Proof Search for Justification Logic in Python

Judith Fuog

Universität Bern, Informatik

13. Nov 2014

Outline

Motivation

Combination of

- practical coding in a modern language
- contribute to something existing
- riddling around with logic

Early Stage

Extend an existing project to also handle Justification Logic.

Z3 Theorem prover from Microsoft Research. It can be used to check the satisfiability of logical formulas over one or more theories.
`http://z3.codeplex.com/`

Available APIs

- Python
- C, C++, .NET, Java

Later

Implement a stand-alone algorithm for proof search in Justification Logic.

Input:

- Correct formatted String of a formula with a *constant specification* list.
- Formula may only contain Justification Logic operations and implication ($\rightarrow$).

Output:

- If formula is provable.
  (True or False)

No time restriction or special care for efficiency.

Technologies

Python Although the idea of extending *Z3* was dropped the choice of the language remained.

- version 3.x
- unit tests: module `unittest`

pyCharm IDE for Python

git for versioning

Methods

KISS Simplify as much as possibly and make it run. Add more functionality later on.

Tests Lots of tests

- Tests in advance (TDD)
- Tests for doubts
- Tests while debugging

Result

Basic requirement fulfilled including one addition.

Code

- 4 classes, thereof `Tree` and `ProofSearch` most important.
- Test for all 4 classes. Number of tests vary greatly depending on the complexity of the function.

What is Justification Logic?

Justifications logic are epistemic model logic that use a formal construct to formalize the justification of knowledge of a statement.

Formula **t:F**

- t proof term t is a justification for F.
- F Axiom F must satisfies conditions t.

Basics of Justification Logic

Rules for proof terms

C1 $t : (F \to G), s : F \vdash t * s : G$

C2 $t : F \vdash (t + s) : F$, $s : F \vdash (t + s) : F$

C3 $t : F \vdash !t : t : F$

constant specification **CS**

Finite set of formulas of the form $(c : A)$ where c is a proof constant and A is one of the axioms **A0-A4**.

Example

Find if the following formula is provable for CS:

- $s : G$
- $((s * t) + (!s)) : F$
- ...

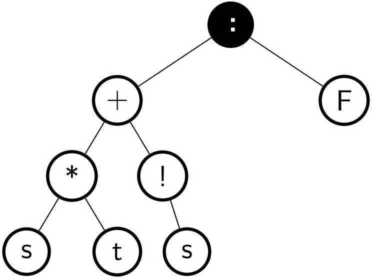
$$CS = \{(s, (t : F)), (s, (G \rightarrow F)), (s, G), (t, (s : G)), (t, G)\} \quad (1)$$

If the proof term is not constant, it must be broken down in pieces.

Binary Tree

Find order of operations

- Regular Expressions?
- Binary Trees!



Types

Often the data types are mixed such that it becomes very difficult to keep an overview.

- **String** Formulas are always passed as String, even if they change their type in between.
- **Dictionary** Pythons implementation of a Hash.
- **Tuple** are immutable in Python. Used for conditions.
- **List** are used to represent tables among other uses.

Example:

$[(\{'X1' :'','X2' :'C','X3' :'B'\},[('X2','X1'),...]),...]$

The Divide and Conquer Principle

Problem

Straight-forward approach proved to hold to many cases to handle them all at once.

divide
: Take a formula apart till only the smallest parts remains.
  - atomize (sumspilt, simplify bang, remove bad bang)
  - get musts

conquer
: For each of these *atoms* check if they are resolvable in CS.
  - configurations and conditions
  - merge

Atomize - Sumsplit

C2 $t : F \vdash (t + s) : F$, $s : F \vdash (t + s) : F$

To check if provable for

$$((s * t) + (!s)) : F$$

it is enough if (any) one of the parts is provable.

$$(s * t) : F, (!s) : F$$

Atomize - Simplify Bang

C3 $t : F \vdash !t : (t : F)$

If a bang is first operation of the left subtree the left child of the right subtree must be the same as what is beneath the bang from the left subtree. So it is enough to simple check the right subtree.

$$(!s) : (s : G) \Rightarrow s : G$$

$$(!s) : F \;\Rightarrow \bot$$

Remove Bad Bang

C1 $t : (F \to G), s : F \vdash t * s : G$

C3 $t : F \vdash !t : (t : F)$

Whenever a bang is a left child of a multiplication the formula cannot be resolved into proof constants.

$$((!a) * b) : H$$

$$\Rightarrow \exists X_1 : (!a : X_1 \to H), (b : H)$$

$$\Rightarrow \exists X_2 : (X_1 \to H) = (a : X_2) \Rightarrow \perp$$

Musts

A list of all proof constant with corresponding formula for a atomized formula that must be looked up in CS. If the formula contains any multiplication, there will be so-called *Wilds*.

Example

$$(s * t) : H$$
$$\Rightarrow \{s : X_1 \rightarrow H, t : X_1\}$$

Find in CS

For each entry in a list of a *must*, see if there is a possible match in CS. For those atomized formuals that contain *Wilds*, there can be more than one possibility.
One *must* will return a configuration table.

Example

Search $a : X_1 \rightarrow F$ in
$CS = \{(a, (b : B) \rightarrow F), (a, H), (a, A \rightarrow F)\}$

$\Rightarrow X_1 \in \{A, (b : B)\}$

… with conditions

CS can contain special entries, such as $Y_1 \to (Y_2 \to Y_1)$ where Y_i can be any formula. As a consequence a configuration may also have a condition.

Example

compare $X_1 \to (X_2 \to (b : X_1))$ with CS entry $Y_1 \to (Y_2 \to Y_1)$

$\Rightarrow$
X_1, X_2 beliebig
$X_3 = b : X_1$

Merge

For an *atomic* formula to be satisfiable there must be a match for each *must*. So the different configurations of the musts of one formula must be *merged*.

Example

s: $X_1 \in \{A, A \to B, b : B\}$, $X_2 \in \{H, G\}$

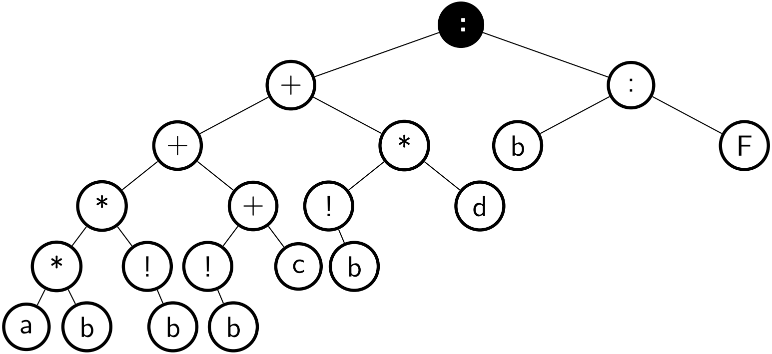
t: $X_1 \in \{b : B, C\}$, $X_2 \in \{G\}$

$\Rightarrow$

$X_1 = b : B, X_2 = G$

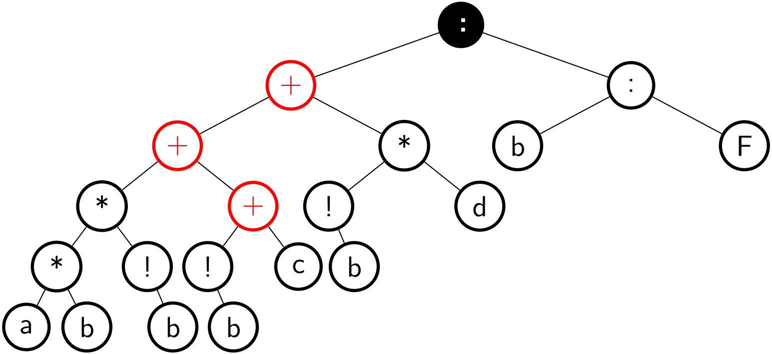
Formula

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

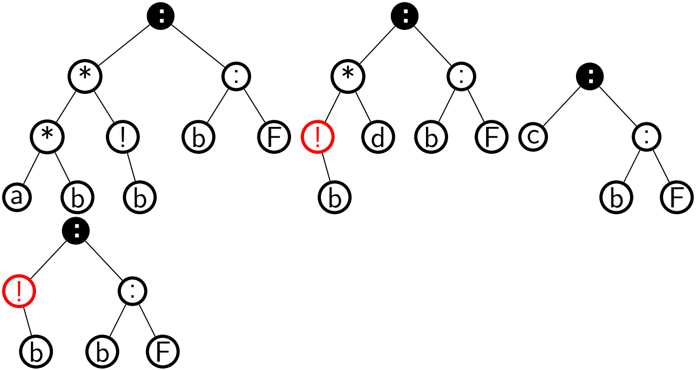


Sumsplit

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

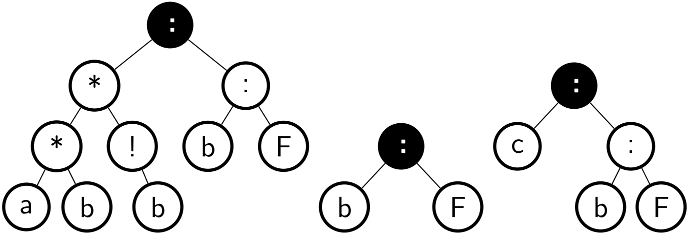


Bangs



$$((a*c)*(!b)) : (b : F), (\,!\,b) : (b : F), c : (b : F), ((\,!\,b)*d) : (b : F)$$

Atomized



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

Musts

I $((a * b) * (!b)) : (b : F)$
  i $a : X_3 \rightarrow ((b : X_2) \rightarrow (b : F))$
  ii $b : X_2$
  iii $b : X_3$

II $b : F$
  - $b : F$

III $c : (b : F)$
  - $c : (b : F)$

If *II* or *III* are found in CS, then the formula is satisfiable.
For *I* we need to make a configuration table.

Configs

$$CS = \{(a, G \to ((b : B) \to (b : F))), (a, Y_1 \to (Y_2 \to Y_1)),$$
$$(b, b : F), (b, G)\}$$

- I $((a * c) * (!b)) : (b : F)$
  - i $a : X_3 \to ((b : X_2) \to (b : X_2) \to (b : F))$
  - ii $b : X_2$
  - iii $b : X_3$

	X_2	X_3
i	B	G b:F[1]
ii	b:F G	
iii		b:F G

Table 1: Configs for I

[1] from condition: $(X_3, b : F)$

Merge

	X_2	X_3
i	B	G
		b:F
ii	b:F	
	G	
iii		b:F
		G

Table 2: Before merge

X_2	X_3
b:F	b:F
b:F	G

Table 3: After merge

Since we found at least one valid configuration, the formula is provable with the given CS.

How to get started

The theorie(s) behind it

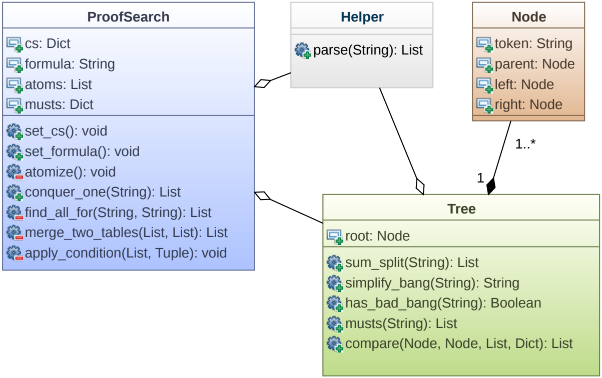
- How much of what do I need to know?
- How well do I have to understand it to be able to implement it correctly?

Class Design when implementing an Algorithm

Expectation: Python modern language, object-oriented approach for implementation.
Result: Not very O-O style.

- High cohesion very difficult to achieve
- Tight coupling almost inevitable

UML


ProofSearch
cs: Dict
formula: String
atoms: List
musts: Dict
set_cs(): void
set_formula(): void
atomize(): void
conquer_one(String): List
find_all_for(String, String): List
merge_two_tables(List, List): List
apply_condition(List, Tuple): void
Helper
parse(String): List
Node
token: String
parent: Node
left: Node
right: Node
1..*
1
Tree
root: Node
sum_split(String): List
simplify_bang(String): String
has_bad_bang(String): Boolean
musts(String): List
compare(Node, Node, List, Dict): List

Those were some of the problems that took the longest for me to figure out.

- **musts** How can a formula be broken down such that I know what I must be looking for in *CS*.
- **compare** How to compare two trees and what to do with the result (*Wilds!*).
- **Y-Wilds** What difference do they make when comparing *musts* proof constants with *CS* entries. Are all cases covered?

What must still be done?

- Decide for a standard output.
  (Only True/False, table of must for which a configuration was found or all tables of musts for which a configuration was found.)
- A few in-code documentation additions.
- Final refactoring.

What could be done?

- Add more logic operators.
  (such as $\vee, \wedge, \neg$)
- User-friendly GUI (simple) with input check.
- Clearly state assumptions and proof them.

Personal Conclusion

- new language
- riddling with logic
- implementing an algorithm
- clean coding (documentation and presentation)

Question?