Knowledge Representation & Reasoning: LK Calculus Reasoner

Luke Slater (luke.slater.1@kaust.edu.sa)

October 1, 2015

1 Implementation

The basic LK system has been implemented with all logical and structural rules, discluding the cut rule and those involving quantifiers. It is implemented in Javascript. In retrospect, I should not have implemented an undecidable problem in Javascript. Javascript is not an appropriate language for logical argument deduction.

The algorithm uses a breadth-first search to incrementally consider all possible expansion options until reaching the axiom ($A \vdash A$) for all subformulas in the given track. I use subformula to mean each of the formulae which must be solved to create a proof for the initial formula (so initially there will be one, and more will be created in the case that a rule which creates two formulas e.g. IL is run).

1. Create a set of tracks, and add the initial formula.
2. Iterate all tracks
   (a) Check if all subformulas are axioms. If so, print proof and exit.
   (b) Iterate subformulas and apply all rules.
   (c) Create a new track for each applicable rule (including a track for each possible combination of subformula result).
3. Replace the set of tracks with the new tracks generated by the current round.

The implementation operates on lists of symbols, which are modelled internally by arrays - one for each side of the sequent. Operations are represented in object notation - and these representations are also required as the input format due to the difficulty of parsing prefix or infix logical notation in Javascript. These representations are, however, converted to infix notation for the output stage.

2 Hilbert Calculus Proofs

Axiom 1 Proof

```
1 Step 0
2 Formula 0
3   Operation: IN
4   Value:  ⊢ A ⇒ A
5 Step 1
6 Formula 0
7   Operation: IR
8   Value: A ⊢ A
```

Axiom 2 Proof

```
1  Step 0
2  Formula 0
3    Operation: IN
4    Value:  ⊢ A ⇒ (B ⇒ A)
5  Step 1
6  Formula 0
7    Operation: IR
8    Value: A ⊢ B ⇒ A
9  Step 2
10 Formula 0
11   Operation: IR
12   Value: A, B ⊢ A
13 Step 3
14 Formula 0
15   Operation: WL
16   Value: A ⊢ A
```

Axiom 4 Proof

```
1  Step 0
2  Formula 0
3    Operation: IN
4    Value:  ⊢ ((¬A) ⇒ (¬B)) ⇒ (B ⇒ A)
5  Step 1
6  Formula 0
7    Operation: IR
8    Value: (¬A) ⇒ (¬B) ⊢ B ⇒ A
9  Step 2
10 Formula 0
11   Operation: IR
12   Value: (¬A) ⇒ (¬B), B ⊢ A
13 Step 3
14 Formula 0
15   Operation: PL
16   Value: B, (¬A) ⇒ (¬B) ⊢ A
17 Step 4
18 Formula 0
19   Operation: IL
20   Value:  ⊢ ¬A, A
21 Formula 1
22   Operation: IL
23   Value: B, ¬B ⊢
24 Step 5
25 Formula 0
26   Operation: NR
27   Value: A ⊢ A
28 Formula 1
29   Operation: NL
30   Value: B ⊢ B
```

Axiom 4m Proof

```
1  Step 0
2  Formula 0
3    Operation: IN
```

```
4     Value:  ⊢ (A ⇒ B) ⇒ ((A ⇒ (¬B)) ⇒ (¬A))
5   Step 1
6   Formula 0
7     Operation: IR
8     Value: A ⇒ B ⊢ (A ⇒ (¬B)) ⇒ (¬A)
9   Step 2
10  Formula 0
11    Operation: CR
12    Value: A ⇒ B ⊢ (A ⇒ (¬B)) ⇒ (¬A), (A ⇒ (¬B)) ⇒ (¬A)
13  Step 3
14  Formula 0
15    Operation: IL
16    Value:  ⊢ A, (A ⇒ (¬B)) ⇒ (¬A)
17  Formula 1
18    Operation: IL
19    Value: B ⊢ (A ⇒ (¬B)) ⇒ (¬A)
20  Step 4
21  Formula 0
22    Operation: PR
23    Value:  ⊢ (A ⇒ (¬B)) ⇒ (¬A), A
24  Formula 1
25    Operation: IR
26    Value: B, A ⇒ (¬B) ⊢ ¬A
27  Step 5
28  Formula 0
29    Operation: IR
30    Value: A ⇒ (¬B) ⊢ ¬A, A
31  Formula 1
32    Operation: IL
33    Value:  ⊢ A, ¬A
34  Formula 2
35    Operation: IL
36    Value: B, ¬B ⊢
37  Step 6
38  Formula 0
39    Operation: WL
40    Value:  ⊢ ¬A, A
41  Formula 1
42    Operation: PR
43    Value:  ⊢ ¬A, A
44  Formula 2
45    Operation: CR
46    Value: B, ¬B ⊢
47  Step 7
48  Formula 0
49    Operation: NR
50    Value: A ⊢ A
51  Formula 1
52    Operation: NR
53    Value: A ⊢ A
54  Formula 2
55    Operation: NL
56    Value: B ⊢ B
```

3 Semantic Entailment

4 Modus Ponens

Modus Ponens Proof

```
1  Step 0
2  Formula 0
3    Operation: IN
4    Value: A ⇒ B, A ⊢ B
5  Step 1
6  Formula 0
7    Operation: PL
8    Value: A, A ⇒ B ⊢ B
9  Step 2
10 Formula 0
11   Operation: IL
12   Value: A ⊢ A
13 Formula 1
14   Operation: IL
15   Value: B ⊢ B
```

5 Time Complexity

The implementation is undecidable, and so it is very difficult to calculate the time complexity - there is no lower bound. The best one can say is that if there is a proof to be found, then given unlimited time and unlimited resources, it will eventually be found.

The growth of the number of tracks which must be evaluated at every step of the breadth first search increases exponentially, but the degree of growth is highly dependent on the input.

The reason for this is that the application of certain rules leads to a massive number of new potential tracks compared to others. These particularly explosive rules are those which lead to multiple formulas e.g. IL. They lead to many new tracks initially because one must create a new track for every possible manner of distributing the remaining symbols in the formula between the two resulting formulas, and in subsequent steps because one must create new tracks for every combination of every possible rule applicable to every subformula in a given track repeatedly.

This issue is particularly apparent while trying to prove the Hilbert calculus axioms, which rely on nested implications - leading to several applications of the IL rule.

6 Performance Optimisation

It seems that there are many potential performance enhancements which could be applied to attempt to improve the performance of the calculus prover.

6.1 Implemented Improvements

In the methods I have attempted, the focus has been on reducing the number of track growth between steps of the search:

- Remove dead tracks: these are tracks with subformulas without any possible solutions, such as:
  - $\vdash$ A
  - Empty set both sides of the sequent
  - Total of one symbol on both sides of the sequent
- Limitation of structural rules; many unhelpful tracks were being generated by the constant application of structural rules, so these were limited. The following cases were disallowed:
  - The last rule was CR and the current rule is WR
  - The last rule was CR and the current rule is CR
  - The last rule was PR and the current rule is CR
  - The last rule was PR and the current rule is WR
  - The last rule was CL and the current rule is WL
  - The last rule was CL and the current rule is CL
  - The last rule was PL and the current rule is CL
  - The last rule was PL and the current rule is WL

I think that there is potential that some of these rules could prevent the discovery of proofs in some cases - particularly the limitation of repeated structural rules in the case of more complicated formulas. However, I did not come accross any such problems during testing.

6.2 Structural Rules

Given more time to rewrite the system to be a bit more flexible, I think that a major improvement could be to disallow the use of structural rules entirely, until it has been found that there is no possible solution to be found by the application of logical rules only. It should be possible to traverse all possibilities, since the LK system without structural rules should terminate.

The main difficulty here is to implement effective backtracking, figuring out where to start applying structural rules in the case of a dead path. It might seem like the better option to start applying structural rules from the last non-dead step, but this will cause problems. Take the example of the Law of Excluded Middle ($\vdash A \vee \neg A$), the proof for which requires an application of the CR rule as the first step - it is possible to apply a maximum of two steps before exhausting logical rule possibilities (in the case we use OR2 and NR).

Thus, a solution would have to attempt to find a solution using only logical rules, and upon failure apply structural rules on the first step, potentially leading to many more sub-problems than is actually necessary.

I think the superior solution would be to implement a modified version of the LK system which operates on sets rather than lists. This works because we know that the contraction and permutation rules mean the order and number of symbols in the sequence don't matter, and cannot prevent a valid proof being found.

Furthermore, many implementations of LK calculus also use 'weakening,' which allows the addition of arbitrary elements to the set of formulae. This too allows us to assume a proof (in which a formula is considered equivalent to $A \vdash A$) without the use of structural rules. This also leads to shorter proofs.

6.3 Existing Performance Issues

While track growth can be limited by the modification or elimination of structural rules, it does not eliminate the problem. The major performance issues with the presented implementation are twofold, and both stem from the exponential growth of the number of tracks which need to be processed.

The first is that a situation quickly arises in which there are too many tracks to process. Some of the methods noted above attempt to reduce the amount of unnecessary tracks generated, but it is still severely limiting. The current implementation is single-threaded, and an alternative solution in Groovy or similar would implement multi-threading to allow tracks to be evaluated by all cores.

Another issue which arises from the track growth in combination with the data model used, is that the RAM quickly grows to become out of control. The primary reason for this is that each track stores its entire history from step zero - when there are lots of tracks, this means a substantial amount of unnecessarily duplicated memory. It was implemented this way originally because it makes printing the eventual proof easy; in an improved solution, track histories would be stored only once, and each current track would just have a pointer to its corresponding history.