Justification Proof Search Implementation in Python

Bachelorarbeit

der Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

vorgelegt von

Judith Fuog

2014

Leiter der Arbeit:
Prof. Dr. Thomas Studer
Institut für Informatik und angewandte Mathematik

Abstract

todo

Contents

1 Introduction **1**

1.1 Motivation . . . 1

1.2 Goal . . . 1

1.3 Overview . . . 1

2 Justification Logic **3**

2.1 Background . . . 3

2.2 Rules and Definitions . . . 3

3 A Divide and Conquer Algorithm **5**

3.1 Core Idea . . . 5

3.2 Divide . . . 5

3.2.1 Atomize . . . 6

3.3 Conquer . . . 8

3.3.1 Get Must . . . 9

3.3.2 X-Wilds in the Configuration Table . . . 9

3.3.3 Merge . . . 11

4 Implementation **13**

4.1 Model Overview . . . 13

4.2 Operation Syntax Tree . . . 14

4.3 Important Methods . . . 15

4.3.1 Tree.musts . . . 15

4.3.2 Tree.compare . . . 16

4.3.3 ProofSearch.apply_conditions . . . 16

4.3.4 ProofSearch.merge . . . 18

4.4 Tests . . . 19

5 Example **20**

5.1 Initialization . . . 20

5.2 Walking in Trees: Atomize . . . 20

5.2.1 Bangs . . . 21

5.3 Looking up and merging . . . 22

5.3.1 Musts . . . 22

5.3.2 Configurations and Conditions . . . 23

5.3.3 Merging of the Config Table . . . 25

5.4 The Final Result . . . 26

6 Results **28**
6.1 Application 28
6.2 Enhancement 28

Bibliography **28**

Chapter 1

Introduction

1.1 Motivation

Todo: What's the motivation behind it? Not **MY** motivation, but the scientific motivation.

1.2 Goal

The initial goal was to extend an existing proof search engine Z3 [Microsoft Research] such that it could also handle Justification Logic. Deeper investigation into that project revealed that to make it handle also Justification Logic the given interface in Python would not work. Instead it would have to be look into the core of the programm which is written in C. The expenses it would require to get so much deeper into the material that the actual indented work would be only secondary. So instead of extending Microsoft Research project the actual goal changed to implementing a simplified proof search for Justification Logic. It meant that the implementation would be easier since it does not depend on anything else anymore. As a downside a lot of the functionallity that was hoped go get from Z3 would have to be implemented as well or left out.

The program should satisfy to following conditions:

Input The formula to be proven as well as a list of formulas needed for the proof is given as string. It may be presumed that the string is exactly formatted in the way needed. It must not be checked for syntax error or general typing mistakes.

Output A simple *True* or *False* for the provability of the formula. [1]

1.3 Overview

The second chapter will present a short introduction to Justification Logic, but will go only as deep as needed to understand the problem as well as the develop algorithm.

[1] Optional the output could give information about how a proof was found if the formula is provable.

The heart of the third chapter will introduce the algorithm used in the program. Since this thesis concerns itself more with the practical side of implementation and not the theoretical side of mathematical logic theory there will be little proof here but instead many example to show how the algorithm works.

Todo: Chapter about implementation

Todo: Chapter about Examle

Finally the last chapter will discuss the result of the work and give some ideas about how the work of a Justification Logic proof search implementation could be improved.

Chapter 2

Justification Logic

The theory of Justification Logic as it is used here requires little knowledge of the wide fields of Modal Logic apart from some very basic knowledage about logic theory. For the purpose of this proof search a few basic rules and definitions are sufficient to provide the needed knowledge.

The theory presented here is oriented mainly on the work of Goetschi [2005] as well as the older reference Paper and also from the homepage Stanfort. This definitions and rules given here are not complete to the justification logic. Priority was given to those informations which are vital for the implementation. So however briefly and incomplete the theory is presented here full reference can be found in the named sources.

2.1 Background

Justification Logic has its origins from the field of modal logic. In model logic $\Box A$ means that A is *know* or that we have *proof* of A. In justification logic the equivalent would be $t : A$ where t is a *proof term* of A. This provides us the notion that *knowledge* or *proofs* may come from different sources. Justification logic lets us connect different *proofs* with a few simple operators and thus give us a better desciption of the proof. It may be said that where in model logic the knowledge is implicit it is explicit in Justification Logic[1].

2.2 Rules and Definitions

The language of justification logics is given here in a more traditional format with falsum and implication as primary propositional connectives. Although for the work done with this implementation only the implication has been used and the falsum has been ignored. [2] Also not all available syntactic objectes are introduces here but only those implemented.

Definition 1. *Apart from formulas, the language of justification logics have another type of syntactic objects called* justification terms*, or simply* terms *given*

[1] Goetschi [2005]

[2] cite here! S. 17

by the following grammar:

$$t ::= c_i^j | x_i | \bot | (t \cdot t) | (t + t) | !t$$

where i and j range over positive natural numbers, c_i^j denotes a (justification) constant *of level j, and x_i denotes a (justification)* variable*.*

The binary operations $\cdot$ and $+$ are called application *and* sum*. The unary operation ! is called* positive introspection*.*

Rules. *Application, sum and positive introspection respectively.*

C1 $t : (F \to G, s : F \vdash t \cdot s : G$

C2 $t : F \vdash (t + s) : F, \;\; s : F \vdash (t + s) : F$

C3 $t : F \vdash !t : t : F$

Formulas are constructed from propositional letters and boolean constants in the usual way with an additional clause: if F is a formula dn t a term, then $t : F$ is also a formula.

Definition 2. Justification formulas *are given by the grammar:*

$$A ::= P_i | (A \to A) | (t : A)$$

where P_i denotes a proposition, as in the modal language, and t is a justification term in the justification language.

This is almost all we need for the proof search of a (justification) formula. The last definition gives us something like a reference for the proof constants.

Definition 3. *A constant specification,* CS*, is a finite set of formulas of the form $c : A$ where c is a proof constant and A is a axiom of Justification language.*

The axioms mention in this definition are *C1-C3* in addition to $t : F \to F$ and the Axioms of the classical propositional logic in the language of LP.

Chapter 3

A Divide and Conquer Algorithm

3.1 Core Idea

To search a formula for is provability it had to be found a way which allows to do the same steps, now matter what form the formula actually has. A first attempt was to strictly use recursion. This method should have worked but it proved to be very difficult to implement, because there are so many different cases to consider in one recursion step. Also the stack created by this could become problematic for very large formulas.

Instead a *Divide and Conquer* approach is used. Diving will break even a large and complicated formula down to its most simple elements. Then these elements can be tested for their provability and in the conquer-step the results of the elements are put together giving the final result. Since the *Divide and Conquer* algorithm design pattern uses multi-branched recursion there still remains some recursion but as this takes place at a much deeper level the cases within a recursion are reduced as well as the size of the recursion stack.

3.2 Divide

The motivation behind the divide-step existed already long before the actual idea of the Divide and Conquer approach. Given a formula there would be no way to know what kind of formula it was or more precise: what operations were to be found within the *justification term.* The original goal was to find a way to restructure any given formula so that handling it would always need the same steps and not depend to much on what the formula looks exactly. I was looking for something like the CNF[1] and use it in a similar way as CNF is used in PSC[2]. As the Sum-Rule for *justification terms* works straight forward like a disjunction it is rather simple to restructure the formula as far as the Sum-Operator goes.

[1] *conjunctive normal form*, a conjunction of clauses, where a clause is a disjunction of literals

[2] *Proof Search Calculus* as it is introduced in Goossens et al. [1993]

add graph or formula

But since the Multiplication-Operator is not even symmetric operator it does not work like the conjunction know from the CNF and thus makes to restructuring of a formula all the more difficult. In addition there is the unary Bang-Operator which in itself is rather simple but still adds to the overall complexity.

In the end the restructuring would look like the following:

- For each Sum-Operator in the formula, split it in two formulas.
- If the first operation of a formula is a Bang-Operation, check if it can be simplified. If not, remove this formula.
- There are certain positions of a Bang-Operator within the formula that cause the whole formula to be false. Those formulas shall be eliminated as well.

Those three steps which are called *Atomize* in the source code break a given formula down to several simpler formulas which only contain the Multiplication-Operator as well as valid Bang-Operators. It is only then that a recursive method is called to analize the formula in a way that makes it possible to check if this subformula is provable.

This basically concludes the Divide-Part of this algorithm. The only thing left to done in the Conquer step is to check each of these formula. In the case of Justification Logic it means they need to be looked up in the *constant specification*.

3.2.1 Atomize

In this section *formula* usually refers only to the justification term of the formula and is used as a synonym. If it should be understood differently it will be stated so explicitly.

Definition 4 (atomized)**.** *A formula or term is called* ***atomized*** *if it fulfills the following conditions:*

- *The term contains no Sum-Operations.*
- *A Bang-Operation can neither be the top operation of a term nor be the left operant of a Multiplication-Operation.*

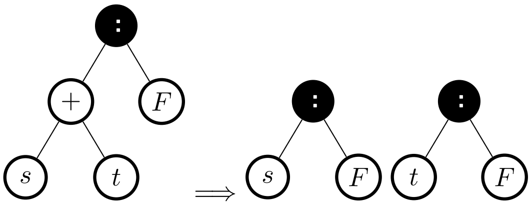
To make the content presented here more understandable the following example will illustrate the steps taken.[3]

Sumsplit

From the XX Rule of Justification Logic it follows that checking for provability in a formula where the top operation is a sum is equal to checking either operant of the sum and if any of it is provable so is the original formula.

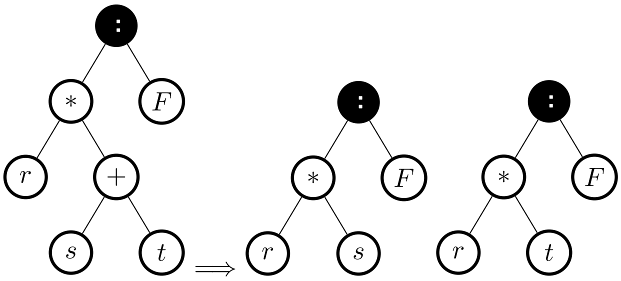
[3] It is on purpose that the *justification term* is by far more complicated than statement $b : F$ that follows the *justification term.* As far as this algorithm goes the complexity of the statement is of no further consequence and thus is kept as simple as possible to allow a easier overview.

$$\begin{aligned}(s+t):F&\\ &\Rightarrow s:F\vee t:F\end{aligned}\tag{3.1}$$



This is of course also true for formulas where Sum is not the top operation. Here x denotes a arbitrary *justification term*.

$$\begin{aligned}(r*(s+t)):F&\\ &\Rightarrow r:x\to F\wedge(s+t):x\\ &\Rightarrow(r:x\to F\wedge s:x)\vee(r:x\to F\wedge t:x)\end{aligned}\tag{3.2}$$

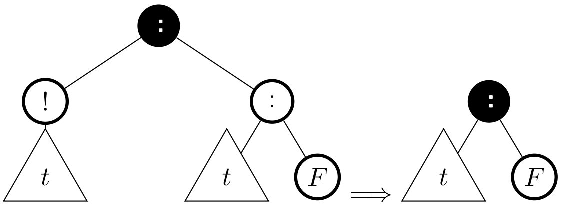


Simplify Bang

In this step the aim is to get rid of any Bang-Operator that is the first operation of a formula. Either the Bang can be removed and the formula simplified or else the formula is not provable at all and can be discarded.

Derived from the XX Rule we get the following:

$$\begin{aligned}!t:(t:F)&\\ &\Rightarrow t:F\end{aligned}\tag{3.3}$$



Speaking in the manner of a Syntax Tree it needs to be checked, if the child of the Bang-Operation is identical with the left child of the right child of the root. In that case the formula can be simplified to right child of the root only. Else there is no way to resolve the Bang-Operation.

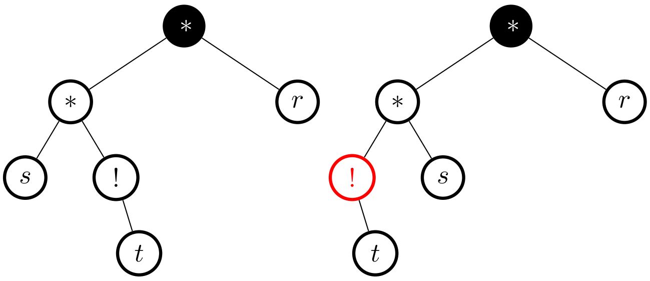
Remove Bad Bang

This last step in atomizing the formula proved to be on of the hardest to realize. Only countless examples support the claim that the Bang-Operation must not be the direct left child of a Multiplication-Operation. In coming to that conclusion it has been helpful that no Sum-Operation could make the situation more complex. Because of this and also the fact that a Bang-Operation is never the top operation in a formula it is guarantied that a Bang-Operation must be either a right child or a left child of a Multiplication-Operation.

Assertion 1 (Tree Version)**.** *A Bang-Operator that is the direct left child of a Multiplication-Operator causes the whole term to be invalid (unprovable), given that the term is without Sum-Operators and no Bang-Operator at the top.*

$$\begin{aligned} (!s * t) : F & \\ & \Rightarrow \exists x : \quad !s : x \to F \wedge t : x \\ & \Rightarrow \exists x, y : \quad !s : x \to F = !s : (s : y) \end{aligned} \tag{3.4}$$

The last line gives a contradiction since there is no possible x or y such that would fulfill the condition of $x \to F = s : y$.



This concludes the *atomization* of one formula to many simple formulas which can be checked for provability individually. A formula now consists only of Multiplication-Operations and valid Bang-Operations. The next chapter will show how one atomized formula can be checked for provability.

3.3 Conquer

The conquer-step consists mainly in the analyzing of the formula and only as a very last step the typical recursion is used when the result of all atoms of a formula evaluates to the overall result of the original formula. This is actually very simple compared to the analyzing part that comes before.

First we have to know what to look up in the cs-list for which proof constant. Those proof constants and the corresponding terms will be called *musts* since even now I lack a more suitable word for it and in the source code it is always referred as such.

Once that the *musts* have been obtained we can search the cs-list of terms that match. Since a *must* usually consists of variables that are not determined we will usually get more then one possible match per proof term. Also since cs-list allows terms that contain variables as well this will impose further conditions on the possible choice of the term of a proof term. In the second step all those possibilities and conditions are collected.

Then in the third and most important step those configurations and conditions will be merged. It will be checked if there is a possible combination from the given options such as the atomized formula is provable. It is then only a small step to collect the results of all other atoms of the original formula to determine the provability of the original formula.

3.3.1 Get Must

The operator rules which were presented in Chapter XX gives us the instruction how we can take a formula apart to look the individual proof term up in the cs-list. The rule for the Sum-Operation was already used in the divide-step for the sumsplit in 3.2.1.

It can be summarized that each Multiplication-Operation in a term adds one variable, which will here be called *wilds* or *x-wilds* where as a Bang-Operation will replace an existing *wild* with a new wild combined with the proof constant of that case.

The algorithm starts at the top of the syntax tree of the proof term of the formula and for each level it descends it recursively calls the same method on this child nodes.

So for a term like this $(a * (!b)) * c) : F$ the following will be evaluated:

$$\begin{aligned} &(a * (!b)) * b) : F \\ &\qquad\qquad \Rightarrow a * (!b) : X_1 \to F \\ &\qquad\qquad \Rightarrow b : X_1 \end{aligned} \tag{3.5}$$

$$\begin{aligned} &(a * (!b)) : X_1 \to F \\ &\qquad\qquad \Rightarrow a : X_2 \to (X_1 \to F) \\ &\qquad\qquad \Rightarrow !b : X_2 \end{aligned} \tag{3.6}$$

$$\begin{aligned} &!b : X_2 \\ &\qquad \Rightarrow X_2 = b : X_3 \end{aligned} \tag{3.7}$$

X_2 will be replaced by $b : X_3$ so our final *must*-list will look like this:

$$musts_{(a*(!b))*c):F} = \{a : [(b : X_3) \to (X_1 \to F)], b : [X_1, X_3]\} \tag{3.8}$$

As can be seen in this example a proof constant my have more than one term that needs to be looked up in the cs-list.

3.3.2 X-Wilds in the Configuration Table

Now the *must*-list from the previous step will be compared with the given cs-list. The terms of a proof constant in *must* will be matched against those the same proof constant in the cs-list. This will give us what I call a *configuration table*. The conditions which are only important in special situations will be postponed for a later example.

For convenience it will be asserted that the cs-list is given in the same structure as the must-list. The function used in the code to do this operates on a proof constant level, that means it iterates through the must-list and for each of the proof constants terms it compares it to the terms of the same proof constant of the cs-list. Therefore we will look only at one proof constant in this example as well.

Term from must-list: $a : (X_2 \to (X_1 \to F))$
Cs-list[4] : $\{a : [A \to (A \to F), (b : B) \to A, B, C \to (A \to F), (b : B) \to (B \to F)]\}$

Term	Configuration Table	
$(X_2 \to (X_1 \to F))$	X_1	X_2
$A \to (A \to F)$	A	A
$(b : B) \to A$	-	-
B	-	-
$C \to (A \to F)$	A	C
$(b : B) \to (B \to F)$	B	$b : B$

Table 3.1: Matching the term of a from the must-list with those of the cs-list. A configuration table contains only constants. If a X_i can not yet be evaluated the place is left empty.

Y-Wilds and Conditions

Later the the definition of the cs-list was changed such that it may also include more general formulas using variables. The purpose behind it was to have a way to include concepts such as tautologies. As a convention those special variables in the cs-list will be called *y-wilds* and be denoted as Y_i. An example of such a special entry in the cs-list would be the proof term $t : Y_1 \to (Y_2 \to Y_1)$ where Y_1 and Y_2 may represent any term[5]

As mentioned before this additional feature made the current process of comparing the *musts* with the corresponding entries in cs-list as well as the following *merge* step a lot more complicated. Suddenly it comparing is not only a match or no match, but the result may be relation/dependency between different variables. The problems lies in the fact that when comparing X_i with Y_j either could be anything. Before there were only different possibilities of what one certain X_i could be, but now the choice of a X_i depends on the choice of the Y_i and vice a verse. The table as it is used most only contain constants but the found information about the relations could not be ignored. To handle this situation I had to find a way of passing those *conditions* on.

Definition 5. *For one match between a configuration term from the cs-list and a* must-*term (one line in the configuration table) there may be zero or more* conditions *that apply to this certain match.*
A condition *is a tuple with two positions. The first position gives the X_i to which the condition applies and the second position holds the condition posed on this* X-wild. *The condition may contain any term, including* X-wilds *or* Y-wilds *but it is always* on *a* X-wild.

[4] contains only entries for the proof constant a

[5] For this implementation only terms that use only the operations for justification logic as introduced in Chapter XX may be used.

Example. *Conditions on X_i*

- (X_1, X_3)
  This is a X-to-X condition meaning that X_1 and X_3 must have the same value.

- $(X_1, X_2 \to F)$
  This condition allows us to set $X1$ as soon as we know $X2$.

- $(X_1, Y_1 \to X_3)$
  In this condition Y_1 may be any value, but if Y_1 is present in an other condition in this configuration, it must be replaced by the value set here for Y_1.

For this kind of condition applying it was necessarily to follow the following assertion:

Assertion 2. *Each X_i occurs at most once within one* must-*term.*[6]

Example. *$t : X_3 \to (X_3 \to F)$ is not possible.*

An other assumption I previously assumed has proven to be wrong only now when I tried to present it here. The assertion stated that a condition never contains a *X-wild* and a *Y-wild* in the same term as you find it in the last example for the conditions. This assertion would have simplified the merging of configurations with conditions quiet a bit. Unfortunately the assertion doesn't hold. The example that disproves it will be presented below. Solving this was easier that I feared but it was still a nasty bit of work.

Example for Y *and* X in condition

To summarize we have now per one *must* of a atomized formula a configuration table which includes the possible matches of the needed proof terms from the cs-list. The next step will try to merge this different configurations together to find one or more configuration for the $X - wilds$ that works for all proof constants.

3.3.3 Merge

The merging of the different configuration is the last important step in this algorithm. If a merge is possible this atomized formula is provable and if all atomized formulas are provable then so is the original given justification term.

Performing a merge without having any conditions is a straight forward implementation, where priority was given to a simple and easy understandable structure and not to a optimal performance time. Each line of one table is check with every other line of a second table, if the merge is possible, the result will be listed in a new table, if the merge is not possible the next line is used. The resulting table is then merged with the table of the next proof constant, so in the end the finale table may contain one or several lines or none at all.

As for the conditions which where introduced in the previous chapter, those will be evaluated only if there is a merge of the constants as shown in the table

[6] This basically follows from the way how the *X-wilds* are generated. Each Multiplication-Operator gives exactly one new *X-wild*. For more details see the Implementation Chapter.

	X_1	X_2	X_3	X_4
i	A		B	
	B		C	
	C		A	
ii			B	A
			C	A
			C	B
iii	A	A		A
	A	C		A
	B	B		B

Table 3.2: Configuration table. Each section of the table represents the configuration table for one proof constant.

	X_1	X_2	X_3	X_4
Merge of *i* and *ii*	A		B	A
	B		C	A
	B		C	B
Adding *iii* to merge	A	A	B	A
	B	B	C	B

Table 3.3: Merge table. The last section is the final configuration. Since there is a possible configuration the must term from which these configurations are is provable.

is possible. After the merge is successful each condition will be applied one after another. A condition can break the merge of two lines and thus make it not usable, it may also add further constants to the line or it might be, that the condition is not relevant yet and thus is simply passed on for the next line merge. Again this is best show in a example. It will be covered in the next Chapter.

Examle with a condition that sets some but not all *X-wilds*

This concludes not only the merge step but the whole divide and conquer chapter. I personally have found it rather easy to understand the individual steps but difficult to not get lost in the overall view. For that reason chapter 5 will cover one single example designed to show all aspects of the algorithm and run it through to understand it better.

Chapter 4

Implementation

4.1 Model Overview

For the implementation of this algorithm there was only one model that truly stood out in the sense of object orientated programming. For all other functions it proved rather difficult to find a clear class where it belonged to and also making good use of responsibilities of those classes. As mentioned in chapter 2 the main work was done using syntax trees and so it was the most obvious class to build. I decided to make a extra class for the *Nodes* where all the small things like setting a child, and checking if it is a root and such would be handled. *Tree* and *Node* could have been merge to be one class only but I found it easier to work with the code if the more standard and trivial stuff of binary trees was separated from what was more specific for this algorithm. A lot of the *atomize* part is handled by the *Tree* class since it works within the formula and also changes the structure of it.

The most important class however is the *ProofSearch* class. It acts as a sort of main class as the initialization of the formula and the cs-list takes place here. It is also here that the methods of the *Tree* class are called from. Its responsibility is to handle all algorithmic task that can be done without using a syntax tree. So the major logic of the conquer step is implemented here.

Last there is the typical *Helper* class. The methods here are usually rather short and simple and serve the purpose of making the *Tree* and *ProofSearch* class appear cleaner. It may be argued that some of those methods present in *Helper* should be better placed in *ProofSearch* and vice a verse but then again the argument for a clean object orientated model design for an algorithm is questionable and very difficult to archive.

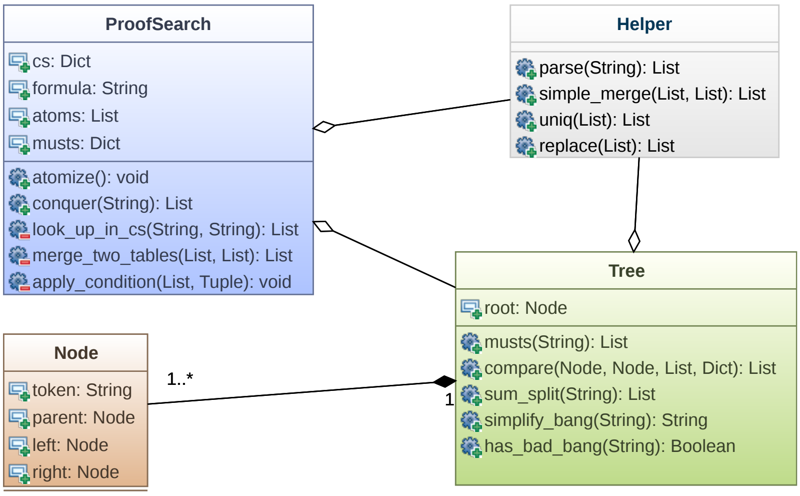


Figure 4.1: Simplified UML graphic of the classes used for implementing the algorithm. The list of methods and attributes is by no means complete and should simply give an idea of the construction.

4.2 Operation Syntax Tree

One of the earliest challenges was a useful representation of a formula with which I could work decently. Interestingly enough a binary tree came only later into my mind, after I tested various libraries from Python. There were libraries that seemed very useful at first as they were math-specific. Analyzing formulas that contained $*$ or $+$ were fairly easy but as : and ! are not very common operations I could not customize the tested libraries enough to handle those as well.

So it happened while I was searching yet for another library that I tumbled over the possibility to use binary trees to represent the syntax of a mathematical formula. Remembering a lot of what I learned in the lecture about Datastructure and Algorithms I realized that this is the best choice for me. A binary tree gives me not only a way to represent a formula in a way that interprets the order of operations but with the knowledge about trees it became suddenly very easy to also manipulate such a formula for example by deleting or swapping subtrees and still keep a valid operation.

I decided to implement my own tree for that purpose. It might be argued that a lot of work could be saved if I used available syntax trees but for one thing I relished the idea of implementing a tree structure that I would use myself and thus finally use what I have learned in lecture ages ago and second I would have to make custom changes to a finished solution anyway and those changes are probably more work than the implementation of a binary tree which is rather simple.

I tried to keep the tree as simple as possible, giving the nodes only a value and not a unique key. The greatest challenge given by implementing a syntax tree was to handle the unary operator !. As braces serve to determine the depth of a tree and a binary operation tells you when to start climbing up again, it required so extra case handling for the ! operator. From the point on when the tree was working, it was not only important to the algorithm, but could also be used to check if the input was written correctly. Therefore most of the tests that test the string handling of a tree are the result of formulas used somewhere else but which needed syntax spell checking.

4.3 Important Methods

In this section I want to show and explain some of the more complicated methods that are important and make up the heart of the algorithm.

4.3.1 Tree.musts

The method *musts* expects a given proof term to be already *atomized* as it only distinguishes between ! and $*$ operations. The algorithm takes the formula in form of a tree apart from top to bottom, generating new, smaller terms for every operation it takes apart until the remaining proof term is only a proof constant. Since the resolve of a $*$ operation needs a new *X-wild* and the resolve of a ! operation replaces an existing *x-wild* and therefore needs a new as well, the current i for a new *X-wild* X_i is stored and increased in `v_count`.

```
...
if len(proof_term.to_s()) == 1: # constant
    consts.append((proof_term.to_s(), subformula))
else:
    if proof_term.root.token == '*':
        left = proof_term.subtree(
            proof_term.root.left).to_s()
        right = proof_term.subtree(
            proof_term.root.right).to_s()
        temp.append(Tree('('+left+
            ':(X'+str(v_count)+'->'+subformula+'))'))
        temp.append(Tree('('+right+
            ':X'+str(v_count)+')'))
        v_count += 1
    elif proof_term.root.token == '!':
        left = proof_term.subtree(
            f.root.left.right).to_s()
        s = '('+left+':X'+str(v_count)+')'
        temp.append(Tree(s))
        swaps.append((subformula, s))
        v_count += 1
```

Figure 4.2: Excerpt from $Tree.musts$

If for example the current justification term would be $((a * (!b)) : F)$, it would be taken apart to the two subformulas $(a : (X_i \rightarrow F))$ and $((!b) : (X_i))$. Because of the *atomization* in the steps before it is guaranteed that every ! is a (right) child of a $*$ and since every $*$ creates a new X_i, a term here that starts with a $*$ is always on a X_i. Since from $!b : X_i$ follows $\exists X_j \quad s.t. \quad !b : (b : X_j)$, all X_i that occurred up to now must be replaced by $(b : X_j)$. In the end we will have only proof constants remaining.

4.3.2 Tree.compare

This method gives us foundation to the method *apply_condition* which we will look at later. It is also only very tedious and was probably written and rewritten again more than any other part of the code. It basically compares the value of each node of a tree with the corresponding value of a node of another tree. As long as the token (value) of the nodes are the same it will just process, but in case the two nodes are not identical several cases will be distinct.

```
...
elif cs_node.token[0] == ’X’:
    # condition containing Xs and constants.
    if ’X’ in orig_node.to_s():
        t = (cs_node.token, orig_node.to_s())
        conditions.append(t)
    # wild
    else:
        wilds[cs_node.token] = orig_node.to_s()
# unresolved ’Y’ in cs-subtree to corresponding ’X’ in orig
elif orig_node.token[0] == ’X’
        and (’Y’ in cs_node.to_s() or ’X’ in cs_node.to_s()):
    # condition containing Xs, Ys and/or constants.
    t = (orig_node.token, cs_node.to_s())
    conditions.append(t)

# is compared with a formula containing only constants.
elif orig_node.token[0] == ’X’ or orig_node.token[0] == ’Y’:
    wilds[orig_node.to_s()] = cs_node.to_s()
else:
    # no match possible
    conditions, wilds = None, None
```

Figure 4.3: Excerpt from *Tree.compare*

If the current node of the term from the cs-list has a *Y-wild* all other occurrences of this *Y-wild* within this term will be replaces with whatever node or subtree is found in the term of the *musts*. As a consequence it is possible to find a *X-wild* within the term from the cs-list. Whenever that is the case, the value found in the node/subtree of the term of the *musts* is a condition to this X_i. If the condition is merely a constant value, it will be set directly.

As can be seen here the return value of this method is a list of conditions and a dictionary containing the found wilds. If a contradiction is found during the comparison both will be set to `None` rather than being returned empty, since it is possible that a comparison need neither conditions nor wilds but still works.

Graphic with wild replacement for compare method

4.3.3 ProofSearch.apply_condition and ProofSearch.apply_all_conditions

These methods are called from within the method `full_merge_of_two_configs` if the wilds of the configurations can be merged. It is used to make sure that the new found configuration still holds for all conditions that applied before to each of the old configurations before they were merged.

This again proved more difficult than expected so I split it in two methods where the first simply handles one condition on one term and the other method makes sure that all conditions are taken care of. In the cases I studied so fare there is usually only one condition to a configuration but since it is possible that a configuration holds several conditions the method must hold in those cases as well.

Apply One Condition

The input arguments given to the method are the merged configuration and one condition tuple.

```
x_index = int(condition[0][1:])-1
config_term = config[x_index]
condition_term = update_condition_with_x(condition[1], config)
```

Figure 4.4: Excerpt from *ProofSearch.apply_condition*.

Since a condition is always on one X_i the term we find in the configuration for X_i is of high interest. But because the condition term may also contain other X_j we need the whole configuration and not only the term on which the condition is. If there is now term for X_i in the configuration yet the condition cannot be applied and must be kept for later reference.

In the case that we do find an entry for X_i in the configuration it will be compared with what we have in the condition term.

```
if config_term != '':
    conds, wilds = Tree.compare(Tree(condition_term).root,
                                Tree(config_term).root, [], {})
    if wilds is not None:
        assert conds == []
        y_wilds = {}
        for key in wilds:
            if key[0] == 'X':
                i = int(key[1:])-1
                if wilds[key] == config[i] or config[i] == '':
                    config[i] = wilds[key]
                else:
                    return None, None, None
            elif key[0] == 'Y':
                y_wilds[key] = wilds[key]
        return config, None, None if y_wilds == {} else y_wilds
    else:
        return None, None, None
```

Figure 4.5: Excerpt from *ProofSearch.apply_condition*.

Generally spoken there are three outcomes for a condition when it is checked against the merge of two configurations.

- The condition is in contradiction with the merge of the two configurations. The merge is abandon and not further considered.

- The condition holds against the new merge but a comparison gave use *X*- and/or *Y-wilds*.

  - Found *X-wilds* will be written into the configuration merge.
  - Found *Y-wilds* will be passed on to be handled later on.

- The condition holds against the new merge and no *wilds* emerged. The condition is discarded since it is not used anymore.

Apply All Conditions

Applying all conditions instead of just one should be straight forward but I found it more difficult than expected. Since the number of conditions may change during iteration and since it is also possible that a condition that held a *Y-wild* can change and has to checked again, a simple `loop` would not do.

```
todo_conditions = list(conditions)
updated_config = config
remaining_conditions = []

while todo_conditions:
    current_condition = todo_conditions.pop()
    updated_config, mod_conditions, y_wilds =
        apply_condition(config, current_condition)

    if updated_config:
        if mod_conditions:
            remaining_conditions.append(mod_conditions)
        if y_wilds:
            updated_conditions =
                get_all_with_y(todo_conditions, y_wilds.keys())
                 + get_all_with_y(remaining_conditions, y_wilds.keys())
            for key in y_wilds:
                updated_conditions =
                    update_y(updated_conditions, key, y_wilds[key])
            todo_conditions = todo_conditions + updated_conditions
    else:
        break
return updated_config, remaining_conditions
```

Figure 4.6: Excerpt from *ProofSearch.apply_condition*. The

If the condition can be satisfied the variable `updated_config` will be true for the `if`-query. If a condition is still returned it must be kept for later reference thus. If `Y-wilds` contains any entries all conditions that contain any of the wilds (even those that have been checked already) will be updated and checked again. This is for the case that a *Y-wild* has been found and although a *Y-wild* may stand for any term, it must be the same term for all Y_i.

The `break` in the code indicates the situation that one of the conditions could not be satisfied thus the whole merge is a fail.

4.3.4 ProofSearch.full_merge_of_two_configs and merge_two_tables

Similar as in the previous section the process that uses the methods above and actually merges all configurations together to get a solution is split in two parts where the first part handles a merge of only two configurations and the other handles the full merge.

Todo

4.4 Tests

Todo

Chapter 5

Example

5.1 Initialization

In this chapter I would like to walk through one example and covering as many special cases as possible. As such, the justification term we will look at is rather complicated. But this example will also show how nicely this will be broken down in more simpler formulas.

$$f = (((!(a + c)) + ((a + (!a)) * (b * (!c)))) : (c : F)) \tag{5.1}$$

The cs-list used for this example will only be relevant later on but still be presented here as reference:

$$\begin{aligned}
cs = \{ & \\
& a : [(H \to (c : F)), ((E \to (c : D)) \to (c : F)), (E \to (c : D))], \\
& b : [((c : F) \to H), ((c : D) \to (a : F)), ((H \to G) \to H), (Y_1 \to (Y_2 \to Y_1))], \\
& c : [(c : F), G, D, (G \to F)] \\
& \}
\end{aligned} \tag{5.2}$$

The data presented here is in the same form as it would be entered into the program. Therefore the cs-list is rather a *python* dictionary than a simple tuple-list and there are more brackets explicitly written then required by convention.

5.2 Walking in Trees: Atomize

The given formula f will be transformed into a syntax tree using *parse_formula* of *Tree*. If the formula is provided when the *ProofSearch* object is initialized the formula will automatically be atomized without having to call this method separately.

$$(((!(a + c)) + ((a + (!a)) * (b * (!c)))) : (c : F))$$

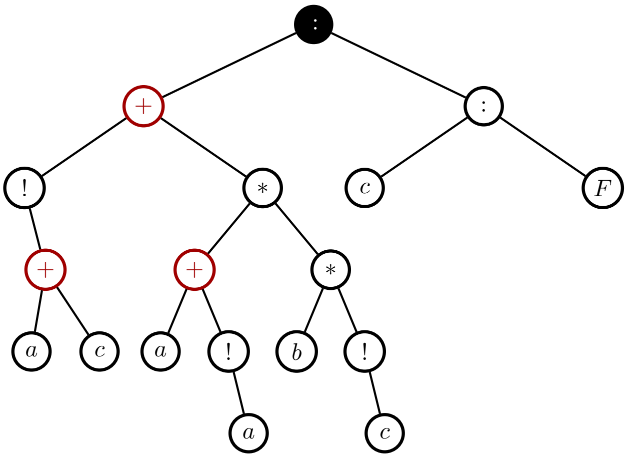


Figure 5.1: Syntax tree of given formula f before it is atomized.

The *sum_split* from *Tree* will give us the following terms in form of a list.

$$((!a) : (c : F)) \tag{5.3}$$

$$((!c) : (c : F)) \tag{5.4}$$

$$((a * (b * (!c))) : (c : F)) \tag{5.5}$$

$$(((!a) * (b * (!c))) : (c : F)) \tag{5.6}$$

5.2.1 Bangs

Looking at each of the terms individually we will now further look at them to discard any that have a *bad bang*, meaning a bang that is the left child of a multiplication or if there are terms with bang which can be simplified.

Term 5.3

In this term we find a bang which is valid, since it is not a left child of a multiplication, but trying to simplify the term shows us that it cannot be resolved thus letting us discard is term.

$$((\,!\,a) : (c : F))$$

Term 5.4

As before the bang within the term is valid but in contrast to the previous example the term here can be simplified, giving us our first *atom* for formula f.

$$\begin{aligned} ((\,!\,c) : (c : F)) &\Rightarrow \\ a_1 &:= (c : F) \end{aligned} \tag{5.7}$$

Term 5.5

In this term we find the bang operation neither a left child of a multiplication nor as top operation of the proof term and thus there is nothing to do.

$$\begin{aligned}((a * (b * (\,!\, c))) : (c : F)) \Rightarrow& \\ a_2 := ((a * (b * (!c))) : (c : F))&\end{aligned} \tag{5.8}$$

Term 5.6

Finally this term has two bangs of which the first is the left child of a multiplication and thus makes the term invalid. The second bang would be valid, but the first term causes the whole term to be discarded.

$$(((\,!\, a) * (b * (\,!\, c))) : (c : F))$$

This completes the *atomize* step for the formula f giving us two *atoms*. Showing that at least one of those is provable is enough to show that f is provable.

5.3 Looking up and merging

$$f = (((!(a + c)) + ((a + (!a)) * (b * (!c)))) : (c : F)) \tag{5.1}$$

For our formula f we have found the two atoms 5.7 and 5.8. The next steps will be determining the *musts* if needed, matching them against the cs-list and finally merge the possible configurations together to determine if one of the musts is provable.

$$a_1 = (c : F) \tag{5.7}$$

$$a_2 = ((a * (b * (!c))) : (c : F)) \tag{5.8}$$

5.3.1 Musts

Atom 5.7

Since a_1 consists already only of one proof constant with the correspond term to it there is nothing further to to here.

$$a_1 : \quad musts = [(c, F)] \tag{5.9}$$

Atom 5.8

For a_2 we need to take the proof term apart bit by bit. The first operation we will take apart is a multiplication. Extracting proof constants from a multiplication proof term will always us give a *X-wild*. Whenever a new *X-wilds* appears the i of X_i will simply be increased by 1.

$$
\begin{aligned}
&((a * (b * (!c))) : (c : F)) \Rightarrow \\
&\qquad a : (X_1 \rightarrow (c : F)) \\
&\qquad (b * (!c)) : X_1
\end{aligned}
$$

The proof constant a has been isolated but $(b*(!c))$ still needs further taking apart. We repeat the step from above and introduce yet another *X-wild*.

$$
\begin{aligned}
&(b * (!c)) : X_1 \Rightarrow \\
&\qquad b : (X_2 \rightarrow X_1) \\
&\qquad (!c) : X_2
\end{aligned}
$$

Now b has been isolated as well, leaving only $(!c)$. Having a bang in a situation like this results in a new *X-wild* in combination with the proof term which will replace a previous *X-wild*.

$$
\begin{aligned}
&(!c) : X_2 \Rightarrow \\
&\qquad X_2 = (c : X_3)
\end{aligned}
$$

This finally gives us all the *musts* for a_2. As can be seen below the *X-wild* X_2 has been replaced by $(c : X_3)$.

$$a_2 : \quad musts = [(a, (X_1 \rightarrow (c : F))), (b, ((c : X_3) \rightarrow X_1)), (c, X_3)] \tag{5.10}$$

It should be noted here that a proof constant may be in more than one of the *musts* for one *atom*.

5.3.2 Configurations and Conditions

Now that we have all proof constants for each *atom* isolated we can try to match them with the entries found in the cs-list.

$$
\begin{aligned}
cs = \{ & \\
& a : [(H \rightarrow (c : F)), ((E \rightarrow (c : D)) \rightarrow (c : F)), (E \rightarrow (c : D))], \\
& b : [((c : F) \rightarrow H), ((c : D) \rightarrow (a : F)), ((H \rightarrow G) \rightarrow H), (Y_1 \rightarrow (Y_2 \rightarrow Y_1))], \\
& c : [(c : F), G, D, (G \rightarrow F)] \\
\} &
\end{aligned}
\tag{5.2}
$$

Configs for musts of atom a_1

The *atom* a_1 actually has only one rather simple *must*: (c, F). As it can easily be seen there is no such match with the cs-list for the proof constant c.

If we in fact would have found a valid match we would have also shown that the formula f is provable. Also the algorithm would usually stop as soon as one must is provable unless explicitly told so.

Configs for musts of atom a_1

$$a_2: \quad musts = [(a, (X_1 \rightarrow (c : F))), (b, ((c : X_3) \rightarrow X_1)), (c, X_3)] \tag{5.10}$$

Matching the *musts* with the entries in cs gives us the following table. Since the *X-wild* X_2 in never used it will be omitted here. In the implementation it would just be an additional empty column.

	X_1	X_3	condition
a	H	-	
	$(E \rightarrow (c : D))$	-	
b	H	F	
	$(a : F)$	D	
	H	$(H \rightarrow G)$	
	-	-	$(X_1, (Y_2 \rightarrow (c : X_3)))$
c	-	$(c : F)$	
	-	G	
	-	D	
	-	$(G \rightarrow F)$	

Table 5.1: Config table for a_2

All match except the match with the *Y-wilds* are straight forward and need no further explanations, and the *Y-wilds* match shall be explained in more detail here.

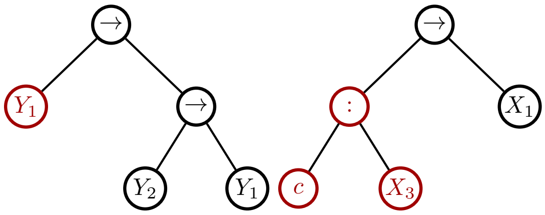


Figure 5.2: Comparing a term from cs-list with Y-wilds with a must term.

When the *compare* method from *Tree* encounters a node containing a *Y-wild* it will replace it and all other occurrences with the same *Y-wild* in its tree with whatever it finds in the node of the comparing tree. In this case it will replace all Y_1 with $(c : X_3)$.

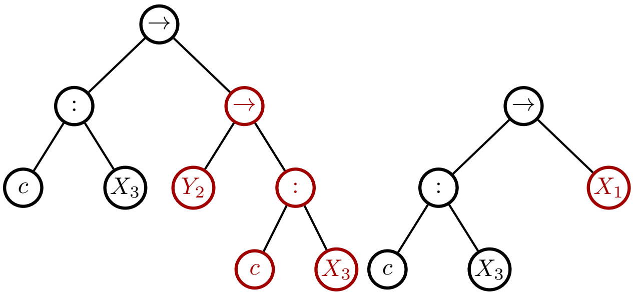


Figure 5.3: Updated cs-term (copy) with resulting condition.

When the methods now compares the right child of the root if finds a *X-wild* in the *must* term, it now checks if there are *X-wilds* or *Y-wilds* present in the subtree of the Node from the cs term. If that is the case, as it is here, the subtree be be a condition to the X_i. If the term would have consisted of operators and constants only the subtree would have been written directly into the configuration table. But as that is not the case here, we end up with a empty row in the config table and the condition $(X_1, (Y_2 \rightarrow (c : X_3)))$.

5.3.3 Merging of the Config Table

As we have only one atom left that might still be provable we have to merge the different sections from the config table 5.1 to find at least one configuration for all *X-wilds* that is contradiction free for all proof constants. The *merge*-step in the algorithm merges two sections and adds to the result the next table, so in this example we will first merge a and b and to the result of that merge c.

	X_1	X_3	condition
a	H	-	
	$(E \rightarrow (c : D))$	-	
b	H	F	
	$(a : F)$	D	
	H	$(H \rightarrow G)$	
	-	-	$(X_1, (Y_2 \rightarrow (c : X_3)))$
$a \cap b$	H	F	
	H	$(H \rightarrow G)$	
	$(E \rightarrow (c : D))$	D	

Table 5.2: Merge of section a and b

The first two merges are obvious, there are however some comments on the last one. As $E \rightarrow (c : D)$ for fits perfect the condition $Y_2 \rightarrow (c : X_3)$ for X_1 giving us $X_3 := D$, the second line from a can be merged with the third from b. In this case the condition can be dropped because it has fully been satisfied, but in other cases, especially if there are more *X-wilds* it is possible that a condition still persists after a merge. Also we found that Y_2 must be E which would matter, if there were other conditions on the same configuration line that also contained a Y_2 (rather unlikely but very possible).

Now we will merge the section from c to our current result of table 5.2.

	X_1	X_3	condition
$a \cap b$	H	F	
	H	$(H \to G)$	
	$(E \to (c : D))$	D	
c	-	$(c : F)$	
	-	G	
	-	D	
	-	$(G \to F)$	
$(a \cap b) \cap c$	$(E \to (c : D))$	D	

Table 5.3: Merge of section c with the previously merged a and b.

5.4 The Final Result

As we have seen in table 5.3 we did find a configuration that fits the *X-wilds* and thus the original formula f is provable. I want to show here what this configuration actually means and what it has to do with the provability of f. To do this we will actually go through the algorithm backwards but having already the solution at hand.

What we gained from the last step is a configuration for the *X-wilds*.

$$\begin{aligned} &X_1 = (E \to (c : D)) \\ &X_2 = (c : X_3) = (c : D) \\ &X_3 = D \end{aligned} \tag{5.11}$$

Knowing these we can use them and replace them in the musts of the atom a_2. That way we get what we where looking for in the cs-list.

$$\begin{aligned} musts\, a_2 : \quad & [(a, ((E \to (c : D)) \to (c : F))), \\ & (b, ((c : D) \to (E \to (c : D)))), \\ & (c, D)] \end{aligned} \tag{5.10}$$

$$\begin{aligned} cs = \{ & a : [(H \to (c : F)), ((E \to (c : D)) \to (c : F)), (E \to (c : D))], \\ & b : [((c : F) \to H), ((c : D) \to (a : F)), ((H \to G) \to H), (Y_1 \to (Y_2 \to Y_1))], \\ & c : [(c : F), G, D, (G \to F)]\} \end{aligned} \tag{5.2}$$

As expected we find all the terms from *must* precisely like that in the cs-list. Also we can reconstruct a_2 with the *musts* given above:

$$\begin{aligned} c : D &\Rightarrow (!c) : (c : D) \\ b : ((c : D) \to (E \to (c : D)))), (!c) : (c : D) &\Rightarrow ((b * (!c)) : (E \to (c : D))) \\ a : ((E \to (c : D)) \to (c : F))), (b * (!c)) : (E \to (c : D)) &\Rightarrow (a * (b * (!c)) : (c : F)) \end{aligned}$$

$$a_2 = ((a * (b * (!c))) : (c : F)) \tag{5.8}$$

Reconstructing the whole formula f instead of just the atom follows the same approach as shown above but it will become very verbose and it is very little gained if one is not in doubt about the *atomization*.

$$f = (((!(a + c)) + ((a + (!a)) * (b * (!c)))) : (c : F)) \tag{5.1}$$

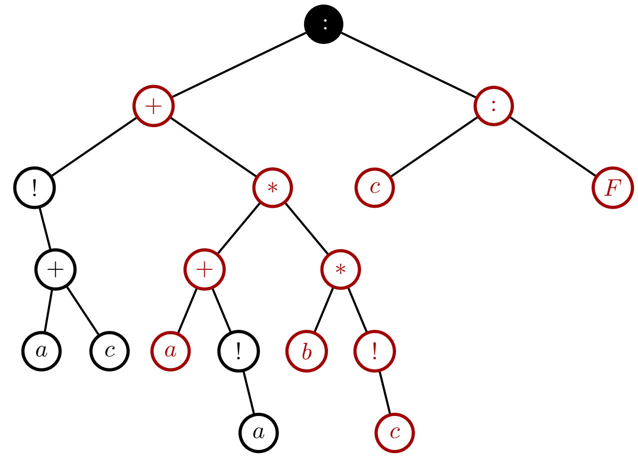


Figure 5.4: Syntax tree of formula f with atom a_2 hightlighted.

This concludes this chapter where I tried to show as much as possible with an example that is as short and simple as possible and still fits the purpose.

Chapter 6

Results

Todo

6.1 Application

6.2 Enhancement

Bibliography

Remo Goetschi. On the realization and classification of justification logics. 2005.

Michel Goossens, Frank Mittelbach, and Alexander Samarin. *The LaTeX Companion.* Addison-Wesley, Reading, Massachusetts, 1993.

Microsoft Research. Z3, high performance theorem prover. URL `http://z3.codeplex.com/`.

Artis Paper.

Plato Stanfort.

Todo list

todo i

Todo: What's the motivation behind it? Not **MY** motivation, but the scientific motivation. 1

Todo: Chapter about implementation 2

Todo: Chapter about Examle 2

add graph or formula 5

Example for Y *and* X in condition 11

Examle with a condition that sets some but not all *X-wilds* 12

Graphic with wild replacement for compare method 16

Todo 19

Todo 19

Todo 28