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

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.*

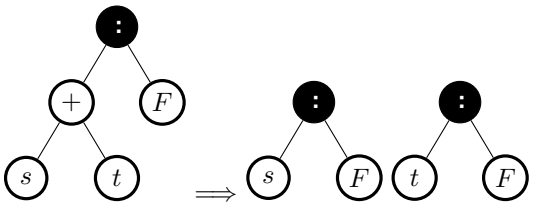
To make the content presented here more understandable the following example will illustrate the steps taken.[3]

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.

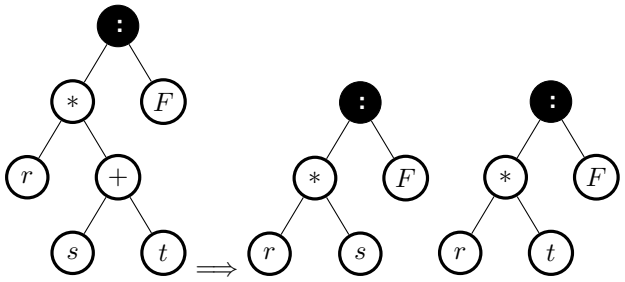
[3]It is on purpose that the *justification term* is by far more complicated than statement $b : F$ that follows the *justification term*. As far as this algorithm goes the complexity of the statement is of no further consequence and thus is kept as simple as possible to allow a easier overview.

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



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

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

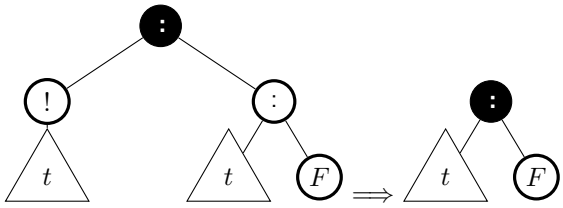


Simplify Bang

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.

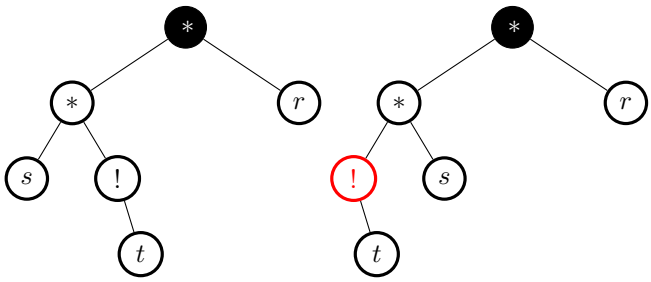
Remove Bad Bang

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

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 for terms that match it. Since a *must* usually consists of variables (*X-wilds*)that are not determined it is possible that we get more then one match per proof term. Also since the cs-list allows terms that contain variables (*Y-wilds*)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:

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

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

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

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 Matching and Conditions

Until very recently this step had a very different approach, which unfortunately proved to hold more than one mistake. Some could be fixed but one remained that could not be fixed with the original approach so after I though I would be done with coding I had to reimplement this whole part again. But in exchange it is no operating as it is supposed to.

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.

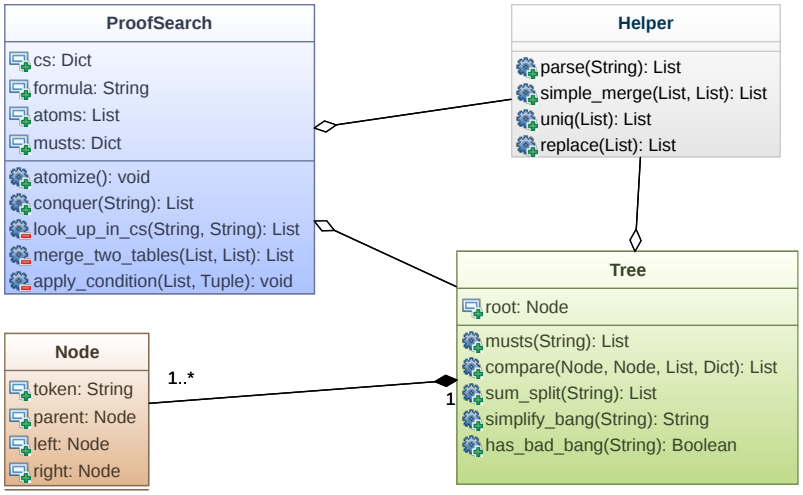


Figure 4.1: Simplified UML graphic of the classes used for implementing the algorithm. The list of methods and attributes is by no means complete and should simply give an idea of the construction.

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 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 $(a : (X_i \to F))$ 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 unify

I spend probably most of my implementing time on this method, or rather on its predecessor. It used to be a lot longer and more complicated because it differentiated various cases if a formula would contain one or another kind of variable. With this new and (hopefully) last implementation there is no different handling for *X-wild* or *Y-wild* variables on that level. Only much later when all results are put together will those variables be handled different accordingly.

The method is actually quiet forward: It takes two formulas[1] and compares them on the basis of their tree structure. While the structure remains the same thus operations and constants match for both formulas the methods pushes the subtrees of both on the stack until either one of the subtrees consists only of a node containing a variable or a mismatch is found. In the later case `None` will be return the method is stopped. In the case that we find a node with a variable for one tree it will be formed into a *condition* for that variable, where the variable is the key and whatever we find in the other tree at this place is the value.

```
stack = [(Tree(f1), Tree(f2))]
result = []
while len(stack) > 0:
    current = stack.pop()
    # If the root node is the same (either operation or constant)
    if current[0].root.token == current[1].root.token:
        if current[0].root.token in ['->', ':']:
            stack.append((current[0].subtree(current[0].root.left),
                current[1].subtree(current[1].root.left)))
            stack.append((current[0].subtree(current[0].root.right),
                current[1].subtree(current[1].root.right)))
        else:
            pass
    # If the root is not the same, either it is a mismatch, or there are variables.
    else:
        if (_has_no_wilds(current[0].to_s()) and _has_wilds(current[1].root.token))
            (_has_no_wilds(current[1].to_s()) and _has_wilds(current[0].root.token)):
            result.append((current[0].to_s(), current[1].to_s()))
        elif _has_wilds(current[0].to_s()) and _has_wilds(current[1].to_s()):
            assert _has_wilds(current[0].root.token) or _has_wilds(current[1].root.
            result.append((current[0].to_s(), current[1].to_s()))
        else:
            return None
return condition_list_to_dict(result)
```

Figure 4.3: Excerpt *unify* from *FormulaTools*.

All those conditions are stored as tuples in a list and are returned in form of a dictionary, where all conditions for one variable can be accessed by the variable itself as key. At the current state conditions that apply to a variable may be contradictory, but it the responsibility of this method is only to collect those and not to valuate them. This is done by the method `simplify` and in a further extension also in the method `resolve_conditions`.

[1] It is expected that the only occurring operations are → and :. It should be rather easy to extend the code at this point to accept also other operations but from what I can expect as input this is not necessary here.

4.3.3 simplify

This methods is used to get a result once all conditions are obtained by `unify`. The method is implemented in a way that it does only *simplify* the conditions for *one* Variable. To *simplify* a whole set of variables with their conditions a very simple method called `resolve_conditions` in `FormuaTools` is used.

The aim of this method is that after it run there is only one condition term left for this variable and the variable itself does not occur anywhere else except as key to its condition. The reason behind this it that eventually we will have for every variable only one condition left. We would know for each variable what its value would be. As it is so often this idea proved harder to realize than first anticipated.

Where as I usually present here an excerpt from the original source code in the case of this method I rather present the idea in pseudo code as the source code contains a lot of indexing and parsing of data types that make it cumbersome to read and understand it.

X shall be our variable, and $F_1, F_2, ...F_n$ are all conditions that are mapped for this variable.

- Preprocess all F_i: If any F_i contains X as part of its term we have a contradiction and the method will return `None` to indicate the contradiction. An exception is however if $F_i == X$ since this is not a very useful but a valid statement.
- For each pair F_i, F_j we will use `unify` to check if those conditions are compatible. As we have seen above we will get a set of conditions again from `unify`. From the preprocess we are guarantied, that all those new conditions do not contain the variable X. If there is one pair of conditions that is not compatible the method will return `None` again to indicate the found contradiction.
- Choose one F_k to be the only of the F_n conditions to be put back in the set of original conditions. Since F_k was matched with every other F_i in the preceding step this one condition on X together with the newly found conditions ensure that we don't loose any information and at the same time have only one condition on X left.
- Replace all occurrences of X in conditions of other variables with the single condition set to X by the step above.

The steps above are those originally intended, while implementing it I found that there's a catch. The algorithm work fine the way it is described above for one variable but when iterating through all variables there was a problem because the conditions gained by *unifying* the conditions of one variable might lead to new variable which where not present before because they only appear as part of a condition but not by themselves. The simplest solution seamed to be so initialize all variable and just leave the conditions empty, but because the order in which the variable are *simplified* is not reliable it would be possible that a variable with no conditions is *simplified* and only afterwards new conditions would be added. So instead I worked recursively[2] so that those new variables

[2]To be precise I work with a stack and push any new variable on top so that it is next to be *simplified*.

would be *simplified* and thus replaced everywhere except a variable key and with one condition before the algorithm would randomly take the next variable to *simplify*.

4.3.4 conquer

Write about resolve conditions

4.4 Tests

The tests I have written for this algorithm have been most important to the success of it. They served me in two ways: To check if my code would behave and actually do what I expected it to do and the other use was when I suddenly stumbled across a example or a situation where I did not know what I would expect I could see what my implementation would do with it, thus helping me to understand it better. Of course blessing of the tests is also a curse as it is because of the tests that I found so many mistakes that I've made and forcing me to do it again and again.

As can be seen when looking at the source code not all all methods are tested on a same quality level. Methods that I deemed simple usually have only one or two tests. An example for that is `nice` from `ProofSearch`, this methods simply rearranges the elements of a dictionary and returns a nice readable output that summarized the content of the original input. In contrast to this methods are methods like the `conquer` methods and the `divide` method from `ProofSearch` or the `to_s` from `Tree` which basically tests if the parsing between String and Tree works correctly.

To name some numbers, there are currently[3] a total of 66 tests. Almost halve of which are found in `ProofSearch`.

As for the technology I used a simple unittesting framework.

[3]Even though the program is finished it is still possible that I add more tests to get rid of any doubts, so the numbers are not fix.

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.

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

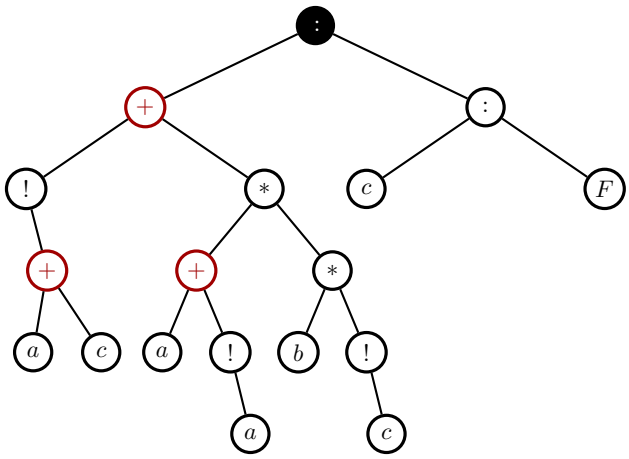


Figure 5.1: Syntax tree of given formula f before it is atomized.

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.

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

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.

$$((a * (b * (!c))) : (c : F)) \Rightarrow \begin{array}{l} a : (X_1 \to (c : F)) \\ (b * (!c)) : X_1 \end{array}$$

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*.

$$(b * (!c)) : X_1 \Rightarrow \begin{array}{l} b : (X_2 \to X_1) \\ (!c) : X_2 \end{array}$$

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*.

$$(!c) : X_2 \Rightarrow X_2 = (c : X_3)$$

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

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

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

Finish Example

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