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

In short what's it all about.

Contents

Chapter 1

Introduction

1.1 Motivation

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:

Should this really be here in this chaper?

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.

j-logic

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

Background

The theory of Justification Logic as it is used here requires little knowledge of the wide fields of Modal Logic. For the purpose of this proof search a few basic rules and definitions are sufficient to provide the reader with the needed knowledge.

From a more practical angle a basic data structure was also needed to represent formulas. Already widely known and used *Binary Syntax Trees* revealed to be just the perfect way to handle formulas.

2.1 Justification Logic

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.

2.1.1 Origins

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. So we have 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 describe better the proof. It may be said that where in model logic the knowledge is implicit it is explicit in Justification Logic[1].

2.1.2 Rules and Definitions

Definition. [2]

[1] Goetschi [2005]
[2] Goetschi [2005] Page 17, incomplete

Is is ok to just leaf '?' and such out? How does it have to be referenced?

Justification terms *or just* ***terms*** *are syntactic objects given by the grammar*

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

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.

2.2 Order of Operation Tree

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 (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.*

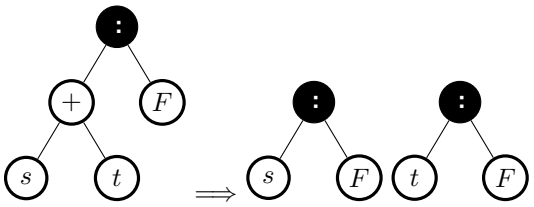
To make the content presented here more understandable the following example will illustrate the steps taken.[3]

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.

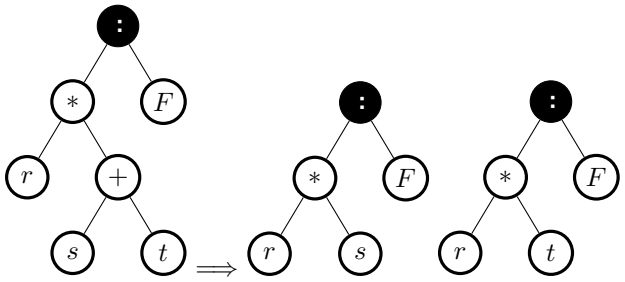
[3]It is on purpose that the *justification term* is by far more complicated than statement $b : F$ that follows the *justification term*. As far as this algorithm goes the complexity of the statement is of no further consequence and thus is kept as simple as possible to allow a easier overview.

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



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

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

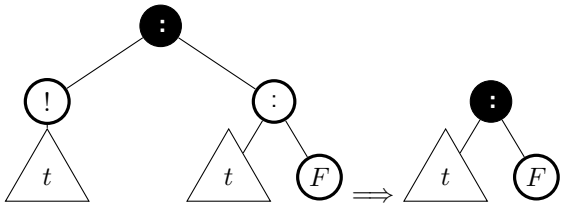


Simplify Bang

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.

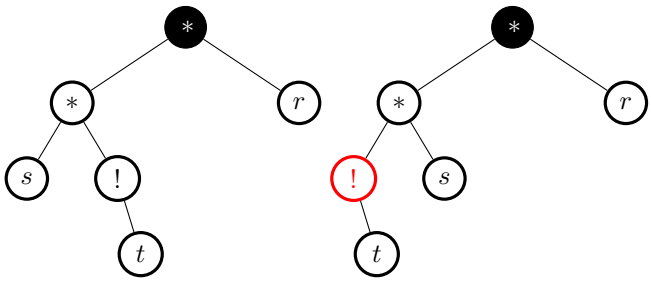
Operation? Operator?

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.*

Top-Operation? is it clear that I don't mean ':'?

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

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.

Find (and write!!) Chapter

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:

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

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

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

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 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]

Would be nice to have a X-Wild title somewhere as well

As mentioned beforethis 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.

where?

Definition. *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 included in the example later . Solving this was easier that I feared but it was still a nasty bit of work.

Link and Write Chapter

Table/Example?

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 section.

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 the next chapter 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 2.2 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.

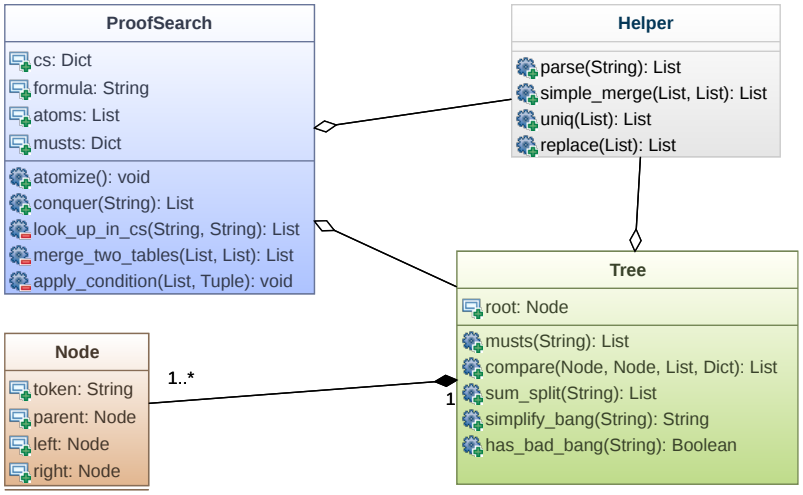


Figure 4.1: Simplified UML graphic of the classes used for implementing the algorithm. The list of methods and attributes is by no means complete and should simply give an idea of the construction.

4.2 Important Methods

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

4.2.1 Tree.musts

Actually the main achievement of this method is the atomizing step, as only this simplifications made this method possible. The algorithm takes the formula apart from top to bottom. The main cases which are differenciated are if the current operation is either a *-Operation, a !-Operation or if the term is simply a constant.

```
...
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: Tree#musts Taking a formula apart which has a * as top operation.

If for example the current justification term would be $((a * (!b)) : F)$, it would be taken apart to the two subformulas $(a : (X_i \to F))$ and $((!b) : (X_i))$. Because of the *atomization* in the steps before it is guaranteed that every Bang is a (right) child of a Multiplication and since every Multiplication creates a new X_i, a term here that starts with a Bang 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.

4.2.2 Tree.compare

This method is not only very tedious but gives to foundation to the method *apply_condition* which we will look at later. It basically compares each node of a tree with the corresponding 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: Tree#compare Here is some very verys useful text

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.

4.2.3 ProofSearch.apply_condition and ProofSearch.apply_all_conditions

4.2.4 ProofSearch.full_merge_of_two_configs

4.3 Tests

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.

$$f = (((((a*b)*((a*c)+(!c)))+(!(a+b)))+((a+(!b))*(b*a))) : (a : F)) \quad (5.1)$$

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

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

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.

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

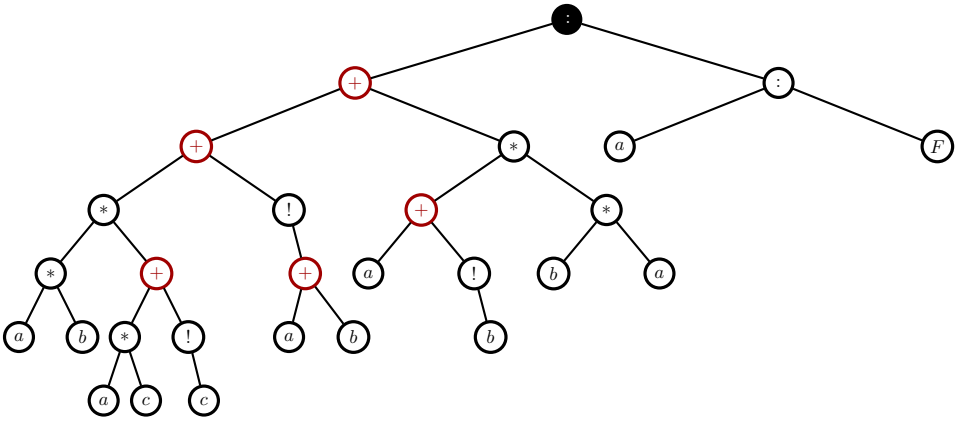


Figure 5.1: Syntax tree of given formula f before it is atomized.

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

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

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

$$((\,!\,a) : (a : F)) \tag{5.5}$$

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

$$(((a * (b * a)) : (a : F)) \tag{5.7}$$

$$((\,!\,b) * (b * a)) : (a : F))) \tag{5.8}$$

Further the Bang-Operation is important is the following cases, either because it can be simplified or removed.

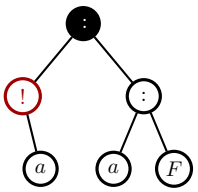


Figure 5.2: sample picture

The term of 5.5 is an example of a Bang operation that can be directly resolved and therefore simplified. Using the rules given in 2.2 we see that instead of $((!a) : (a : F))$ we may simply write $(a : F)$.

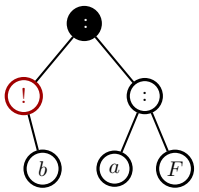


Figure 5.3: sample picture

The term of 5.6 is very similar to the one of 5.5 but where the second my be resolved the first is unresolvable, thus it can never be provable and we will discard it from here on.

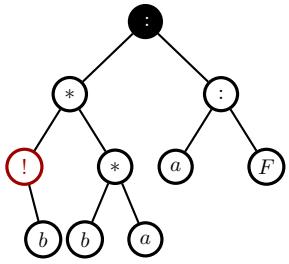


Figure 5.4: sample picture

The last term highlighted 5.8 is another example of a term that is discarded but for other reasons that the one in 5.6. In this term we find the Bang operator as a left child of a multiplication operator and as explained in 3 is also not provable.

With these steps of simplifying and removing bad bangs the *atomization* of the original formula f is concluded. The other formulas 5.3, 5.4 and 5.7 remain as they are. Although 5.7 has a bang, it is perfectly valid since it is the right child of a multiplication.

5.3 Extracting the Proof Constants: musts

Although a lot of work has been done up to now, we are not really further in the process of looking up proof constants in the cs-list to find out if the given formula f is provable or not. What we've done up to now is simplified the formula into many small ones so that the step we are on to now is easier.

$$\begin{aligned} f_{atoms} = \quad & [(((a * b) * (a * c)) : (a : F)), (((a * b) * (!c)) : (a : F)), \\ & (((a * (b * a)) : (a : F)), (a : F)] \end{aligned} \tag{5.9}$$

To extract the proof constants we will look at each *atom* individually. The last *atom* $(a : F)$ will be ignored, as it consists already of once proof constant only.

5.3.1 a_1

For this whole term we only need the multiplication rule. Simply put: The first operant will produce a new *X-wild* that implies to the subformula and the second operant will simple be the proof for the new *X-wild*.

$$\begin{aligned}(((a*b)*(a*c)):(a:F)) \Rightarrow \\ (a*b):X_1 \rightarrow (a:F) \\ (a*c):X_1\end{aligned} \tag{5.10}$$

Now the same wille be applied to the mid-results of the previous step.

$$\begin{aligned}(a*b):X_1 \rightarrow (a:F) \Rightarrow \\ a:X_2 \rightarrow (X_1 \rightarrow (a:F)) \\ b:X_2\end{aligned} \tag{5.11}$$

$$\begin{aligned}(a*c):X_1 \Rightarrow \\ a:X_3 \rightarrow X_1 \\ c:X_3\end{aligned} \tag{5.12}$$

Thus we get the *musts* for the first atom. Each *must* of this atom will be saved in a tuple where the first position is the proof constant and the second the term which will be matched with the terms found in the cs-list.

$$a_{1musts} = [(a, X_2 \rightarrow (X_1 \rightarrow (a:F))), (b, X_2), (a, X_3 \rightarrow X_1), (c, X_3)] \tag{5.13}$$

5.3.2 a_2

The second atom contains a Bang operator which causes a different step in addition to what we have seen in the previous atom. But the first step is practically the same as we have seen above.

$$\begin{aligned}(((a*b)*(!c)):(a:F)) \Rightarrow \\ (a*b):X_1 \rightarrow (a:F) \\ (!c):X_1\end{aligned} \tag{5.14}$$

$$\begin{aligned}(a*b):X_1 \rightarrow (a:F) \Rightarrow \\ a:X_2 \rightarrow (X_1 \rightarrow (a:F)) \\ b:X_2\end{aligned} \tag{5.15}$$

For the term with the bang, the X_i that belong to the bang-term is replace with a new X_j such that the term can be valid.

$$\begin{aligned}(!c):X_1 \Rightarrow \\ X_1 = (c:X_3)\end{aligned} \tag{5.16}$$

A list collects all X_i which must be replaced and after the all terms of this atoms are resolved, the X_i will be replaced by the new term. So we get for the second atom the following *musts*

$$a_{2musts} = [(a, (X_2 \rightarrow ((c:X_3) \rightarrow (a:F))), (b, X_2), (c, X_3)] \tag{5.17}$$

5.3.3 a_3

The third atom is very similar to the first so the equations will be put here without further comment.

$$\begin{aligned}((a*(b*a)):(a:F))\Rightarrow\\ a:X_1\to(a:F)\\ (b*a):X_1\end{aligned}\tag{5.18}$$

$$\begin{aligned}(b*a):X_1\Rightarrow\\ b:X_2\to X_1\\ a:X_2\end{aligned}\tag{5.19}$$

$$a_{3musts}=[(a,(X_1\to(a:F))),(a,X_2),(b,X_2\to X_1)]\tag{5.20}$$

As we have seen in all three examples above it is possible to have the same proof constant with different term for one *atomized* formula.

The algorithm calculates the *musts* for all atoms, to the result which is also an attribute of the *ProofSearch* class is a dictionary where the *musts* can be accessed as a list by the *atomized* term as key.

$$\begin{aligned}f_{atoms\quad with\quad musts}=\{\\ (((a*b)*(a*c)):(a:F)):&[(a,X_2\to(X_1\to(a:F))),(b,X_2),(a,X_3\to X_1),(c,X_3)],\\ (((a*b)*(!c)):(a:F)):&[(a,(X_2\to((c:X_3)\to(a:F)))),(b,X_2),(c,X_3)],\\ ((a*(b*a)):(a:F)):&[(a,(X_1\to(a:F))),(a,X_2),(b,X_2\to X_1)],\\ (a:F):&[(a,F)]\\ &\}\end{aligned}\tag{5.21}$$

5.4 Looking up and matching: Config and Conditions

Chapter 6

Results

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