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 of modal logic provides some means to give more information about a proof. Information and researche about this topic are currently still limited.

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

Contents

Chapter 1

Introduction

1.1 Motivation

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

1.2 Goal

The initial goal of this project was to extend the existing proof search engine Z3 [4] to also handle justification logic. Among the interface languages provided was Python, a language that I was interested in for quite some time already. Deeper investigation into Z3 revealed that to make it also handle justification logic, the interface given would not work. Instead it would have to be integrated into the core of the program which is written in C. The effort required to understand enough of Z3 to do the this integration would be to great and as a consequence would leave very little resources for the intended implementation.

So instead of extending from Microsoft Research's Z3, 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 I 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 language of choice even though there might have 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, given as strings. It may be presumed that all input is exactly formatted in the way excepted. The input will not be checked for syntax errors or general typing mistakes by the program.

Output *True* if the formula is provable and *False* otherwise.
Optionally, there would be a second output in case the formula is provable detailing one or more possible proofs.

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 an 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 no formal proofs here, but instead I will focus on examples to illustrate how the algorithm works.

The forth chapter provides a selection of the classes and methods of the source code. For a thorough understanding 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 the very basics of logic theory. For the purpose of this proof search a few basic rules and definitions are sufficient to provide the knowledge needed.

The theory presented here is based mainly on the work of Goetschi [5] as well as the older reference Artemov [2] and also from the Stanford Encyclopedia of Philosophy webpage [3]. The definitions and rules given here do not fully encompass all of justification logic. Priority was given to those vital to the implementation. However briefly and incompletely the theory is presented here, full reference can be found in the sources named.

2.1 Background

Justification logic has its roots in the field of modal logic. In modal 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:

> [...] justification logic studies explicit knowledge or belief, while modal logic studies implicit knowledge or belief.

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 while the falsum has been ignored. Also, not all available syntactic objects are introduced here - only those implemented.

Definition 1. *Apart from formulas, the language of justification logic has 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 mentioned 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 deeply nested. It was a sheer impossibility to keep track of what had to be done, where and under which circumstances. Whenever I thought I had it, I found more cases that needed special care. What I really needed was a strategy. I started experimenting with the proof terms of the justification formula, trying to take it apart somehow, and restructure the formula in a way that would make it easier to handle. 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 formulas have far less variety in the manner of their operations and thus are easier to further analyze.

The comprehension that mine 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 taken apart so that only their constants with corresponding proof terms containing variables remain. 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 the provability of the originally given justification formula from the results of each *atom*.

[1] *Proof Search Calculus* as it is introduced by Jäger [6].

[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.

3.2 Divide

The aim of this first step is to split the given formula 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.*
- *An introspection operation can neither be the top operation of a term nor be the left operand of a application operation.*

To make the content presented here more accessible the steps necessary will be illustrated through an example.[3]

Sumsplit

From the sum rule of justification logic in 2.2 follows: Checking for provability in a formula where the operation is a sum is equal to checking either operand of the sum for provability. If any of the operands is provable, so is the original formula.

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

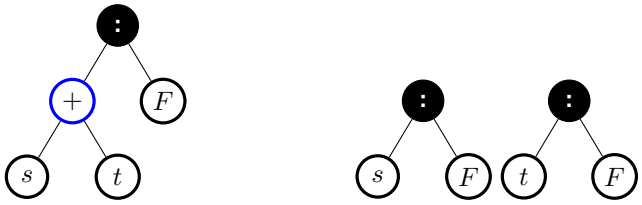

Figure 3.1: Example of a simple sumsplit.

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

$$(r*(s+t)):F \;\Rightarrow r:X \to F \wedge (s+t):X \tag{3.2}$$
$$\Rightarrow (r:X \to F \wedge s:X) \vee (r:X \to F \wedge t:X) \tag{3.3}$$

[3]It is on purpose that the *justification term* is by far more complicated than term F that follows the *justification term*. As far as this algorithm goes, the complexity of the statement is of no further consequence and thus it is kept as simple as possible.

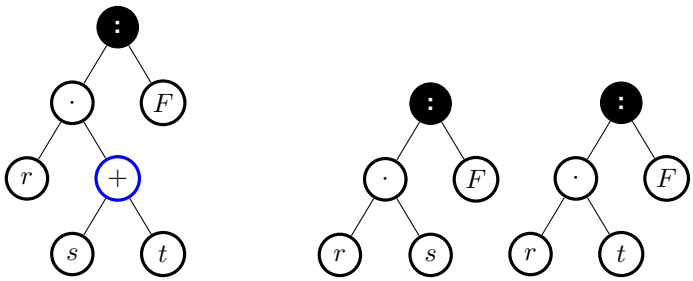

Figure 3.2: Example of a sumsplit where the sum is not the top operation.

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:

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

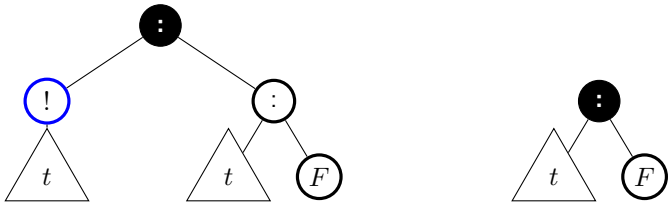

Figure 3.3: Example of a simplification of an introspection: If the child of the introspection operator is identical to the left child of the justification operator of the right subtree, the formula can be simplified to be the right subtree only.

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 operator must not be the left child of an application operation. Coming to that conclusion, it has been helpful that no sum operator could make the situation more complex. Because of this, and also the fact that an introspection operator is never the top operator in a formula, it is guaranteed that an introspection operator must be either a right child or a left child of an application operator.

$$((!s) \cdot t) : F \Rightarrow \exists X_1 : ((!s) : (X_1 \to F) \;\wedge\; t : X_1) \tag{3.5}$$

$$(!s) : (X_1 \to F) \Rightarrow \exists X_2 : ((X_1 \to F) = (s : X_2)) \qquad \lightning \tag{3.6}$$

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)**.** *An introspection operator that is the left child of an application operator causes the whole term to be invalid (unprovable), given that the term is without sum operators and no introspection operator at the top.*

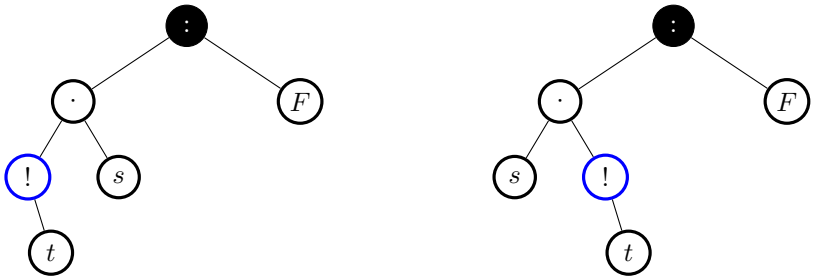

Figure 3.4: The left tree shows an introspection that gives a contradiction, while the right tree is valid.

This concludes the *atomization* of one formula into multiple simple formulas which can be checked for provability individually. An atomized formula now consists only of application operator and valid introspection operator.

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 give us the instructions as to 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 operator 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:

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

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

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

$$\big(a, ((b : X_2) \to F)\big),\ \big(b, X_1\big)$$

3.3 Conquer

Once the *musts* have been obtained we can search the *cs-list* for terms that match it. Since a *must* term usually consists of variables that are not determined, it is possible that we get more than one match. Since the *cs-list* allows terms that contain variables as well, this imposes further conditions on the possible choice of the variables. All those possibilities and conditions are collected during the comparison of the musts with the *cs-list*.

Then, in a second and more challenging step, 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 to 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 can contain multiple proof constants, of which each holds exactly one term. This term can consist of multiple variables. The terms we find within the *cs-list* are not only terms with constants but axioms containing variables as well. This means that results of comparisons between 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 losing any informations to[4]:

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

Every entry in the *cs-list* that we compare our *must* to 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*, a set of conditions of each *must* of an atom have to be merged together.

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

3.3.2 Merging Conditions to Configurations

Suppose we have *musts* $m_1, m_2, \ldots, m_n$ for a 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, giving 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 achieve this result, the sets of conditions are first simply joined, ignoring possible contradictions, giving us two new sets of conditions.

For the first set of conditions 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. Also, $(X_3 : B)$ gives us $(X_4 : B)$. [7]

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

In the second set resolving the conditions does not work out. From X_1 we get $(X_2 : (A \to X_3))$. This is not compatible with the existing condition $(X_2 : A)$. Consequently, the second set is discarded.

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

If the first set had failed as well there would be no proof for this atom.

3.3.3 Analyzing the Results

In the end we get a set of none, one or multiple possible configurations for each atom from the original formula. Getting no configurations at all means that this atom is unprovable. Getting one configuration means that the atom is provable in one way, whereas getting multiple configurations means that there is more than one way to prove this atom.

Since proving one atom of a formula proves the whole formula, an algorithm could stop as soon as it finds the first provable atom, but this implementation checks all the atoms and, in addition to giving a simple `True` or `False` it provides also the configuration(s) of the variables for all provable atoms.

This concludes the Divide and Conquer chapter. I personally found it rather easy to understand the details of the individual steps but difficult not to lose

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

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

[7] If all the variables occurring in all the other musts are $X1$ to $X4$ we have found a configuration, thus proving the atom.

sight of the big picture. For that reason chapter 5 will cover one single example designed to show all aspects of the algorithm and run through it from start to end to help the reader understand.

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 a bunch of static methods. I think for a 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 represents a single algorithm and thus is not as intuitive to break into smaller pieces.

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 would be won by doing so.

4.1.1 Operation Syntax Tree

One of the earliest challenges was to find a useful representation of formulas with which I could work decently.

I looked through different Python libraries, but those I considered didn't fit my purpose. It was in this process of searching that I stumbled over the possibility to use binary trees to represent the syntax of a mathematical formula. Remembering what I learned in the lecture *Datastructures and Algorithms* I realized that binary trees enable me not only to represent, but also to manipulate formulas as needed.

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 had used available syntax trees libraries, but I relished the idea of implementing a tree structure that I would use myself. Finally, I would put to use what I have learned in that lecture. 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. I tried to keep the tree as simple as possible, giving only a value to the nodes without supplying a unique key. The greatest challenge given by implementing a syntax tree was the handling of a unary operator. Braces serve to determine the depth of a tree, wherein a binary operator tells us when to start climbing up again. However, the unary introspection operator required additional case handling.

The tree was not only important to the algorithm, but could also be used to check if the input was written correctly.

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 implementations of a syntax tree made to precisely fit my purpose.

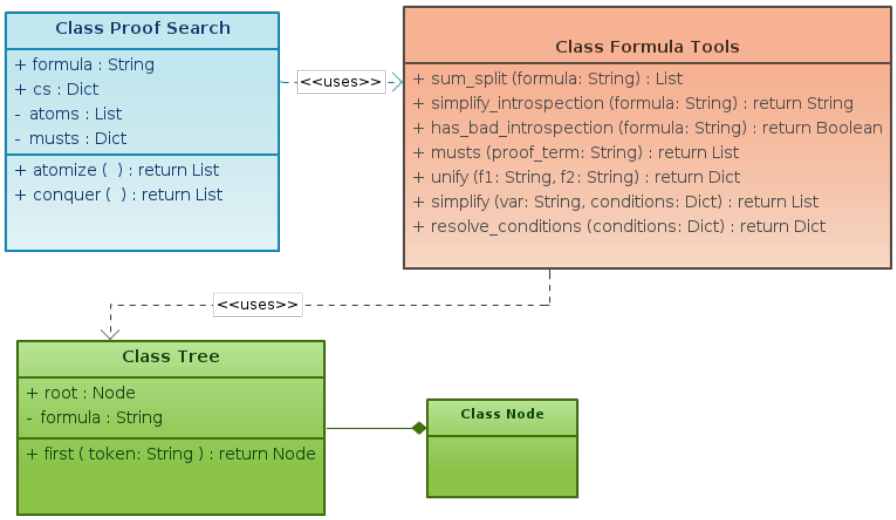

Figure 4.1: Simplified UML graphic of the classes used for implementing the algorithm. Additional helper methods and attributes are hidden.

`ProofSearch`

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

`FormulaTools`

The name of the module already reveals its usage. This module doesn't wholly deserve to be called a class since it does not describe any model but simply serves as a box of tools which perform functions not solely related to the model `Tree`

and `Node` nor to `ProofSearch`. It is something of an in-between. Removing this module would result in a lot of static methods where it would be unclear if they would best fit `Tree` or `ProofSearch`. Its core responsibility lies in modifying and analyzing formulas. This stands in contrast to the `Tree` class, which is only concerned with the single formula that it describes and the `ProofSearch` class, that does not actually handle formulas but only evaluates the result of an action on a formula and decides on how to proceed.

4.2 Selected Methods

In this section I want to show and explain some of the more important and thus more complicated methods that make up the heart of the algorithm. The source code of the methods presented here are excerpts only. Their in-line comments were shortened in order to keep the snippets as short as possible and to avoid unnecessary repetition. The aim of this section is to provide a insight into the source code without the need to read through all of it.

4.2.1 `atomize, ProofSearch`

The method `atomize` can be seen as the *divide* step of the algorithm. I was tempted to name it such because it would have fit 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 concept and does not fit this algorithm completely.

The method is a straightforward implementation of the algorithm in chapter 3.2. It splits the formula for each sum operator found. Then it tries to simplify all subformulas that start with an introspection. Subformulas that are not resolvable [1] are removed, leaving only those that we call *atoms*.

```
...
# 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
```

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

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

4.2.2 musts, FormulaTools

The method *musts* expects a given proof term to be atomized as it only distinguishes between introspection and application operators. The algorithm takes the formula apart from top to bottom, generating new, smaller terms for every operator until the remaining proof term is but one proof constant.

Since the resolution of an application operator introduces a new X variable and the resolution of a introspection operator 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`.

```
...
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 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 $!b : X_i$ follows $\exists X_j \; s.t. \; !b : (b : X_j)$, all X_i that occurred up until 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 spent most of my time implementing this method, or rather its many predecessors. It used to be a lot longer and more complicated because it differentiated various cases concerning X and Y variables. In this final implementation, X or Y variables are handled the same on this level.

The method `unify` takes two formulas [2] and compares their tree structure. If the roots of both trees are operators and matching, the subtrees of both nodes are pushed on a stack to be further compared later. If one of the trees being compared is only a leaf and does not match the root of the other tree we either have found a contradiction or a *condition*. In the first case the method will

[2] It is assumed that the only occurring operator are $\to$ and :. It would be easy to extend the code at this point.

return `None`. In the other case, the term of the second tree becomes a *condition* for the variable of the first tree.

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 a key. At the current stage conditions that apply to the same variable may contradict each other, but this method is responsible only for collecting conditions and not for evaluating them. This will done by the method `simplify` and 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 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 $FormulaTools$.

4.2.4 `simplify, FormulaTools`

The aim of this method is to replace all occurrences of the variable given as input except for its usage as key from the condition set. [3].

For example, if we have $(A \to B)$, (X_2), and $(Y_1 \to Y_2)$ as conditions for X_1, and (X_1) as a condition for X_2, the method returns $(A \to B)$ as the only condition for X_1, $(A \to B)$ and $(Y_1 \to Y_2)$ 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 than expected because I underestimated the significance of the newfound variables. The method `resolve_conditions` handles the order in which `simplify` is called on each variable. `resolve_conditions` pushes the newfound variables on top of the stack to make sure they are given 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 might be expected, it returns the list of newfound variables.

[3] In the source code the condition on the variable is referred to as *the chosen one*.

```
...
# 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 gives a readable output.

There are two main loops in `conquer_one_atom`. The first loop 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.

```
...
# 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 an overall configuration that is compatible with at least one of the configurations per *must*. If at one stage there is no entry left in `merged_conditions`, it means that the conditions posed by the newly encountered configuration of the musts are not compatible with the old ones and thus the atom is unprovable.

```
...
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 unit tests I have written for this algorithm have been most important to its success. They served me in two ways: First to check if my code would behave and actually do what I expected. Second, when I encountered an example where I did not know what to expect, I would simple write a test for it and see what happened. This helped me understand the problem. It is because of the tests that I have found so many mistakes, some minor, others major, making me re-code bits and parts again and again at a stage when I had hoped to be done with coding.

As can be seen when looking at the source code, not all methods are tested on a same level of quality. Methods that I deemed simple usually have only one or two tests. An example for that is `summarize` from `ProofSearch`. This method

simply rearranges the elements of a dictionary and returns a nice, readable output summarizing the content of the original input. In contrast to this, `__str__` from `Tree` which tests if the parsing of a formula works correctly contains lots of tests.

There are currently[4] a total of 66 tests, almost half of which are related to `ProofSearch`.

[4]Even though the program is finished I might add more tests to get rid of any doubts.

Chapter 5

Example

In this chapter I will walk through a example covering a few key cases. The justification term we will look at is rather complicated but the walk-through will also show how it can be broken down into simpler terms.

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 a simple list of pairs as given by the definition of justification logic. Also, there are more brackets explicitly written than required by convention to enable parsing.

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

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

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.

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

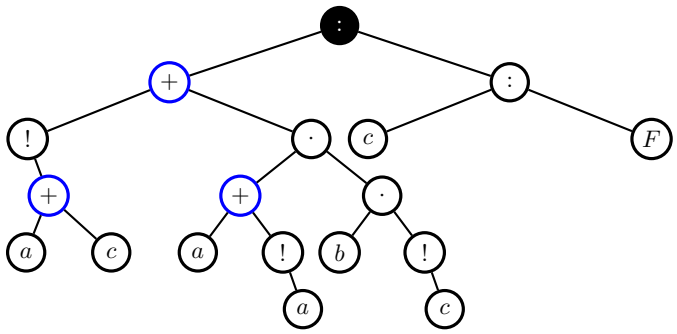

Figure 5.1: Syntax tree of given formula f before it is atomized.

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 with regard to their introspection operators.

5.2.2 Introspections

If the proof term starts with an introspection operator, there are two possibilities: First, the term could be simplified. Second, the term could be discarded. See the examples below for an illustration of those cases.

If we find an introspection operator as a child of a multiplication operator, there are again two possibilities: First, if we find the introspection operator to be the right child of the multiplication operator, the term is valid, meaning it contains no inherent contradictions. Second, if we find the introspection operator to be the left child of the multiplication operator, the term is invalid.

An invalid term is discarded.

Term (5.4)

The introspection operation for this justification term 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 (5.3)

In this term we find an introspection operation which when trying to simplify shows us that it cannot be resolved. Thus, it is no atom and we discard this term.

$$((!\,a) : (c : F)) \;\Rightarrow\; \lightning$$

Term (5.5)

In this term we find that the introspection operator is neither a left child of a multiplication operator nor a top operator of the proof term and thus we have our second *atom*.

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

Term (5.6)

This proof term has two introspection operators of which the first is the left child of a multiplication operator, thus making the term invalid. The second introspection would be valid, but the first term causes the whole proof term to be discarded.

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

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

We have found the two atoms a_1 and a_2 for the formula f. The next step determines the *musts*, matching them against the cs-list and finally merge the possible configurations together to determine if one of the musts is provable.

5.3.1 Musts

Atom a_1 (5.7)

Since a_1 consists of only one proof constant already, there is nothing further to here.

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

Atom a_2 (5.8)

For a_2 we need to take the proof term apart bit by bit.

$$\begin{aligned}((a \cdot (b \cdot (!c))) : (c : F)) \Rightarrow \\ a :\ & (X_1 \to (c : F)) \\ (b \cdot (!c)) :\ & X_1\end{aligned} \tag{5.10}$$

The proof constant a has been isolated. We repeat the step above to take apart $(b \cdot (!c))$, introducing yet another X variable.

$$\begin{aligned}(b \cdot (!c)) : X_1 \Rightarrow \\ b :\ & (X_2 \to X_1) \\ (!c) :\ & X_2\end{aligned} \tag{5.11}$$

Now b has been isolated as well, leaving $(!c)$ as the only unresolved proof constant. The introspection operator in this proof term results in a new X variable, which combined with the proof constant, replaces the previous X variable.

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

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

$$musts\ a_2 : \quad [(a, (X_1 \to (c : F))),\ (b, ((c : X_3) \to X_1)),\ (c, X_3)] \tag{5.12}$$

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

5.3.2 Using the CS-List

We now have to look up each *must* of an 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 more work. First we select and compare all *musts* of a_2 with the corresponding entries in the cs-list. We do this by searching for the entries of the proof constants, which double as keys to lists of terms in the cs-list. The value of the *must* is then matched to the entries of the list of terms. This gives us one or more sets of conditions per proof constant.

Proof Constant a

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

$$\begin{aligned}(H \to (c : F)) &\Rightarrow \{X_1 : H\} \\ ((E \to (c : D)) \to (c : F)) &\Rightarrow \{X_1 : (E \to (c : D))\}\end{aligned} \tag{5.13}$$

Proof Constant b

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

$$\begin{aligned}((c : F) \to H) &\Rightarrow \{X_1 : H,\ X_3 : F\} \\ ((c : D) \to (a : F)) &\Rightarrow \{X_1 : (a : F),\ X_3 : D\} \\ ((Y_1 \to (Y_2 \to Y_1)) &\Rightarrow \{X_1 : (Y_2 \to Y_1),\ Y_1 : (c : X_3)\}\end{aligned} \tag{5.14}$$

We note that for the last set of conditions we now have Y variables aside from the X variables 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 sets of condition.

$$\begin{aligned} (c:F) &\Rightarrow \{X_3 : (c:F)\} \\ G &\Rightarrow \{X_3 : G\} \\ D &\Rightarrow \{X_3 : D\} \\ (G \to F) &\Rightarrow \{X_3 : (G \to F)\} \end{aligned} \tag{5.15}$$

5.4 Constructing the Final Result

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

5.4.1 Merging Conditions

We put together a new set that contains one set of conditions per proof constant. Then we try to simplify this new set. If we encounter a contradiction in doing so, the new set is discarded. If the new set stays without contradictions, we have found a configuration. For example we could pick the first set from each, but this would give us a contradiction since X_3 can only be either F or $(c:F)$ but not both.

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

$$a \cap b : \{X_1 : (E \to (c:D)),\ X_1 : (Y_2 \to Y_1),\ Y_1 : (c:X_3)\} \tag{5.16}$$

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 sets of c we will resolve the current conditions if possible.

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

$$\{X_1 : (E \to (c:D)),\ X_3 : D,\ Y_1 : (c:D),\ Y_2 : E\} \tag{5.17}$$

As a consequence of merging the set of conditions (5.13) from a with set of conditions (5.14) from b there is no choice left for any of the X variables. The final result depends on finding a set of conditions of proof constant c that doesn't contradict the value for X_3. As it happens this is the case for the set of conditions in the second last row (5.15).

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 configurations of 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 about the Y variables. We know that $X_1 = (E \to (c : D))$ and $X_3 = D$. We replace the X variables in the *musts* with those values. This gives us:

$$\begin{aligned} a_2 : \ & [(a : ((E \to (c : D)) \to (c : F))), \\ & (b : ((c : D) \to (E \to (c : D)))), \\ & (c : D)] \end{aligned} \tag{5.18}$$

As we see these entries can all be found exactly like that in the cs-list (5.2). From those we can also reconstruct the term of a_2:

$$\Rightarrow (c : D) \tag{5.19}$$
$$\Rightarrow ((!c) : (c : D)) \tag{5.20}$$
$$\Rightarrow ((b \cdot (!c)) : (E \to (c : D))) \tag{5.21}$$
$$\Rightarrow ((a \cdot (b \cdot (!c))) : (c : F)) \tag{5.22}$$

And with (5.22) for a_2 we have again what we started with right after the atomization step in (5.5). In the graph below the path with the tree of the atom a_2 is highlighted.

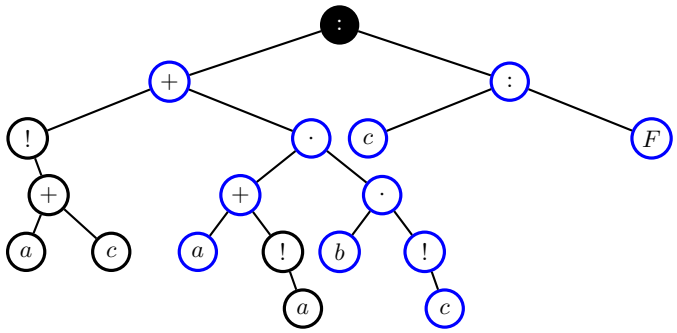

Figure 5.2: Syntax tree of formula f with atom a_2 hightlighted.

This concludes the example chapter.

Chapter 6

Conclusion

Looking back there are many things I could have done differently. For example, there is a python module called *Pyparser*[1]. It allows the creation and parsing of simple grammars. It would have served me to get rid of some of the hard-coded string comparison. An other point I often reflected on is the choice of language. There might have been more suitable programming languages for this task than Python, for example a functional language like Haskell.

On the other hand there are a lot of things that could be done to further improve this implementation. It would be advantageous for users to have a user interface. More importantly, it would be benficial if the algorithm didn't just handle implications but other common operators as well. A natural first step would be adding negation to achieve completeness.

A last point that should be considered is the possible optimization of the implementation of this algorithm. As mentioned before, no special care has been taken to consider the efficiency. Also the algorithm has not been tested for extremely long input formulas which might cause the implementation to break.

Bibliography

[1] Pyparsing. http://pyparsing.wikispaces.com/. Accessed: 2015-03-30.

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

[3] Sergei Artemov and Melvin Fitting. Justification logic. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Fall 2012 edition, 2012. http://plato.stanford.edu/archives/fall2012/entries/logic-justification/.

[4] 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. http://z3.codeplex.com/.

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

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