A Proof Search Implementation in Python for Justification Logic

Bachelorarbeit

der Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

vorgelegt von

Judith Fuog

2015

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

Abstract

Justification Logic as part of the larger field modal logic provides some means to give more information about a proof. Information and researches about this topic are currently still rather limited.

However the thesis presented here does not concern itself with the theoretical details of Justification Logics but focus on a proof search approach for this specific logic. The implementation is done in the language Python.

Contents

Chapter 1

Introduction

1.1 Motivation

Justification Logic is not very common and finding example of how it works it difficult. This implementation provides the possibility to search simple examples for their provability thus providing an easier approach to Justification Logic.

1.2 Goal

The initial goal of this project was to extend the existing proof search engine Z3 [Microsoft Research] to also handle Justification Logic. Among the interface languages provided was Python, a language that interested me for quiet some time already. Deeper investigation into Z3 revealed that to make it handle also Justification Logic the given interface in Python would not work. Instead it would have to be integrated into the core of the program which is written in C. The expenses it would require enough understanding of Z3 to do the this integration would be far too costly and as a consequence would leaf me with very little resources left for the intended implementation.

So instead of extending Z3 from Microsoft Research the actual goal was altered into implementing a stand-alone proof search for Justification Logic. That meant the implementation would be easier since it did not depend on anything else anymore. Conversely a lot of the functionality that was hoped to get from Z3 would have to be implemented as well or discarded entirely. The decision to abandon Z3 entirely was made after I had already started implementation with some prototypes in python. As a consequence Python remained the choice of language even thought there might had been more suitable languages for this task.

It was agreed that 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 all input is exactly formatted in the way excepted. The input will not be checked for syntax error or general typing mistakes by the program.

Output *True* if the formula is provable and *False* otherwise.
Ideally there would be a second output in case the formula is provable

giving a proof.

1.3 Overview

The next chapter starts with a short introduction to Justification Logic. It will go only just deep enough into the theory to gain sufficient understanding of the given task.

The third chapter introduces the algorithm used in the implementation on a abstract level. This thesis concerns itself more with the practical side of implementation and not the theoretical side of mathematical logic theory. There will be little to no proof here, but instead focus on examples to illustrate how the algorithm works.

The forth chapter provides a selected insight into the classes and methods of the source code. For a thorough study it is however recommended to take a look at the source code itself as this chapter only covers the essentials.

The fifth chapter combines the previous two chapters by going through an example from start to end.

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.

write last chapter before you summarize it.

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 a Justification Formula

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.*

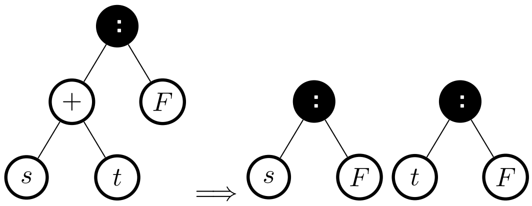
To make the content presented here more understandable the following example will illustrate the steps taken.[3]

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.

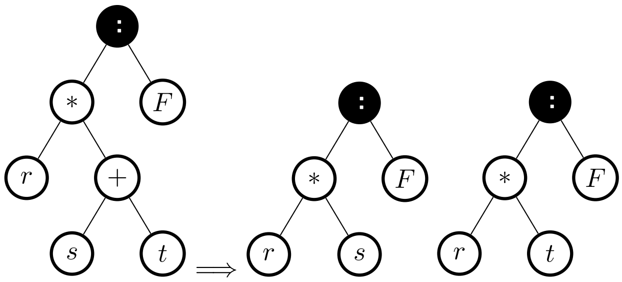
[3]It is on purpose that the *justification term* is by far more complicated than statement $b : F$ that follows the *justification term.* As far as this algorithm goes the complexity of the statement is of no further consequence and thus is kept as simple as possible to allow a easier overview.

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



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

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

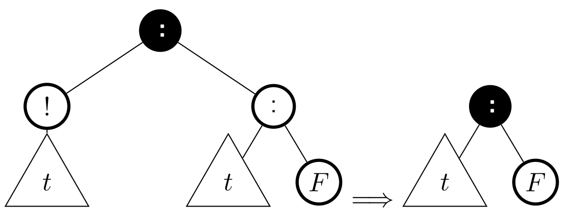


Simplify Bang

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:

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



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.

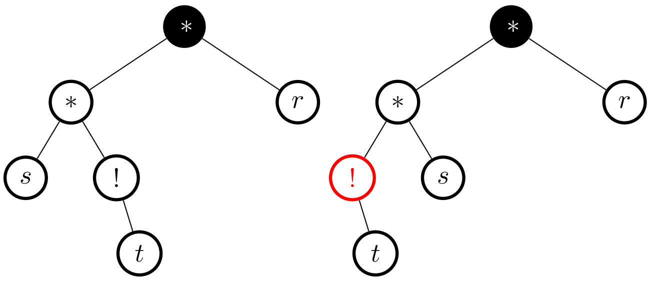
Remove Bad Bang

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.*

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

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.

3.2.2 Get the Must Terms

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.

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:

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

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

3.3 Conquer

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 Matching with CS-List

Central for the whole *conquer* part is the approach how two formulas are compared with each other and what to do with the result. This will be needed when we first try to match our *musts* with what we find in the cs-list and later again when we assemble the different conditions and merge them to configurations.

For one atom we have now several *musts*, each of these musts corresponds to a proof constant and holds a term usually made up from at least one variable (*X-wilds*). On the other hand the terms we find within the cs-list are not only terms with constants but also axioms that can contain variables (*Y-wilds*) as well. This means that the result of a comparison of such two formulas are conditions that apply to certain variables.

If we compare the term $(X_2 \to (X_1 \to F))$ of a must with the term $(Y_1 \to (Y_2 \to Y_1)$ from the cs list for example, we get:

$$\begin{aligned} X_1 &: \{Y_2\} \\ X_2 &: \{Y_1\} \\ Y_1 &: \{X_2, F\} \\ Y_2 &: \{X_1\} \end{aligned}$$

Which can be shorted without loosing any informations to:

$$\begin{aligned} X_1 &: \{Y_1\} \\ X_2 &: \{F\} \\ Y_2 &: \{F\} \end{aligned}$$

So for every entry which we compare to our *must* gives us a set of *conditions* on the occurring variables. Each set represents a possible proof for the *must*, but since all *musts* have to be proofed and since they contain variables that also occur in other *musts* the sets of conditions for one *must* has to be combined with all the sets of the other *musts*.

3.3.2 Merging Conditions to Configurations

Suppose we have *musts* $m_1, m_2, ..., m_n$ for a certain atom. From the previous step each of these a_i has at least[4] one set of conditions, possible more. Our aim no is to find one set of conditions for each *must* such that when we put all those conditions together we will have not contradiction. This will give us the final configuration of the variables[5].

Let us say we have for the *musts* m_k and m_{k+1} the following sets of conditions: For m_k we find only one set, for m_{k+1} we shall have two.

$$\begin{aligned} m_k &: [\{X_1 : \{(A \to X_3)\}, X_2 : \{A\}\}] \\ m_{k+1} &: [\{X_1 : \{(X_2 \to B)\}, X_4 : \{X_3\}\}, \\ &\quad \{X_1 : \{X_2\}, X_4 : \{B\}\}] \end{aligned}$$

We see already that the first set of a_j is compatible with the set of a_i and the second set of a_j is not. To archive the same result with the algorithm the two conditions are are joined:

$$\begin{aligned} m_k \cup m_{k+1} : [&\{X_1 : \{(A \to X_3), (X_2 \to B)\}, X_2 : \{A\}\}, X_4 : \{X_3\}\}, \\ &\{X_1 : \{(A \to X_3), X_2\}, X_2 : \{A\}\}, X_4 : \{B\}\}] \end{aligned}$$

For the first set of condition we get from the join, resolving the conditions for X_1 will give us $X_2 : A$ which fits with the condition for X_2 that is already present and $X_3 : B$ which will give us also $X_4 : B$.

[4]If there is not entry in the cs-list that matches the criteria of a *must* it makes the whole atom unprovable.

[5]We are only concerned for the *X-wilds* but we will still need to tag the *Y-wilds* along.

In the second set the resolve of the conditions does not work out. From X_1 we get that $X_2 : (A \to X_3)$ which is not compatible with the existing condition on X_2 that states $X_2 : A$.

So as result from the merge above we will get

$$m_k \cap m_{k+1} : [\{X_1 : \{(A \to B)\}, X_2 : \{A\}\}, X_3 : \{B\}, X_4 : \{B\}\}]$$

Merging one *must* after another until all m_n are included in will either give us the configuration of the variables eventually or fail because there is no possible configuration for this atom.

3.3.3 Analyzing the Results

In the end we get for each atom from the original formula a set of possible configurations. A set may contain several configurations, meaning that the variables of this atom can be configured differently, it may contain only one configuration or none at all, meaning that there are no valid configurations for the variables of this atom.

Sine proofing one atom of a formula proves the whole formula, the last step taken by the algorithm is to check if at least one atom is provable. In theory the algorithm could stop as soon as it finds the first provable atom, but in this implementation is checks all the atoms and aside from giving a simple `True` or `False` it provides also the configuration of the variables for all valid atoms.

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

Finding a good model design for this algorithm was rather hard. I ended up with a few model classes in a traditional sense and the inevitable helper class that contains a bunch of static methods. I think for a very clean design all of the source code would best be put in one, or at most two classes and be shipped as a single module. As can be seen in the simplified UML presented below there is no real interaction between the classes. They mainly serve the purpose to hide complicated code and provide a certain level of abstraction. For that reason all, or most of it could be included in the master class `ProofSearch`. But I find it more agreeable to browse through different files of code than to have it all clustered up in one big file.

I believe that the reason for this situation of design it the fact that the code altogether represents a single algorithm and thus it is not as intuitive to model as other things where the domain implies more straight forward objects with corresponding responsibilities and relations.

On the other hand it shall be pointed out that it surely would be possible to structure this project in a more object-oriented style. But doing so would probably cost more time and effort then what eventually would be won by it.

4.1.1 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, not needing 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.

4.1.2 Classes

`Tree` and `Node`

As seen in the previous section I use a binary tree to represent, search and manipulate formulas. The class `Tree` and also the associated class `Node` are a standard implementation of a syntax tree made to precisely fit my purpose.

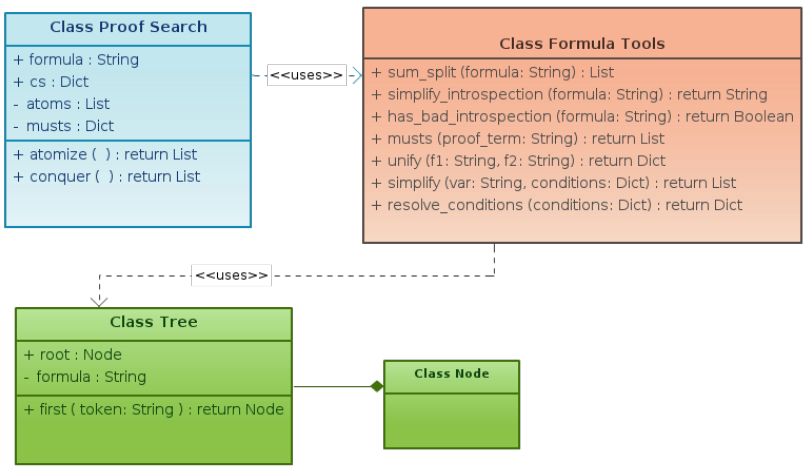


Figure 4.1: Simplified UML graphic of the classes used for implementing the algorithm. The list of methods and attributes is by no means complete and should simply give an idea of the construction.

`ProofSearch`

This class can be considered as the main class of this project. It takes the user input, evaluates the justification formula and finally returns if the formula is provable or not. If it is, the output will also provide a proof.

`FormulaTools`

The name of the class reveals already its usage. This class doesn't quiet deserve to be called such since it does not describe any model but simply serves as a box of tools. Tools which perform functions that are not solely related with the model `Tree` and `Node` or with `ProofSearch`. It is something like a go-in-between for the those. Removing this class would result in a lot of static methods for both the `Tree` and the `ProofSearch` class and for many of them it would not even be clear where to.

But it is still possible to describe it as a model its responsibilities. It main concern lays in handling formula and any task given for one or a set of formulas. In contrast to that the `Tree` is only concerned with the formula that makes up its structure and the `ProofSearch` that does not actually handle formulas but evaluates the result a action on a formula gives.

4.2 Selected 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. The source code of the methods presented here are excerpts only and occasional their in line comments where shortened or to keep the snippets as short as possible and to avoid unnecessary repetition of information since the code comments are of course very similar to the description presented in this chapter. The aim of this section is to provide a insight of the source code without the need to read through all of it.

Still the source code attached is provided with a full documentation, not only containing method description but also rich in line comments, examples and explanations.

4.2.1 `atomize, ProofSearch`

The method `atomize` can be seen as the whole *divide* step of the algorithm. Although I was tempted to name it as such I did not change it although it would have fitted very nicely with the corresponding method `conquer` which will be presented here as well. I felt that the name *atomize* carries more meaning than *divide* and after all the *divide* and *conquer* approach is more a general one and does not fit a hundred percent for this algorithm.

The method is straight forward, doing what has been described before in chapter 3: It splits the formula for each + found and then tries to simplify all subformulas that start with a !. Subformulas that are not resolvable[1] are removed, leaving those that we call *atoms*.

[1]Such would be formulas that start with a ! but cannot be simplified or formulas that contain a ! on the left side of a $*$.

```
...
if proof_term.root.is_leaf():
  consts.append((str(proof_term), str(subformula)))
elif proof_term.root.token == '*':
  left = subtree(proof_term.root.left)
  right = subtree(proof_term.root.right)
  todo.append(Tree('(%s:(X%s->%s))' %
        (str(left), str(v_count), str(subformula))))
  todo.append(Tree('(%s:X%s)' %
        (str(right), str(v_count))))
  v_count += 1
elif proof_term.root.token == '!':
  left = subtree(f.root.left.right)
  s = '(%s:X%s)' % (str(left), str(v_count))
  todo.append(Tree(s))
  assignments.append((str(subformula), s))
```

Figure 4.2: Excerpt *atomize* from *ProofSearch*.

4.2.2 musts, FormulaTools

The method *musts* expects a given proof term to be *atomized* already as it only distinguishes between ! and $*$ operations. The algorithm takes the formula 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* with a new, the current i for a new *X-wild* X_i is stored and increased in `v_count`.

```
...
if proof_term.root.is_leaf():
  consts.append((str(proof_term), str(subformula)))
elif proof_term.root.token == '*':
  left = subtree(proof_term.root.left)
  right = subtree(proof_term.root.right)
  todo.append(Tree('(%s:(X%s->%s))' %
        (str(left), str(v_count), str(subformula))))
  todo.append(Tree('(%s:X%s)' %
        (str(right), str(v_count))))
  v_count += 1
elif proof_term.root.token == '!':
  left = subtree(f.root.left.right)
  s = '(%s:X%s)' % (str(left), str(v_count))
  todo.append(Tree(s))
  assignments.append((str(subformula), s))
  v_count += 1
```

Figure 4.3: Excerpt *musts* from *FormulaTools*

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.

4.2.3 `unify, FormulaTools`

I spend probably most of my implementing time on this method, or rather on many 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 final implementation no difference is made between handling *X-wild* or *Y-wild* variables on that level. Only much later when all results are put together will those variables be handled accordingly.

The method takes two formulas[2] and compares them on the basis of their tree structure. If roots of both trees are operations and matching, the children of both Nodes are pushed on a stack to be further compared later on. If either one of the trees being compared consists only of a leaf that does not match the other node we either have found a contradiction or a *condition*. In the first case `None` will be return the method is stopped. In the other 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.

All those conditions are stored as tuples in a set 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 only to collect those and not to valuate them. This will done by the method `simplify` and in a further extension also in the method `resolve_conditions`.

```
...
while stack:
  f1, f2 = stack.pop()
  # If the root node is the same (either operation or constant)
  if f1.root.token == f2.root.token:
    # If the its a operator, go on. If it's a constant we're done.
    if f1.root.token in ['->', ':']:
      stack.append(
        (subtree(f1.root.left), subtree(f2.root.left)))
      stack.append(
        (subtree(f1.root.right), subtree(f2.root.right)))
  # If the root is not the same, either it is a mismatch,
  # or there is at least one variable.
  elif f1.is_wild() or f2.is_wild():
    result.append((str(f1), str(f2)))
  else:
    return None
return condition_list_to_dict(result)
```

Figure 4.4: Excerpt $unify$ from $FormulaTools$.

[2] It is expected that the only occurring operations are $\rightarrow$ 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.

4.2.4 `simplify, FormulaTools`

The aim of this method is that after it has run there is only one condition term left for the variable it takes as input and this variable does not occur anywhere else except as key to its condition. For example, if we had $[(A \to B), X_2, (Y_1 \to Y_2)]$ as conditions for the variable X_1 and $[(X_1)]$ as condition for X_2 after running the method we would get $[(A \to B)]$ for X_1, $[((A \to B), (Y_1 \to Y_2)]$ for X_2 and also $[(A)]$ for the new found variable Y_1 and $[(B)]$ for the new found variable Y_2. Thus we have eliminated all occurrences of the variable X_1 and as a consequence of this we found new variables that were not present before.

Implementing this method proved harder then first excepted since I didn't anticipated the role of the new found variables at first. The method `resolve_conditions` handles the order in which this method is called on each variable. It simply pushes the new found variables on top of its stack to make sure they are not forgotten. Because `resolve_conditions` needs to know the new variables `simplify` makes changes to the condition set in place and instead of returning the conditions, it returns a list with all newfound variables.

```
...
# Unify each with another: Gives new conditions.
# If any match returns None, we have a contradiction and stop.
new_conditions = defaultdict(set)
for f1, f2 in itertools.combinations(fs, 2):
  conditions_unify = unify(f1, f2)
  if conditions_unify is None:
    return None
  new_conditions.update(conditions_unify)

# Keep one of the (X1, Fi) and replace all X1
# in the Fis of the other Variables.
# X1 will only occur as the chosen one.
chosen = fs.pop()
for key in conditions:
  conditions[key] = set(
        item.replace(var, chosen) for item in conditions[key])

# Add the chose one and the new conditions to old conditions.
# Collect new variables to return.
conditions[var].update([chosen])
new_vars = []
for key in new_conditions:
  if not conditions[key]:
    new_vars.append(key)
  conditions[key].update(new_conditions[key])
...
```

Figure 4.5: Excerpt *simplify* from *FormulaTools*.

4.2.5 `conquer, ProofSearch`

Although the method `conquer` is the one returning the final result, the actual work is done by the method `conquer_one_atom`. As the name suggests it checks the provability of one atom only. `conquer` then simply summarizes the result of each atom and give a readable output.

`conquer_one_atom` is structured in two main loops. In the first loop it collects all possible configurations for each of the *musts* of the atom. If for any *must* no valid configuration can be found the method will terminate because one invalid *must* makes the whole atom unprovable.

```
...
# Collect all conditions for each must.
# Example: a - proof_constant, X1->F - condition_term
for proof_constant, condition_term in self.musts[atom]:

  proofs_for_atom = []

  for cs_term in self.cs[proof_constant]:
    configuration = match_with_cs_term(
                                    cs_term, condition_term)

    if configuration is not None:
      proofs_for_atom.append(configuration)

  if proofs_for_atom:
    all_conditions[(proof_constant, condition_term)]
                 = proofs_for_atom
  else:
    return None
...
```

Figure 4.6: First excerpt *conquer_one_atom* from *FormulaTools*.

The second loop then tries to find one (or more) overall configuration(s) that is compatible with at least one of the configurations per *must*. If at some stage there is no entry left in `merged_conditions`, it means that the conditions posed by the new encountered configuration of the musts are not compatible with the old ones and thus the atom is not provable.

```
...
merged_conditions = []
for must in self.musts[atom]:
  merged_conditions = merge_conditions(
        all_conditions[must], merged_conditions)

  if merged_conditions is None:
    return None
...
```

Figure 4.7: Second excerpt *conquer_one_atom* from *FormulaTools*.

4.3 Tests

The simple unittests I have written for this algorithm have been most important to the success of it. They served me in two ways: First to check if my code would behave and actually do what I expected it to do and second when I suddenly stumbled across a example or a situation where I did not know what I would expect I would simple write a test for it and see what happens, 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 `summarize` 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`.

[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.

5.2.1 Sumsplit

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

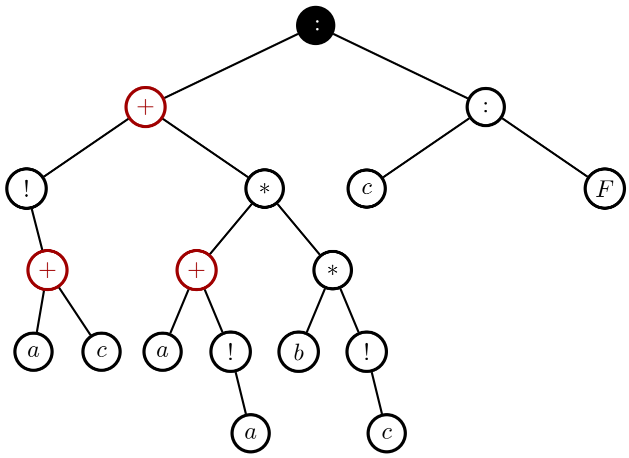


Figure 5.1: Syntax tree of given formula f before it is atomized.

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

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

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

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

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

5.2.2 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 the Musts

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

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 a_1 (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 a_2 (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.

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

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*.

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

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*.

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

This finally gives us all the *musts* for a_2. As can be seen belove 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*.

From the previous steps we have now two atoms with their *musts* which we will check for provability.

5.3.2 Using the CS-List

As can seen immediately when looking at the cs-list the atom $a_1 : (c : F)$ is not provable. The atom itself is already the *must* that we need to check for in the cs-list. Since there is no entry F in cs for the proof constant c the atom is not provable and we are done.

The last remaining atom a_2 needs a little bit more work to. First we will select and compare all *musts* of a_2 with the corresponding entries in cs and then we need to find a configuration for the variables of the musts, that will fit all *musts*.

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

Proof Constant a

Comparing $(X_1 \to (c : F))$ with all entries in cs for the proof constant a will give us the following two condition set which are only on the variable X_1.

$$(H \to (c : F)) \quad \Rightarrow \quad \{X_1 : H\} \tag{5.11}$$

$$((E \to (c : D)) \to (c : F)) \quad \Rightarrow \quad \{X_1 : (E \to (c : D))\} \tag{5.12}$$

Proof Constant b

For the proof constant b with *must* term $((c : X_3) \to X_1)$ we get:

$$((c : F) \to H) \quad \Rightarrow \quad \{X_1 : F, \qquad X_3 : H\} \tag{5.13}$$

$$((c : D) \to (a : F)) \quad \Rightarrow \quad \{X_1 : (a : F), \qquad X_3 : D\} \tag{5.14}$$

$$((Y_1 \to (Y_2 \to Y_1)) \quad \Rightarrow \quad \{X_1 : (Y_2 \to Y_1), \qquad Y_1 : (c : X_3)\} \tag{5.15}$$

We note that for the last condition set we now have also another kind of variable aside from those given by the *must* term. For the moment both kind of variables are treaded exactly the same.

Proof Constant c

Since the must term for proof constant c is simply X_3 we get the following conditions

$$(c : F) \quad \Rightarrow \quad \{X_3 : (c : F)\} \tag{5.16}$$

$$G \quad \Rightarrow \quad \{X_3 : G\} \tag{5.17}$$

$$D \quad \Rightarrow \quad \{X_3 : D\} \tag{5.18}$$

$$(G \to F) \quad \Rightarrow \quad \{X_3 : (G \to F)\} \tag{5.19}$$

5.4 Constructing the Final Result

5.4.1 Merging Conditions

Our aim is that we pick one line from each proof constant and that this merged conditions give us a configuration for the X-variables. For example we could pick from each the top line, but it is obvious that this is not a solution since X_3 can only be either H or $(c : F)$ but not both.

It is clear that not every line of a can be successfully merged with every line of b. We see that we can only take those that have the same term for X_3 or there is a Y-variable. If fact only the two bottom row are compatible, since no entry form b fits $X_1 : H$ from a and only $(Y_2 \to Y_1)$ can be matched with $(E \to (c : D))$.

$$a \cap b : \quad \{X_1 : (E \to (c : D)), \quad X_1 : (Y_2 \to Y_1), \quad Y_1 : (c : X_3)\} \tag{5.20}$$

As we see above there are now two conditions that apply to the variable X_1. Before we move on and try to merge this set of conditions with one of the lines of c we will resolve the current conditions as far as possible.

Comparing the conditions on X_1 we find that $Y_2 : E$ and $Y_1 : (c : D)$. Since we have already a condition for Y_1 that condition is now compared with the new we got from X_1 and we will get $X_3 : D$. Thus all our variables are now configured:

$$\{X_1 : (E \rightarrow (c : D)), \quad X_3 : D, \quad Y_1 : (c : D), \quad Y_2 : E\} \tag{5.21}$$

As a consequence of merging line (5.12) from a with line (5.15) from b there is no choice left for the variable and the final result depends on finding a line from proof constant c that matches the value for X_3 and as it happens this is the case for line (5.18).

5.4.2 Meaning of the Result

Since we found a valid configuration for the atom a_1 (5.7)) we have shown that the formula f (5.1)) is provable. But at this point I would like to show what finding the X-variables has to do with showing the provability of f.

From our previous step we have a configuration for every variable. We are however only interested in the X-variables and do not care further about the Y-variables. So we know that $X_1 = (E \rightarrow (c : D))$ and $X_3 = D$. If we replace that in the *musts* for all of the proof constants we get the following:

$$\begin{aligned} a_2 : \quad & [(a : ((E \rightarrow (c : D)) \rightarrow (c : F))), \\ & (b : ((c : D) \rightarrow (E \rightarrow (c : D)))), \\ & (c : D)] \end{aligned} \tag{5.22}$$

As can be seen these entries can all be found precisely like that in the cs-list. Also from those we can reconstruct the term of a_2:

$$(c : D) \tag{5.23}$$
$$((!c) : (c : D)) \tag{5.24}$$
$$((b * (!c)) : (E \rightarrow (c : D))) \tag{5.25}$$
$$((a * (b * (!c))) : (c : F)) \tag{5.26}$$

And with (5.26)) for a_2 we have again with what we started right after the atomization step in (5.5). In the graph below the path with the tree of the atom a_2 is highlighted.

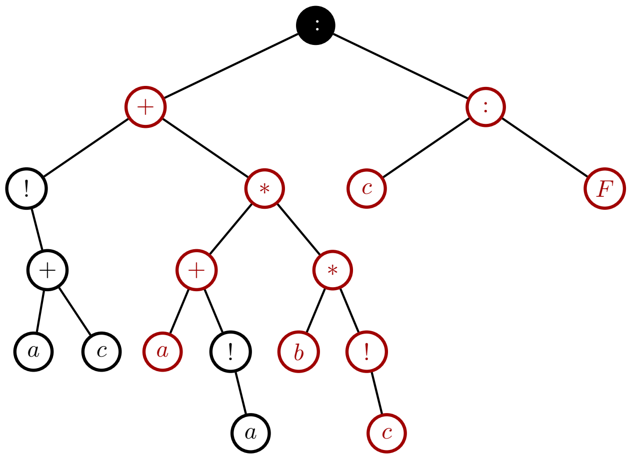


Figure 5.2: Syntax tree of formula f with atom a_2 hightlighted.

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

Until very recently this step (merging and matchin) 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.

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