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 [3] 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.

Chapter 2

Justification Logic

The theory of justification logic as it is used here requires little knowledge of the wide field of modal logic apart from very basic 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 based mainly on the work of Goetschi [4] as well as the older reference Artemov [1] and also from the homepage [2]. These 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 *known* 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 gives us the notion that *knowledge* or *proofs* may come from different sources. Justification logic lets us connect different *proofs* with a few simple operations and thus gives us a better description of the proof. To quote Goetschi [4]: It may be said that where in model logic the knowledge is implicit it is explicit in justification logic.

2.2 Rules and Definitions

The language of justification logic is given here in a more traditional form with *falsum* and *implication* as primary propositional connectives. Although for the work done with this implementation only the implication is used and the falsum has been ignored. Also not all available syntactic objects 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 by the following grammar:*

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

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 and t a term, then $t : F$ is also a formula.

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

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

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 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 The Core Idea

In my earliest attempts the methods of my algorithm had the tendency to explode with the number of `if-else` and `switch` statements. Also they were always very deep nestled. It was sheer impossible to keep track of what had to be done where under which circumstances and whenever I though I had it I found more cases that needed takes special care of. What I really needed was a strategy. I started experiencing with the proof terms of the justification formula, trying to take it somehow apart and restructure the formula in a way that would make handle it easier. I was looking for something like the conjunctive normal form (CNF) and the way how it is used in proof search calculus[1]. Indeed I found a way that allows me to *divide* a justification formula in disjunctive formulas where proving only one of them is also proof for the whole justification formula. The main advantage gained from dividing the justification formula is that the resulting formula have far less variety in the manner of their operations and thus are easier to further analyze.

The comprehension that my approach follows a classic *Divide and Conquer* approach came to me only later when I started *conquering.* The algorithm presented here may not be a model of *Divide and Conquer* but similarities cannot be denied. For that reason I have structured this chapter accordingly.

There are two major steps in the divide part of this algorithm. First the justification formula itself will be split into several smaller pieces and adjusted. Second each of those smaller pieces called *atoms* is also being be taken apart so that only their proof constant with a corresponding proof term containing variables remains. The pair of proof constant and proof term will be called *must*[2].

The conquer step first handles the *musts* of one atom, trying to find a valid match for every *must* in the constant specification list and then evaluates from

[1] *Proof Search Calculus* as it is introduced by Jäger [5].

[2] They are called *musts* in the algorithm because we have to find a match for every single one of them or else the atom is not provable.

the results of each *atom* the provability of the originally given justification formula.

3.2 Divide

The aim of this first step is to split it into smaller pieces and standardize them to make it easier to get the *musts*.

3.2.1 Atomize a Justification Formula

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

- *The term contains no sum operations.*
- *A introspection operation can neither be the top operation of a term nor be the left operant of a application operation.*

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

Sumsplit

From the sum rule of justification logic in 2.2 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.

$$(s + t) : F \quad \Rightarrow \quad s : F \vee t : F \tag{3.1}$$

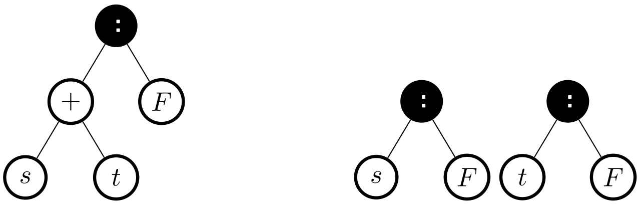


Figure 3.1: Example of a simple sumsplit.

This is also true for formulas where sum is not the top operation. Here X denotes a arbitrary *justification term.*

$$\begin{aligned} (r * (s + t)) : F \quad &\Rightarrow r : X \rightarrow F \wedge (s + t) : X && (3.2)\\ &\Rightarrow (r : X \rightarrow F \wedge s : X) \vee (r : X \rightarrow F \wedge t : X) && (3.3) \end{aligned}$$

[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.

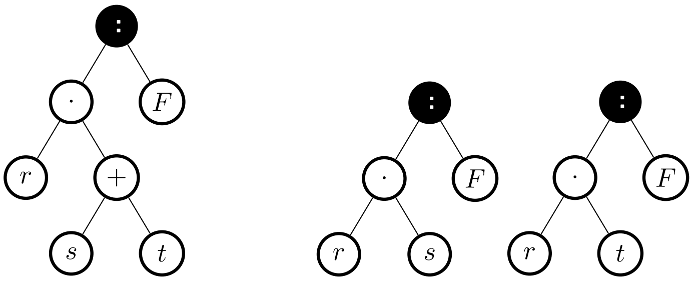


Figure 3.2: Example of a sumsplit where the sum is not the top operation.

Simplify Introspection

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

Derived from the application rule in 2.2 we get the following:

$$!t : (t : F) \quad \Rightarrow t : F \tag{3.4}$$

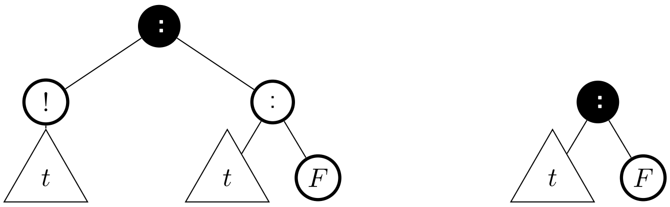


Figure 3.3: Example of a simplification of a introspection.

Speaking in the manner of a syntax tree it needs to be checked, if the child of the introspection 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 introspection operation and the formula has to be discarded.

Remove Contradicting Introspection

This last step in atomizing the formula proved to be on of the hardest to realize. Only countless examples support the claim that the introspection operation must not be the direct left child of a application 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 introspection operation is never the top operation in a formula it is guarantied that a introspection operation must be either a right child or a left child of a application operation.

$$((!s) \cdot t) : F \Rightarrow \exists X_1 : ((!s) : (X_1 \to F) \ \wedge \ t : X_1) \tag{3.5}$$
$$(!s) : (X_1 \to F) \Rightarrow \exists X_2 : ((X_1 \to F) = (s : X_2)) \qquad ↯ \tag{3.6}$$

The last line gives a contradiction since there is no possible X_2 that would fulfill the condition of $X_1 \to F = s : X_2$.

Assertion (Tree Version)**.** *A introspection operation that is the direct left child of a application operation causes the whole term to be invalid (unprovable), given that the term is without sum operations and no introspection operation at the top.*

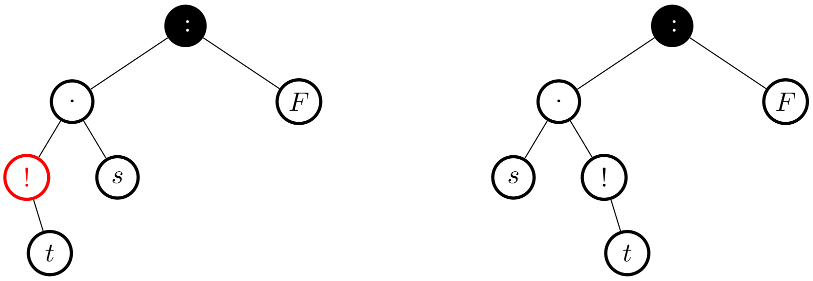


Figure 3.4: The left tree show a introspection that gives a contradiction whereas the right tree is valid.

This concludes the *atomization* of one formula to many simple formulas which can be checked for provability individually. An atomized formula now consists only of application operations and valid introspection operations. The next section will show what further steps are needed to check one atomized formula for its provability.

3.2.2 Finding the Must Terms

To check a justification formula for its provability we need to look up the justification constant from the formula in the constant specification list (from now on called *cs-list* for brevity) and compare the justification term with the term we find there.

The operation rules which were presented in chapter 2.2 gives us the instruction how we can take a formula apart to look up the individual proof terms in the *cs-list*. The rule for the sum operation is described in the previous steps for the `sumsplit` in section 3.2.1.

Each application operation in a term adds one variable. An introspection operation $!t : X_i$ replaces the existing variable X_i with a new term that is of the form $t : X_j$.

So for a term like this $(a \cdot (!b)) : F$ the following is evaluated:

$$\begin{aligned} &(a \cdot (!b)) : F \\ &\qquad\qquad \Rightarrow a : X_1 \to F \\ &\qquad\qquad \Rightarrow !b : X_1 \end{aligned} \tag{3.7}$$

$$\begin{aligned} &!b : X_2 \\ &\qquad \Rightarrow X_1 = b : X_2 \end{aligned} \tag{3.8}$$

X_2 will be replaced by $b : X_2$ so our final *musts* for $(a \cdot (!b)) : F$ looks like this:

$$\begin{aligned} [&\big(a, ((b : X_2) \to F)\big), \\ &\big(b, X_1\big)] \end{aligned}$$

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 that are not determined it is possible that we get more than one match per proof term. Since the *cs-list* allows terms that contain variables as well this imposes further conditions on the possible choice of the proof term. All those possibilities and conditions are collected during the comparison of the musts with the *cs-list*.

Then in the second and most important step in the conquer part those conditions are merged. It is checked if there is a possible combination of the given options so that we have a proof for the atomized formula. The proof of any atom is also a proof of the original formula.

In this section we will merge *conditions* and finally convert them to *configurations*. Since the words *conditions* and *configurations* are similar but distinct let us define them:

Definition 5 (condition)**.** *A condition is a pair (X, T), where X is a variable and T is a term that doesn't contain X. The condition is said to be* on X *and asserts that X is equal to T. T is called the* condition term.

Each variable can have multiple conditions on it that may contradict each other.

Definition 6 (configuration)**.** *A* configuration *is a set of conditions such that*

1. *There is at most one condition on each variable.*
2. *The condition term does not contain any variables.*

3.3.1 Matching with the CS-List

Central for the conquer part is the procedure of comparing two formulas. We use this when we try to match our *musts* with elements of the *cs-list* and again when we find and merge the conditions.

For one atom we have several *musts*, each of which corresponds to a proof constant and holds a term. This term can consist of variables in turn. The terms we find within the *cs-list* are not only terms with constants but axioms containing variables as well. This means that the result of a comparison of two formulas are conditions.

If for example 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*, we get the following conditions:

$$X_1 : \{Y_2\}, X_2 : \{Y_1\}, Y_1 : \{X_2, F\}, Y_2 : \{X_1\}$$

Which can be shorted without loosing any informations to[4]:

$$X_1 : \{Y_1\}, X_2 : \{F\}, Y_2 : \{F\}$$

Every entry in the *cs-list* that we compare to our *must* gives us a set of conditions for the occurring variables. Each set represents a possible proof for one *must*, but since all *musts* must be proven and contain variables that also occur in other *musts* the sets of conditions of all *musts* of an atom have to be merged together.

3.3.2 Merging Conditions to Configurations

Suppose we have *musts* $m_1, m_2, \dots, m_n$ for an certain atom. From the previous step each of these m_i has at least one set of conditions[5] for its variables. Our aim is to find one set of conditions for each *must* such that those conditions contain no contradiction. This gives us the final configuration of the X_i variables[6].

Let us say we have the *musts* m_k and m_{k+1} and the following sets of conditions:

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

We see that the set of m_k is compatible with the first set of m_{k+1} and the second set of m_{k+1} is not.

To algorithmically archive the same result the two conditions are first simply joined, ignoring possible contradictions. This gives us two new sets of conditions.

$$\begin{aligned} &\{X_1 : \{(A \to X_3), (X_2 \to B)\}, X_2 : \{A\}\}, X_4 : \{X_3\}\}, \\ &\{X_1 : \{(A \to X_3), X_2\}, X_2 : \{A\}\}, X_4 : \{B\}\} \end{aligned}$$

For the first set of condition we get from the join, resolving the conditions for X_1 gives us $X_2 : A$ which also fits with the condition for X_2 that is already present. Further $X_3 : B$ gives us also $X_4 : B$. If the variables that we find in m_i and m_j are all that occur in all other *musts* of the atom we have found a configuration for the variables, thus proving the atom.

$$\{X_1 : \{(A \to B)\}, X_2 : \{A\}\}, X_3 : \{B\}, X_4 : \{B\}\}$$

In the second set resolving 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 $X_2 : A$. Consequently the second set is discarded. If the first set had failed as well there would be no proof for this atom.

[4] This is only one of many options to shorten the conditions, another option would be $Y_1 : \{X_2, F\}, Y_2 : \{X_1\}$.

[5] If there is no entry in the *cs-list* that matches the criteria of the *must* it makes the whole atom unprovable.

[6] We are only concerned for the X variables but we still need to tag the Y variables along.

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 containing several configurations means that the variables of this atom can be configured in multiple ways. If it contains only one configuration it means that there is only one possible configuration. Finally if there is none at all, it means that there are not valid configurations for the variables of this atom thus making it unprovable.

Since proving one atom of a formula proves the whole formula 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(s) of the variables for all provable atoms.

This concludes the whole divide and conquer chapter. I personally have found it rather easy to understand the individual steps but difficult not to 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 from start to end to help understanding it better.

Chapter 4

Implementation

4.1 Model Overview

Finding a good model design for this algorithm proved a rather hard task. I ended up with a few model classes in a traditional sense and the inevitable helper class that contains only 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 the interaction between the classes is very limited and usually only one-way. They mainly serve the purpose to hide complicated code and provide a certain level of abstraction and modularity. For that reason all or most of it could be included in the master class `ProofSearch`. But I find it more comfortable 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 is the fact that the code altogether represents a single algorithm and thus it is not as intuitive to break into smaller pieces as other things where the domain implies more straight forward objects with corresponding responsibilities and relations.

On the other hand it should be pointed out that it would be possible to structure this project in a more object-oriented style. But reimplementing it would probably cost more time and effort than what would be won by it.

4.1.1 Operation Syntax Tree

One of the earliest challenges was to find 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 a few libraries from Python.

So while I was searching yet for another library 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 *Datastructures 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 such that it interprets the order of operations but with what I remembered from the lecture manipulating a formula in form of a tree, to delete or swap subtrees becomes very easy.

I decided to implement my own tree for that purpose. It might be argued that a lot of work could have been saved if I used available syntax trees libraries 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 probably would have been 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 introspection 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 additional case handling for the introspection operator.

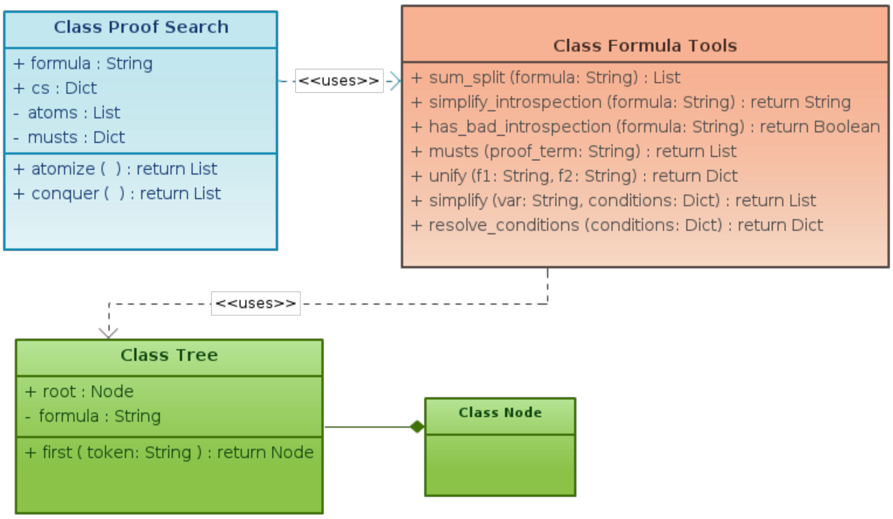
The tree was not only important to the algorithm, but could also be used to check if the input was written correctly. Therefore most of the tests that test the string handling of a tree are the result of formulas used somewhere else but which needed syntax checking.

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 the associated class `Node` are simple 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. Additional helper methods and attributes are hidden.



Class Proof Search
+ formula : String
+ cs : Dict
- atoms : List
- musts : Dict
+ atomize ( ) : return List
+ conquer ( ) : return List
<<uses>>
Class Formula Tools
+ sum_split (formula: String) : List
+ simplify_introspection (formula: String) : return String
+ has_bad_introspection (formula: String) : return Boolean
+ musts (proof_term: String) : return List
+ unify (f1: String, f2: String) : return Dict
+ simplify (var: String, conditions: Dict) : return List
+ resolve_conditions (conditions: Dict) : return Dict
<<uses>>
Class Tree
+ root : Node
- formula : String
+ first (token: String) : return Node
Class Node

`ProofSearch`

This class can be considered as the core logic 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 contain a proof.

`FormulaTools`

The name of the module reveals already its usage. This module doesn't quite deserve to be called a class since it does not describe any model but simply serves as a box of tools that perform functions which are not solely related with the model `Tree` and `Node` nor with `ProofSearch`. It is something like a go-in-between for those. Removing this module would result in a lot of static methods for both the `Tree` and the `ProofSearch` class and for many of them it would not be clear where they would fit best.

Its core responsibility lies modifying and analysing formulas. In contrast to that the `Tree` is only concerned with the single formula that it describes and the `ProofSearch` does not actually handle formulas itself but only evaluates the result of an action on a formula given and decides how to proceed.

4.2 Selected Methods

In this section I want to show and explain some of the more important and thus usually more complicated methods that 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 in order to keep the snippets as short as possible and to avoid unnecessary repetition of information since the code comments are 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 the attached source code.

4.2.1 `atomize, ProofSearch`

The method `atomize` can be seen as the whole *divide* step of the algorithm. I was tempted to name it so but 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 *divide* and *conquer* is more a general approach and does not fit a hundred percent this algorithm.

The method is a straight forward implementation of the algorithm in chapter 3.2. It splits the formula for each application found and then tries to simplify all subformulas that start with a introspection. Subformulas that are not resolvable [1] are removed, leaving only those that we call *atoms.*

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

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

```
...
# first step: make sum-splits
splits = sum_split(self.formula)

# second step: simplify formula if top operation is !
for formula in splits[:]:
  splits.remove(formula)
  new_formula = simplify_introspection(formula)
  if new_formula:
    splits.append(new_formula)

# third step: remove formulas where ’!’ is left child of ’*’
for formula in splits[:]:
  if has_bad_introspection(formula):
    splits.remove(formula)
return splits
```

4.2.2 musts, FormulaTools

The method *musts* expects a given proof term to be atomized already as it only distinguishes between introspection and application 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 resolution of a application operation introduces a new X variable and the resolution of a introspection operation replaces an existing X variable with a new one, the current i for a new X-*wild* X_i is stored and increased in v_count.

Figure 4.3: Excerpt *musts* from *FormulaTools*

```
...
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
```

If for example the current justification term is $((a \cdot (!b)) : F)$, it will taken apart into the two subformulas $(a : (X_i \to F))$ and $((!b) : (X_i))$. Further since from $!b : X_i$ it follows $\exists X_j \quad s.t. \quad !b : (b : X_j)$, all X_i that occurred up to now must be replaced by $(b : X_j)$. These values are stored in `assignments` and eventually replaced.

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 predecessors. 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. In this final implementation X or Y variables are handled the same on this level.

The method `unify` takes two formulas [2] as input and compares them on the basis of their tree structure. If the 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 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 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 python dictionary, where all conditions for one variable can be accessed by the variable itself as key. At the current state conditions that apply to the same variable may contradict each other, but this method is responsible only for collecting conditions and not evaluating them. This will done by the method `simplify` and in a further extension also in the method `resolve_conditions`.

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

```
...
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)
```

[2] It is assumed that the only occurring operations are → and :. It would be easy to extend the code at this point 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[3] it takes as input and this variable does not occur anywhere else except as key to its condition. For example, if we have $[(A \rightarrow B), X_2, (Y_1 \rightarrow Y_2)]$ as conditions for the variable X_1 and $[(X_1)]$ as condition for X_2 after running the method we get $[(A \rightarrow B)]$ as the only condition for X_1, $[((A \rightarrow B), (Y_1 \rightarrow Y_2)]$ 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 than first expected 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 given more priority. Because `resolve_conditions` needs to know the new variables the method `simplify` makes changes to the condition set in place and instead of returning the conditions as it might be expected, it returns the list of newfound variables.

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

```
...
# 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])
...
```

[3] In the source code the combination of the variable and its single condition is referred to as *the chosen one*.

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 unprovable *must* makes the whole atom unprovable.

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

```
...
# 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
...
```

The second loop then tries to find an overall configuration 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.

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

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

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

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 an 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. It is also because of the tests that I have found so many mistakes, some minor others essential, making me recode bits and parts again and again at a stage when I had hoped to be done with the coding.

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 like `conquer` or the `__str__` from `Tree` which basically tests if the parsing of the formula works correctly.

To name some numbers, there are currently[4] a total of 66 tests, almost half of which are related to `ProofSearch`.

[4] 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

In this chapter I walk through a complete example covering as many special cases as possible. As such, the justification term we will look at is rather complicated but it will also show how nicely it can be broken down in more simpler atoms.

5.1 Initialization

The algorithm takes a formula f and a cs-list as input. The data presented here is in the same form as it would be entered into the program. Therefore the cs-list is a *python* dictionary and not simple list of pairs and there are more brackets explicitly written then required by convention.

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

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

5.2 Walking in Trees: Atomize

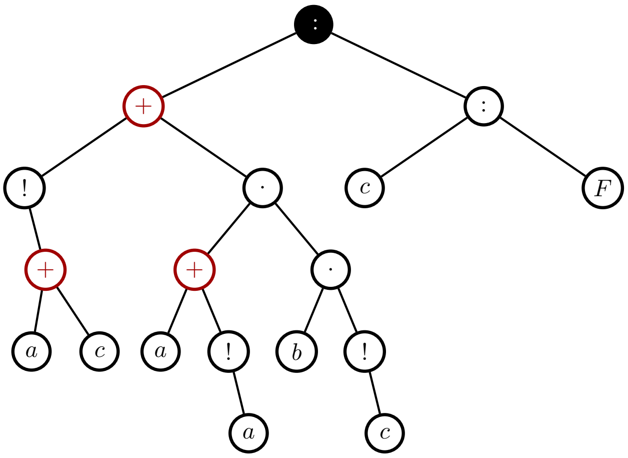
The given formula f is transformed into a syntax tree using `parse_formula` of `Tree`.

5.2.1 Sumsplit

Our first step is to split our formula for every sum we encounter.

$$(((!(a + c)) + ((a + (!a)) \cdot (b \cdot (!c)))) : (c : F))$$

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.

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

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

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

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

Some of those will eventually be the atoms for f, but before they can become atoms of f they need to be simplified and checked.

5.2.2 Introspections

We now look at the terms to discard any that have a introspection operation that is the left child of a multiplication or if there are proof terms that start with a introspection operation and cannot be simplified. If they can be simplified they will of course not be discarded but simplified.

Term (5.3)

In this term we find a introspection 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 this term.

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

Term (5.4)

As before the introspection for the term is valid and 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 introspection operation neither a left child of a multiplication nor as top operation of the proof term and thus we have our second *atom*.

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

Term (5.6)

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

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

This completes the `atomize` step for the formula f giving us the two atoms a_1 and a_2. Showing that at least one of those is provable is enough to show that f is provable.

5.3 Getting and Looking up the Musts

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

We have found the two atoms $a_1$and a_2 for the formula f. The next step determins 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.

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

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

5.3.1 Musts

Atom a_1 (5.7)

Since a_1 consists already only of one proof constant with the corresponding term 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 take apart is a application. Extracting proof constants from a application proof term us give a *X-wild*.

Whenever a new X variable appears the i of X_i will simply be increased by one to ensure that it is fresh in the formula.

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

The proof constant a has been isolated but $(b \cdot (!c))$ still needs to be taken apart further. We repeat the step from above and introduce yet another X variable.

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

Now b has been isolated as well, leaving $(!c)$ as the only unresolved proof constant. Having a introspection in this situation results in a new X variables in combination with the proof term which will replace a previous X variables.

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

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

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

Note that the same proof constant may be in more than one of the *musts* for one *atom*.

5.3.2 Using the CS-List

We now have to look up all *musts* of each atom the see if the atom is provable.

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

The atom a_1 (5.7) is not provable, since its only *must* $c : F$ cannot be found in the cs-list.

The other atom a_2 (5.8) needs a little bit more work. First we select and compare all *musts* of a_2 with the corresponding entries in the cs-list and then we need to find a configuration for the variables of the musts, that will fit all *musts*.

Proof Constant a

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

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

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

Proof Constant b

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

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

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

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

We note that for the last condition set we now have second kind of variable aside from those given in the *must* term. For the moment both kinds of variables are treated exactly the same.

Proof Constant c

Since the must term for proof constant c is simply X_3 we get the following condition sets.

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

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

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

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

5.4 Constructing the Final Result

Now we have several condition sets for each proof constant that have to be put together to a solution.

5.4.1 Merging Conditions

Our goal is to 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 $(Y_2 \rightarrow Y_1)$ can be matched with $(E \rightarrow (c : D))$.

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

As seen 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 for 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:

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

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. Lets take a step back and see, what the X variables have to do with 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 $X_1 = (E \to (c : D))$ and $X_3 = D$. If we replace that in the *musts* for all of the proof constants we get the following:

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

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:

$$(c : D) \tag{5.23}$$

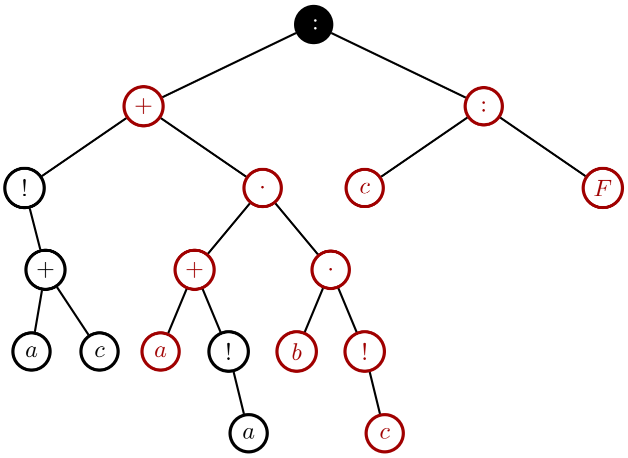
$$((!c) : (c : D)) \tag{5.24}$$

$$((b \cdot (!c)) : (E \to (c : D))) \tag{5.25}$$

$$((a \cdot (b \cdot (!c))) : (c : F)) \tag{5.26}$$

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 showing as much as possible with a concise example.

Chapter 6

Conclusion

Looking back there are many things I would have done differently. There is for example a python module called *Pyparser*[1] . It allows to create and parse simple grammars and it would have served me to get rid of some of the hardcoded string comparison. An other point I often reflected on it the choice of language. There might have more suitable programming languages for this task then python for example a functional language such as haskell.

how to format correctly?

On the other hand there are a lot of things that could be done to further improve this implementation. For user to profit from this implementation it would be advantageous to have a (simple) user interface and even more importantly it would be necessary to implement handling of all formulas and not just such implications. A natural first step is adding negation to archive completeness of the logic structure.

A last point that should be considered is the possible optimization of the implementation of this algorithm. As mentioned already in the introduction no special care has been given to consider the tijme efficiency and the algorithm has not been tested for extremely formulas which might cause the implementation to break.

[1] https://pyparsing.wikispaces.com/

Bibliography

[1] Sergei Artemov. Justification logic. In *Logics in Artificial Intelligence*, pages 1–4. Springer, 2008.

[2] Sergei Artemov and Melvin Fitting. Justification logic. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Fall 2012 edition, 2012.

[3] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[4] Remo Goetschi, FR von Galmiz, and G Jäger. *On the Realization and Classification of Justification Logics*. PhD thesis, PhD thesis, Universität Bern, 2012.

[5] Gerhard Jäger. *Discrete Mathematics and Logic*. Universität Bern, 2011.

Todo list