Justification Proof Search Implementation in Python

Bachelorarbeit

der Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

vorgelegt von

Judith Fuog

2014

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

Abstract

In short what's it all about.

Contents

Chapter 1

Introduction

1.1 Motivation

What's the motivation behind it? Not **MY** motivation, but the scientific motivation.

1.2 Goal

The initial goal was to extend an existing proof search engine Z3 [Microsoft Research] such that it could also handle Justification Logic. Deeper investigation into that project revealed that to make it handle also Justification Logic the given interface in Python would not work. Instead it would have to be look into the core of the programm which is written in C. The expenses it would require to get so much deeper into the material that the actual indented work would be only secondary. So instead of extending Microsoft Research project the actual goal changed to implementing a simplified proof search for Justification Logic. It meant that the implementation would be easier since it does not depend on anything else anymore. As a downside a lot of the functionallity that was hoped go get from Z3 would have to be implemented as well or left out.

The program should satisfy to following conditions:

Should this really be here in this chaper?

Input The formula to be proven as well as a list of formulas needed for the proof is given as string. It may be presumed that the string is exactly formatted in the way needed. It must not be checked for syntax error or general typing mistakes.

Output A simple *True* or *False* for the provability of the formula. [1]

1.3 Overview

The second chapter will present a short introduction to Justification Logic, but will go only as deep as needed to understand the problem as well as the develop algorithm.

[1] Optional the output could give information about how a proof was found if the formula is provable.

The heart of the third chapter will introduce the algorithm used in the program. Since this thesis concerns itself more with the practical side of implementation and not the theoretical side of mathematical logic theory there will be little proof here but instead many example to show how the algorithm works.

j-logic

Finally the last chapter will discuss the result of the work and give some ideas about how the work of a Justification Logic proof search implementation could be improved.

Chapter 2

Background

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

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

2.1 Justification Logic

The theory presented here is oriented mainly on the work of Goetschi [2005] as well as the older reference Paper and also from the homepage Stanfort. This definitions and rules given here are not complete to the justification logic. Priority was given to those informations which are vital for the implementation. So however briefly and incomplete the theory is presented here full reference can be found in the named sources.

2.1.1 Origins

Justification Logic has its origins from the field of modal logic. In model logic $\Box A$ means that A is *know* or that we have *proof* of A. In justification logic the equivalent would be $t : A$ where t is a proof term of A. So we have the notion that *knowledge* or *proofs* may come from different sources. Justification logic lets us connect different *proofs* with a few simple operators and thus describe better the proof. It may be said that where in model logic the knowledge is implicit it is explicit in Justification Logic[1].

2.1.2 Rules and Definitions

Definition. [2]

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

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

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

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

where i and j range over positive natural numbers, c_i^j denotes a (justification) constant of level j, and x_i denotes a (justification) variable.

2.2 Order of Operation Tree

Chapter 3

A Divide and Conquer Algorithm

3.1 Core Idea

To search a formula for is provability it had to be found a way which allows to do the same steps, now matter what form the formula actually has. A first attempt was to strictly use recursion. This method should have worked but it proved to be very difficult to implement, because there are so many different cases to consider in one recursion step. Also the stack created by this could become problematic for very large formulas.

Instead a *Divide and Conquer* approach is used. Diving will break even a large and complicated formula down to its most simple elements. Then these elements can be tested for their provability and in the conquer-step the results of the elements are put together giving the final result. Since the *Divide and Conquer* algorithm design pattern uses multi-branched recursion there still remains some recursion but as this takes place at a much deeper level the cases within a recursion are reduced as well as the size of the recursion stack.

3.2 Divide

The motivation behind the divide-step existed already long before the actual idea of the Divide and Conquer approach. Given a formula there would be no way to know what kind of formula it was or more precise: what operations were to be found within the *justification term*. The original goal was to find a way to restructure any given formula so that handling it would always need the same steps and not depend to much on what the formula looks exactly. I was looking for something like the CNF[1] and use it in a similar way as CNF is used in PSC[2]. As the Sum-Rule for *justification terms* works straight forward like a disjunction it is rather simple to restructure the formula as far as the Sum-Operator goes.

[1] *conjunctive normal form*, a conjunction of clauses, where a clause is a disjunction of literals

[2] *Proof Search Calculus* as it is introduced in Goossens et al. [1993]

add graph or formula

But since the Multiplication-Operator is not even symmetric operator it does not work like the conjunction know from the CNF and thus makes to restructuring of a formula all the more difficult. In addition there is the unary Bang-Operator which in itself is rather simple but still adds to the overall complexity.

In the end the restructuring would look like the following:

- For each Sum-Operator in the formula, split it in two formulas.
- If the first operation of a formula is a Bang-Operation, check if it can be simplified. If not, remove this formula.
- There are certain positions of a Bang-Operator within the formula that cause the whole formula to be false. Those formulas shall be eliminated as well.

Those three steps which are called *Atomize* in the source code break a given formula down to several simpler formulas which only contain the Multiplication-Operator as well as valid Bang-Operators. It is only then that a recursive method is called to analize the formula in a way that makes it possible to check if this subformula is provable.

This basically concludes the Divide-Part of this algorithm. The only thing left to done in the Conquer step is to check each of these formula. In the case of Justification Logic it means they need to be looked up in the *constant specification*.

3.2.1 Atomize

In this section *formula* usually refers only to the justification term of the formula and is used as a synonym. If it should be understood differently it will be stated so explicitly.

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

- *The term contains no Sum-Operations.*
- *A Bang-Operation can neither be the top operation of a term nor be the left operant of a Multiplication-Operation.*

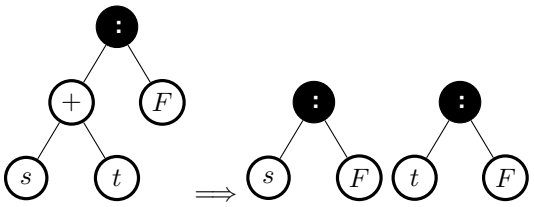
To make the content presented here more understandable the following example will illustrate the steps taken.[3]

Sumsplit

From the XX Rule of Justification Logic it follows that checking for provability in a formula where the top operation is a sum is equal to checking either operant of the sum and if any of it is provable so is the original formula.

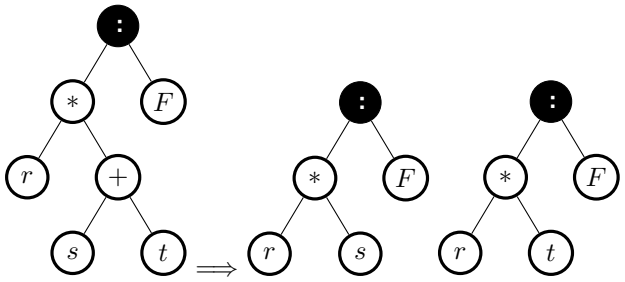
[3]It is on purpose that the *justification term* is by far more complicated than statement $b : F$ that follows the *justification term*. As far as this algorithm goes the complexity of the statement is of no further consequence and thus is kept as simple as possible to allow a easier overview.

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



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

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

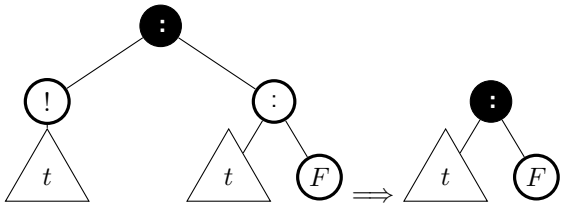


Simplify Bang

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

Derived from the XX Rule we get the following:

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



Speaking in the manner of a Syntax Tree it needs to be checked, if the child of the Bang-Operation is identical with the left child of the right child of the root. In that case the formula can be simplified to right child of the root only. Else there is no way to resolve the Bang-Operation.

Remove Bad Bang

This last step in atomizing the formula proved to be on of the hardest to realize. Only countless examples support the claim that the Bang-Operation must not be the direct left child of a Multiplication-Operation. In coming to that conclusion it has been helpful that no Sum-Operation could make the situation more complex. Because of this and also the fact that a Bang-Operation is never the top operation in a formula it is guarantied that a Bang-Operation must be either a right child or a left child of a Multiplication-Operation.

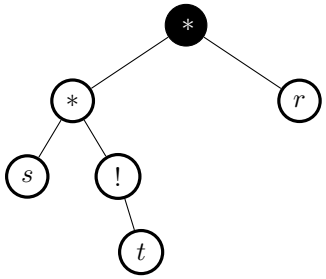
Operation? Operator?

Assertion 1 (Tree Version)**.** *A Bang-Operator that is the direct left child of a Multiplication-Operator causes the whole term to be invalid (unprovable), given that the term is without Sum-Operators and no Bang-Operator at the top.*

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

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

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



3.2.2 Get Must

3.3 Conquer

3.3.1 Configurations and Conditions

3.3.2 Merge

3.4 Implementation

Example.

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

Chapter 4

Results

4.1 Application

4.2 Enhancement

Bibliography

Remo Goetschi. On the realization and classification of justification logics. 2005.

Michel Goossens, Frank Mittelbach, and Alexander Samarin. *The LaTeX Companion*. Addison-Wesley, Reading, Massachusetts, 1993.

Microsoft Research. Z3, high performance theorem prover. URL `http://z3.codeplex.com/`.

Artis Paper.

Plato Stanfort.

Todo list