `a` and `b`, we try to evaluate `a`: *i*) if `a` evaluates to `TRUE`, then there is no need to further evaluate `b` and we can directly return `TRUE`; *ii*) if `a` evaluates to `FALSE`, then we evaluate `b` and return whatever result of its evaluation.

```
eval a => TRUE
-------------------
eval (a|b) => TRUE

eval a => FALSE
eval b => y
-----------------
eval (a|b) => y
```

When evaluating the conjunction of two expressions `a` and `b`, we try to evaluate `a`: *i*) if `a` evaluates to `FALSE`, then there is no need to further evaluate `b` and we can directly return `FALSE`; *ii*) if `a` evaluates to `TRUE`, then we evaluate `b` and return whatever result of its evaluation.

```
eval a => FALSE
--------------------
eval (a&b) => FALSE

eval a => TRUE
eval b => y
----------------
eval (a&b) => y
```

Example run Consider now the evaluation of an expression such as `eval (FALSE | !(TRUE & FALSE))`. Instead of trying out all possible rules, we can observe that this expression is an instance of `eval (a|b)`, therefore we only need to investigate the applicability of `or`-rules. We start with the first `or`-rule:

```
eval a => TRUE
--------------------------------
eval (  a   |         b) => TRUE
eval (FALSE | !(TRUE & FALSE))?
```

This rule matches with binding `a=FALSE` and `b=!(TRUE & FALSE)`, so we can use this rule. We start by replacing the bound values to the variables:

```
eval FALSE => TRUE
---------------------------------------
eval (FALSE | !(TRUE & FALSE)) => TRUE
eval (FALSE | !(TRUE & FALSE))?
```

We evaluate now the first premise[4]:

```
eval FALSE?
eval FALSE => TRUE
---------------------------------------
eval (FALSE | !(TRUE & FALSE)) => TRUE
eval (FALSE | !(TRUE & FALSE))?
```

Clearly the first premise matches with the second basic rule, which will return `FALSE`:

```
------------------
eval FALSE => FALSE
eval FALSE?
eval FALSE => TRUE
---------------------------------------
eval (FALSE | !(TRUE & FALSE)) => TRUE
eval (FALSE | !(TRUE & FALSE))?
```

We can take the result of the rule and set it next to the input of the second premise which we were evaluating, while also deleting the upper rule because its evaluation is completed and its result has been propagated down:

```
eval FALSE => FALSE
eval FALSE => TRUE
---------------------------------------
eval (FALSE | !(TRUE & FALSE)) => TRUE
eval (FALSE | !(TRUE & FALSE))?
```

Now we try to match the result that we got (`FALSE`) and the result that we expected (`TRUE`), but unfortunately they do not match. This means that the first `or`-rule is not in state to give us an answer, so we have to move on to the second `or`-rule:

```
eval a => FALSE
eval b => y
-------------------
eval (   a   |         b        ) => y
eval (FALSE | !(TRUE & FALSE))?
```

[4]The astute reader already noticed that this will not get us very far!

The matching is clearly successful, since the input of this rule is precisely the same as the input of the previous: `a=FALSE` and `b=!(TRUE & FALSE)`. We can replace the variables according to the binding, therefore obtaining:

```
eval FALSE => FALSE
eval !(TRUE & FALSE) => y
------------------------------------
eval (FALSE | !(TRUE & FALSE)) => y
eval (FALSE | !(TRUE & FALSE))?
```

We now move on to the first premise:

```
eval FALSE?
eval FALSE => FALSE
eval !(TRUE & FALSE) => y
------------------------------------
eval (FALSE | !(TRUE & FALSE)) => y
eval (FALSE | !(TRUE & FALSE))?
```

We can apply the second basic rule, which immediately tells us that the result is `FALSE`:

```
--------------------
eval FALSE => FALSE
eval FALSE?
eval FALSE => FALSE
eval !(TRUE & FALSE) => y
------------------------------------
eval (FALSE | !(TRUE & FALSE)) => y
eval (FALSE | !(TRUE & FALSE))?
```

We propagate the result (`FALSE`) next to the premise input, and delete the upper rule because we are done with it:

```
eval FALSE => FALSE
eval FALSE => FALSE
eval !(TRUE & FALSE) => y
------------------------------------
eval (FALSE | !(TRUE & FALSE)) => y
eval (FALSE | !(TRUE & FALSE))?
```

Now we are done with the first premise, and so we match the expected result and the result we got. Both being `FALSE`, the match is successful. We simply delete the premise, and all traces of its execution since we do not need it anymore:

```
eval !(TRUE & FALSE) => y
------------------------------------
eval (FALSE | !(TRUE & FALSE)) => y
eval (FALSE | !(TRUE & FALSE))?
```

It is time to evaluate the second premise, `eval !(TRUE & FALSE)`. Since it is clearly only an instance of a `not`-rule, we can write the first `not`-rule above the first premise without explicitly listing all the failed matches with the other rules:

```
eval a => TRUE
------------------------------
eval !      a         => FALSE
eval !(TRUE & FALSE)?
eval !(TRUE & FALSE) => y
------------------------------------
eval (FALSE | !(TRUE & FALSE)) => y
eval (FALSE | !(TRUE & FALSE))?
```

The binding is clearly successful for `a=TRUE & FALSE`, so we can proceed with the substitution of the bound variables:

```
eval (TRUE & FALSE) => TRUE
------------------------------
eval !(TRUE & FALSE) => FALSE
eval !(TRUE & FALSE)?
eval !(TRUE & FALSE) => y
------------------------------------
eval (FALSE | !(TRUE & FALSE)) => y
eval (FALSE | !(TRUE & FALSE))?
```

We can now proceed with the evaluation of the first premise `eval (TRUE & FALSE)` of the `not`-rule:

```
eval (TRUE & FALSE)?
eval (TRUE & FALSE) => TRUE
------------------------------
eval !(TRUE & FALSE) => FALSE
eval !(TRUE & FALSE)?
eval !(TRUE & FALSE) => y
------------------------------------
eval (FALSE | !(TRUE & FALSE)) => y
eval (FALSE | !(TRUE & FALSE))?
```

The first premise is an instance of any of the two `and`-rules. Let us begin by trying the first out:

```
eval a => FALSE
------------------------------
eval (  a   &    b   ) => FALSE
eval (TRUE & FALSE)?
eval (TRUE & FALSE) => TRUE
------------------------------
eval !(TRUE & FALSE) => FALSE
eval !(TRUE & FALSE)?
eval !(TRUE & FALSE) => y
------------------------------------
eval (FALSE | !(TRUE & FALSE)) => y
```

```
eval (FALSE | !(TRUE & FALSE))?
```

The match is successful with a binding of `a=TRUE` and `b=FALSE`, so we instance the uppermost rule to:

```
eval TRUE => FALSE
------------------------------
eval (TRUE & FALSE) => FALSE
eval (TRUE & FALSE)?
eval (TRUE & FALSE) => TRUE
------------------------------
eval !(TRUE & FALSE) => FALSE
eval !(TRUE & FALSE)?
eval !(TRUE & FALSE) => y
------------------------------------
eval (FALSE | !(TRUE & FALSE)) => y
eval (FALSE | !(TRUE & FALSE))?
```

We now have to work on the first premise, but to save some obvious passages we can immediately observe that the premise `eval TRUE => FALSE` will fail, so we skip these (hopefully now) obvious steps and go back to trying the alternate rule for `and`:

```
eval a => TRUE
eval b => y
-------------------------
eval (  a  &   b  ) => y
eval (TRUE & FALSE)?
eval (TRUE & FALSE) => TRUE
------------------------------
eval !(TRUE & FALSE) => FALSE
eval !(TRUE & FALSE)?
eval !(TRUE & FALSE) => y
------------------------------------
eval (FALSE | !(TRUE & FALSE)) => y
eval (FALSE | !(TRUE & FALSE))?
```

The rule matches, with a binding of `a=TRUE` and `b=FALSE`; we proceed with instancing, therefore obtaining:

```
eval TRUE => TRUE
eval FALSE => y
-------------------------
eval (TRUE & FALSE) => y
eval (TRUE & FALSE)?
eval (TRUE & FALSE) => TRUE
------------------------------
eval !(TRUE & FALSE) => FALSE
eval !(TRUE & FALSE)?
eval !(TRUE & FALSE) => y
------------------------------------
eval (FALSE | !(TRUE & FALSE)) => y
eval (FALSE | !(TRUE & FALSE))?
```

Now we can use the very first rule of our system to succesfully solve the first premise:

```
-------------------
eval TRUE => TRUE
eval TRUE => TRUE
eval FALSE => y
-------------------------
eval (TRUE & FALSE) => y
eval (TRUE & FALSE)?
eval (TRUE & FALSE) => TRUE
------------------------------
eval !(TRUE & FALSE) => FALSE
eval !(TRUE & FALSE)?
eval !(TRUE & FALSE) => y
------------------------------------
eval (FALSE | !(TRUE & FALSE)) => y
eval (FALSE | !(TRUE & FALSE))?
```

We delete the upper rule and the now solved premise, and move on to `eval FALSE => y`:

```
eval FALSE => y
-------------------------
eval (TRUE & FALSE) => y
eval (TRUE & FALSE)?
eval (TRUE & FALSE) => TRUE
------------------------------
eval !(TRUE & FALSE) => FALSE
eval !(TRUE & FALSE)?
eval !(TRUE & FALSE) => y
------------------------------------
eval (FALSE | !(TRUE & FALSE)) => y
eval (FALSE | !(TRUE & FALSE))?
```

Thanks to the second rule of the whole system the uppermost premise is easily solved:

```
--------------------
eval FALSE => FALSE
eval FALSE => y
-------------------------
eval (TRUE & FALSE) => y
eval (TRUE & FALSE)?
eval (TRUE & FALSE) => TRUE
------------------------------
eval !(TRUE & FALSE) => FALSE
eval !(TRUE & FALSE)?
eval !(TRUE & FALSE) => y
------------------------------------
eval (FALSE | !(TRUE & FALSE)) => y
eval (FALSE | !(TRUE & FALSE))?
```

We now have to match the obtained result `FALSE` with the expected result `y`. This is easily achieved with a binding of `y=FALSE`. We should now instance our system with the binding `y=FALSE`, but at first this looks problematic: which of the `y`'s do we need to replace with `FALSE`? Clearly not all of them, but how do we know? A detail that we have so far given for granted is that variables have a limited **scope**, that is they cannot reach beyond the bottom of their rule. So for all practical purposes, the system above is equivalent to:

```
--------------------
eval FALSE => FALSE
eval FALSE => y$_1$
-------------------------
eval (TRUE & FALSE) => y$_1$
eval (TRUE & FALSE)?
eval (TRUE & FALSE) => TRUE
------------------------------
eval !(TRUE & FALSE) => FALSE
eval !(TRUE & FALSE)?
eval !(TRUE & FALSE) => y$_2$
------------------------------------
eval (FALSE | !(TRUE & FALSE)) => y$_2$
eval (FALSE | !(TRUE & FALSE))?
```

and our current binding would have been that $\mathtt{y}_1$ `= FALSE`. We will not explicitly disambiguate variables with the same name, since their scope cannot escape the natural boundaries of rules.

Instantiation thus results, unambiguously, in:

```
--------------------
eval FALSE => FALSE
eval FALSE => FALSE
-----------------------------
eval (TRUE & FALSE) => FALSE
eval (TRUE & FALSE)?
eval (TRUE & FALSE) => TRUE
------------------------------
eval !(TRUE & FALSE) => FALSE
eval !(TRUE & FALSE)?
eval !(TRUE & FALSE) => y
------------------------------------
eval (FALSE | !(TRUE & FALSE)) => y
eval (FALSE | !(TRUE & FALSE))?
```

The upper two rules are now useless, so we delete them and propagate the result to the premise `eval TRUE & FALSE` which was waiting:

```
eval (TRUE & FALSE) => FALSE
eval (TRUE & FALSE) => TRUE
------------------------------
```

```
eval !(TRUE & FALSE) => FALSE
eval !(TRUE & FALSE)?
eval !(TRUE & FALSE) => y
------------------------------------
eval (FALSE | !(TRUE & FALSE)) => y
eval (FALSE | !(TRUE & FALSE))?
```

We must now try and match the expected result of `TRUE` with the obtained result `FALSE`, but alas this cannot be done. This means that we now have to go back to the evaluation of the much earlier premise `eval !(TRUE & FALSE)`:

```
eval !(TRUE & FALSE)?
eval !(TRUE & FALSE) => y
------------------------------------
eval (FALSE | !(TRUE & FALSE)) => y
eval (FALSE | !(TRUE & FALSE))?
```

We must thus arm ourselves with a bit of patience and follow the system as it tries the second rule for `not`, the rule which expects a result of `FALSE`:

```
eval a => FALSE
----------------------
eval !      a => TRUE
eval !(TRUE & FALSE)?
eval !(TRUE & FALSE) => y
------------------------------------
eval (FALSE | !(TRUE & FALSE)) => y
eval (FALSE | !(TRUE & FALSE))?
```

Matching succeeds with binding `a=TRUE & FALSE`, which results in the following instance:

```
eval (TRUE & FALSE) => FALSE
-----------------------------
eval !(TRUE & FALSE) => TRUE
eval !(TRUE & FALSE)?
eval !(TRUE & FALSE) => y
------------------------------------
eval (FALSE | !(TRUE & FALSE)) => y
eval (FALSE | !(TRUE & FALSE))?
```

We have to solve the first premise of `eval (TRUE & FALSE) => FALSE`, for which we try the first `and`-rule:

```
eval a => FALSE
-----------------------------
eval (  a   &   b  ) => FALSE
eval (TRUE & FALSE)?
eval (TRUE & FALSE) => FALSE
-----------------------------
```

```
eval !(TRUE & FALSE) => TRUE
eval !(TRUE & FALSE)?
eval !(TRUE & FALSE) => y
------------------------------------
eval (FALSE | !(TRUE & FALSE)) => y
eval (FALSE | !(TRUE & FALSE))?
```

Matching results in binding `a=TRUE` and `b=FALSE`, and so we can proceed with instance:

```
eval TRUE => FALSE
----------------------------
eval (TRUE & FALSE) => FALSE
eval (TRUE & FALSE)?
eval (TRUE & FALSE) => FALSE
----------------------------
eval !(TRUE & FALSE) => TRUE
eval !(TRUE & FALSE)?
eval !(TRUE & FALSE) => y
------------------------------------
eval (FALSE | !(TRUE & FALSE)) => y
eval (FALSE | !(TRUE & FALSE))?
```

Once again in the interest of brevity we skip the couple of steps that would show how `eval TRUE` actually results in `TRUE`, which cannot be matched with `FALSE`, and roll-back the chosen `and`-rule which clearly did not work:

```
eval (TRUE & FALSE)?
eval (TRUE & FALSE) => FALSE
----------------------------
eval !(TRUE & FALSE) => TRUE
eval !(TRUE & FALSE)?
eval !(TRUE & FALSE) => y
------------------------------------
eval (FALSE | !(TRUE & FALSE)) => y
eval (FALSE | !(TRUE & FALSE))?
```

We now try the second `and`-rule:

```
eval a => TRUE
eval b => y
-------------------------
eval (  a  &   b  ) => y
eval (TRUE & FALSE)?
eval (TRUE & FALSE) => FALSE
----------------------------
eval !(TRUE & FALSE) => TRUE
eval !(TRUE & FALSE)?
eval !(TRUE & FALSE) => y
------------------------------------
eval (FALSE | !(TRUE & FALSE)) => y
eval (FALSE | !(TRUE & FALSE))?
```

Again we have a positive match `a=TRUE` and `b=FALSE`, so we move forward with instance:

```
eval TRUE => TRUE
eval FALSE => y
-------------------------
eval (TRUE & FALSE) => y
eval (TRUE & FALSE)?
eval (TRUE & FALSE) => FALSE
-----------------------------
eval !(TRUE & FALSE) => TRUE
eval !(TRUE & FALSE)?
eval !(TRUE & FALSE) => y
------------------------------------
eval (FALSE | !(TRUE & FALSE)) => y
eval (FALSE | !(TRUE & FALSE))?
```

The upper premise is trivially true thanks to the basic rule for the evaluation of `TRUE`, so we omit its evaluation steps for practicality. We now have to evaluate another simple premise, `eval FALSE`:

```
eval FALSE => y
-------------------------
eval (TRUE & FALSE) => y
eval (TRUE & FALSE)?
eval (TRUE & FALSE) => FALSE
-----------------------------
eval !(TRUE & FALSE) => TRUE
eval !(TRUE & FALSE)?
eval !(TRUE & FALSE) => y
------------------------------------
eval (FALSE | !(TRUE & FALSE)) => y
eval (FALSE | !(TRUE & FALSE))?
```

Once again we omit how, thanks to one of the basic rules, we get to:

```
eval FALSE => FALSE
eval FALSE => y
-------------------------
eval (TRUE & FALSE) => y
eval (TRUE & FALSE)?
eval (TRUE & FALSE) => FALSE
-----------------------------
eval !(TRUE & FALSE) => TRUE
eval !(TRUE & FALSE)?
eval !(TRUE & FALSE) => y
------------------------------------
eval (FALSE | !(TRUE & FALSE)) => y
eval (FALSE | !(TRUE & FALSE))?
```

Matching the obtained and expected results gives us a binding of `y=FALSE`, which we instantiate into:

```
eval FALSE => FALSE
eval FALSE => FALSE
-------------------------
eval (TRUE & FALSE) => FALSE
eval (TRUE & FALSE)?
eval (TRUE & FALSE) => FALSE
-----------------------------
eval !(TRUE & FALSE) => TRUE
eval !(TRUE & FALSE)?
eval !(TRUE & FALSE) => y
------------------------------------
eval (FALSE | !(TRUE & FALSE)) => y
eval (FALSE | !(TRUE & FALSE))?
```

Now we can remove the upper premises and rule, which evaluation is completed, and propagate the result `FALSE` to the premise `eval TRUE & FALSE`:

```
eval (TRUE & FALSE) => FALSE
eval (TRUE & FALSE) => FALSE
-----------------------------
eval !(TRUE & FALSE) => TRUE
eval !(TRUE & FALSE)?
eval !(TRUE & FALSE) => y
------------------------------------
eval (FALSE | !(TRUE & FALSE)) => y
eval (FALSE | !(TRUE & FALSE))?
```

The upper premises trivially match, so we are done with them and can delete them:

```
eval !(TRUE & FALSE) => TRUE
eval !(TRUE & FALSE)?
eval !(TRUE & FALSE) => y
------------------------------------
eval (FALSE | !(TRUE & FALSE)) => y
eval (FALSE | !(TRUE & FALSE))?
```

Now it is the turn of `TRUE` to be propagated down:

```
eval !(TRUE & FALSE) => TRUE
eval !(TRUE & FALSE) => y
------------------------------------
eval (FALSE | !(TRUE & FALSE)) => y
eval (FALSE | !(TRUE & FALSE))?
```

We perform the final match between result and expectation, obtaining `y=TRUE`. We perform the last instancing of the whole process, resulting in:

```
eval !(TRUE & FALSE) => TRUE
eval !(TRUE & FALSE) => TRUE
```

```
-----------------------------------
eval (FALSE | !(TRUE & FALSE)) => TRUE
eval (FALSE | !(TRUE & FALSE))?
```

We delete the solved premises and perform the very last propagation, which tells us the final answer according to which (just like we should have expected), `eval (FALSE | !(TRUE & FALSE)) => TRUE`.

Backtracking, performance, and avoiding repetition The process we have just explored in painstaking detail was chosen because of its emblematicity. Specifically, the most interesting part is that the implementation that we used so far for boolean expressions is both **correct** and **very slow**. The performance of the system is so poor because rules, as they are specified now, can cause a lot of repeated evaluations. Consider for a moment only the rules for `not`:

```
eval a => TRUE
-----------------
eval !a => FALSE

eval a => FALSE
----------------
eval !a => TRUE
```

Even though the definition is correct in the sense of following our common expectation in the behaviour of boolean expressions, should the evaluation of `a` within the first rule result in `FALSE`, then the second rule would **evaluate `a` again**! This is wasteful, mostly because the evaluation of `a` will not change result. The very same issue is visible in both the `or`-rules and the `and`-rules, because they both risk repeating the evaluation of `a` twice. The effect that this has on the number of steps that the system might perform is dramatic. Consider a proposition of depth $N + 1$[5]; each of the internal N symbols of

[5] depth of a proposition is defined as the maximum number of nested elements, so in our current sample:

- `eval TRUE` has a depth of 2
- `eval (!TRUE)` has a depth of 3
- `eval (!TRUE & !(TRUE | FALSE))` has a depth of 5
- etc.

the proposition might, in the worst case, be evaluated twice. Two evaluations of each symbol causes a re-evaluation of all of its internal symbols, resulting in a staggering maximum number of evaluations of 2^N. For a relatively tiny proposition of 21 nested symbols we would get up to $2^2 0 = 1048576$ steps.

The situation is even worse. If we are lucky in the choice of proposition, then each rule will succeed immediately, without need for repeated evaluations. This is particularly bad from an engineering standpoint, because we have build a definition where for no evident reason some propositions will be blazing fast to evaluate while others will be obsecenely slow.

This is clearly a bad start, but with a bit of careful redesign we can fix this issue. For example, we could observe that evaluating a `not`-proposition such as `!a`, we must in both cases evaluate `a`, and subsequently *flip* the result. In this new definition there is clearly no implied need for a repeated evaluation of `a`. We define a new `flip` function which does not handle expressions in favour of exclusively working with values:

```
Func "flip" -> Value : Expr => Value
```

True to its name, `flip` changes a `TRUE` value into a `FALSE`, and viceversa:

```
-------------------
flip TRUE => FALSE

-------------------
flip FALSE => TRUE
```

We can now rewrite the `not`-rules into a single rule that relies on `flip` to perform most of the internal logic of the `not`-symbol:

```
eval a => y
flip y => y'
--------------
eval !a => y'
```

`or`-rules can undergo a comparable treatment. Given a proposition such as `a|b`, instead of risking repeating the evaluation of `a`, we can give both the result of the evaluation of `a` and `b` to a *shortcutOnTrue* function which will only evaluate `b` if the first parameter was `FALSE`:

```
Func "shortcutOnTrue" -> Value -> Expr : Expr => Value
```

```
...

eval a => y
shortcutOnTrue y b => y'
-------------------------
eval (a|b) => y'

------------------------------
shortcutOnTrue TRUE b => TRUE

eval b => y
----------------------------
shortcutOnTrue FALSE b => y
```

Last, but not least, we optimize the definition of `and`-rules with a reverse shortcut that skips evaluating the second term if the first evaluated to `FALSE`:

```
Func "shortcutOnFalse" -> Value -> Expr : Expr => Value

...

eval a => y
shortcutOnFalse y b => y'
-------------------
eval (a&b) => y'

---------------------------------
shortcutOnFalse FALSE b => FALSE

eval b => y
----------------------------
shortcutOnFalse TRUE b => y
```

Consider now a proposition of depth $N+1$. There is no repetition, therefore the maximum number of evaluations will have to unravel the proposition to its full depth, thus performing no more than N steps. This is a far cry from the horrible 2^N.

The new implementation is also far more predictable. Even though the new `shortcut*` functions might cause a few steps to be *skipped*, we will never get terms of comparable lengths performing million of steps in one case and dozens in the other. This means that terms of similar length will only have a marginal, proportionate difference in evaluation times, making this implementation at the same time correct in the result it gives, and both fast and unsurprising in the time it takes to give them.