Proving Satisfiability in Chopsticks, the Finger Game

Progress Report

Ella Holl, James (Nuocheng) Wang, Luciano Székely Corsi, Ashna Srivastava

04.14.2021

Logic & Computation

Jason Hemann

Link to Software Repository:

https://github.com/ellaholl/LCProject

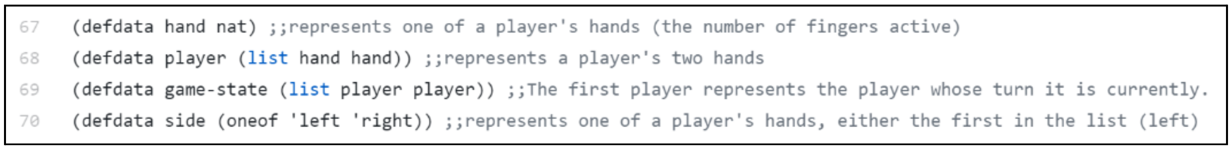
Introduction

Chopsticks is a turn based game played with 2 or more players (we will assume only 2) where each player starts with 1 finger on each hand. On your turn, you can tap one of your opponent's hands with either your left or right hand. Your opponent will then have to raise an extra number fingers on their tapped hand equal to however many fingers you have extended on the hand you tapped them with. If this number causes them to exceed or meet 5 fingers, then they have to lower the tapped hand, and it is out of play. Alternatively, on your turn you can choose to transfer fingers from one of your hands to the other. For example, if you have two fingers on your right hand and two on your left, you could transfer one finger from your left hand to your right, ending up with one finger on your left hand and three on your right. You can similarly use this type of move to put one of your hands back in play, by assuming it has zero fingers and transferring some from your other hand, or to take a hand out of play, by transferring all the fingers from one hand to the other. Obviously, you cannot make a non-move (swapping 0 fingers over, or changing your hand setup from 3 and 2 to 2 and 3), as this would cause the game to potentially continue infinitely. The game ends when one player has both their hands taken out of play; the last player to still have fingers remaining in play, wins.

Requiring nothing but two hands, an opponent, and basic addition and subtraction skills, the game "Chopsticks" has been a simple yet wildly popular tactical game for children. Focusing on the satisfiability of this game, this report turns this game into a boolean satisfiability problem by taking advantage of the fact that the number of fingers on each involved hand at any given game state can be represented as natural numbers. The progress made so far involves defining the variables controlling the game state, such as the number of fingers and enforcing rules on their behavior as a single move is taken. The final goal of this project is to prove satisfiability for various scenarios of the game, given some starting point A and X number of moves.

Walkthrough (What We've Done)

To set up the variables of the game, multiple structures were defined. For each hand, the number of fingers held up were simplified into a natural number between 0 and 5. To represent the 2 hands each player has, a player is represented by a list of two natural numbers. For these lists, it was agreed upon by the group that the first element represents the left hand and the second element represents the second hand. With this agreement, a structure of side could be utilized with “left” and “right” symbols to narrow down when to take the car and cdr of the hand list. Now that hands are grouped into each player, the main variable can be identified as the game state. This game state consists of the two player’s list of hands which allows us to navigate to each player and their specific hands using car, cdr commands. In addition, the side can be paired as a condition to ensure which number of fingers need to be checked.



```
(defdata hand nat) ;;represents one of a player's hands (the number of fingers active)
(defdata player (list hand hand)) ;;represents a player's two hands
(defdata game-state (list player player)) ;;The first player represents the player whose turn it is currently.
(defdata side (oneof 'left 'right)) ;;represents one of a player's hands, either the first in the list (left)
```

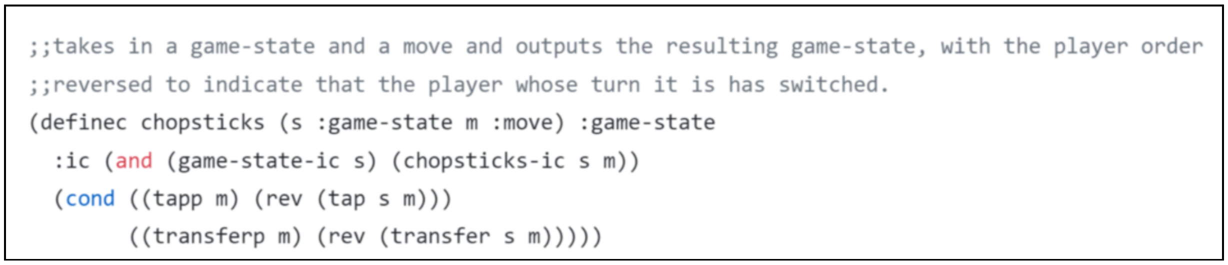
Figure 1: Defining Players Hands

The current code simulates the behaviour of playing the chopsticks game by one move. This means that the main function takes in a game state and moves to return the updated game state. In one variable, game state, a point in the game is defined by the number of fingers on each player’s hand and defined in a structure which helps us access each specific hand. Meanwhile, the inputs which define the move are simplified down to two functions which vary - tap and transfer.

The behavior of a tap from one player’s hand to their opponent’s is represented by a function that takes in two sides - the hand the player is tapping with and the hand the player is tapping to. When entering this function, there is an input contract to ensure that the number of hand tapping or hand being tapped has zero fingers. This is because a hand with zero is considered out of the game and cannot be revived by a tap from the opponent. If the inputs pass this contract, a tap updates the game state by adding the number of fingers from the player's hand to the opponent's hand that they tap.

The behavior of a transfer from one of a player’s hand to their other is represented by a function that takes in the number of fingers they are transferring and the side that they are transferring from. Once again, an input contract is utilized to ensure that the player is not transferring fingers they do not have or eliminating their hand by reaching or exceeding five fingers. If the inputs pass, a new game state will be returned where the input number is subtracted from the players given side and added to the players other side.

When considering how this function will be called recursively to prove satisfiability, it was decided that the location of player 1 and player 2 in the game state will flip each time they go through the step function. This is important because the tap and transfer functions themselves do not input an option on which player is taking on the move. Instead, it is more efficient to implement the behaviour of two players going back and forth. So, when the move function is called, the player position as the first element in the game state list will be carrying out the given move. This behaviour was implemented by calling the built in reverse function and pulling together all the code into a final method, chopsticks, displayed in figure 2 below.



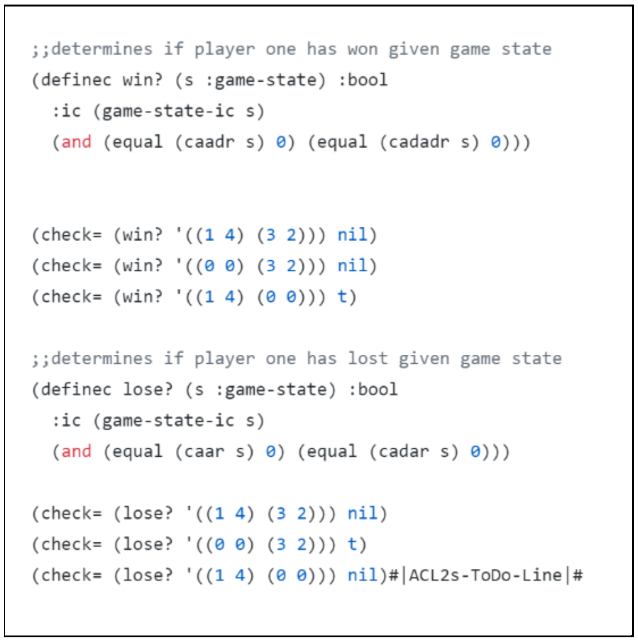
```
;;takes in a game-state and a move and outputs the resulting game-state, with the player order
;;reversed to indicate that the player whose turn it is has switched.
(definec chopsticks (s :game-state m :move) :game-state
  :ic (and (game-state-ic s) (chopsticks-ic s m))
  (cond ((tapp m) (rev (tap s m)))
        ((transferp m) (rev (transfer s m)))))
```

Figure 2: Chopsticks Function

Although no proofs have been completed yet, the group has designed the organization of this chopstick function carefully so that it can be used to summarize each step when proving satisfiability. As each player's number of fingers up increases, the number of possible moves each player can make increases exponentially. Our plan to manage this number of moves is to accumulate all options into lists. Starting with a list with one list of possible moves, the chopstick function can work through implementing each move and then adding additional lists for the next moves that keep the player in the game.

Remaining Work (What We're Prepared to do Next)

With the game that we have set up on ACL2S, our group's final goal is to answer the question "Is it possible at all to win from game state A in under X moves?" This problem will give us the freedom to explore multiple starting points and utilize the step behavior we have defined into a recursive loop.



```
;;determines if player one has won given game state
(definec win? (s :game-state) :bool
  :ic (game-state-ic s)
  (and (equal (caadr s) 0) (equal (cadadr s) 0)))


(check= (win? '((1 4) (3 2))) nil)
(check= (win? '((0 0) (3 2))) nil)
(check= (win? '((1 4) (0 0))) t)

;;determines if player one has lost given game state
(definec lose? (s :game-state) :bool
  :ic (game-state-ic s)
  (and (equal (caar s) 0) (equal (cadar s) 0)))

(check= (lose? '((1 4) (3 2))) nil)
(check= (lose? '((0 0) (3 2))) t)
(check= (lose? '((1 4) (0 0))) nil)#|ACL2s-ToDo-Line|#
```

Figure 3: Determining Outcome

So far, the boolean we return takes in a game state and can determine if player one has won or lost. To expand on this, we want to combine our chopstick function in figure 3 and repeat it until one of the two functions in figure 3 returns true.

When it came to brainstorming which question we would like to answer, we developed many options as shown below.

Question Options:

1. Can we get from A to B in under X moves?
2. Can we force a win starting at A in under X moves?
3. Is it possible at all to win starting at A in under X moves?
4. Can we force a result of position B from position A in under X moves?

Comparing the “style” of each question, we settled with the third because of how we must prove one possible winning method instead of a guaranteed method against artificial intelligence. Therefore, we want to prove that winning is “possible” instead of guaranteed for satisfiability.

Meanwhile, each question is given a constraint of “start at A in under X moves” to give us the freedom to test however many moves are within reason. In addition, there are situations where a win cannot be forced; such as, if both players continue to simply switch fingers for each move instead of tapping.

Noting that we have represented the game programmatically in ACL2S so far, our end goal is to develop a fully functional QBF solver that will be able to determine boolean satisfiability of the “Chopsticks” game. We can declare multiple milestones that we will pass as we successfully work towards our goal, which will serve as our success metric to measure our progress by.

Milestones:

1. Reducing the “Chopsticks” game into a CNF formula, ex. (… ^ (a v b v c) ^ (d v e v f) ^ …)
    a. Combining step function (Figure 2) which updates the game state with a move and boolean functions which determine an outcome (Figure 3). This will determine the boolean satisfiability outcome of an updated game state, which determines if the game has been won by player one.
2. Define function to determine winning state after one move
3. Implement a strategy to accumulate all possible next moves and identify the winning sequence of next moves.
4. Test applying the accumulating strategy to 2 moves and slowly increase the number of moves until the said strategy works with a sufficiently large number of consecutive moves, thus successfully proving that it is in fact possible for player one to win from starting point A in under X moves.

Conclusion (Current Concerns/Issues)

With each step of the game defined through the Chopstick function in Figure 2 and the outcome functions in figure three, our question of “is it possible for player one to win from starting point A in under X moves?” can be explored with multiple cases. Current concerns for this project regards how many moves we will be able to account for given how the options to consider for each move will increase exponentially. A runtime error will be reached due to how many possible moves a list variable will have to accumulate as the number of X moves increases. Therefore, we plan to start small with the number of steps that we can prove an outcome with and then test how well we can accumulate the possible moves on a large scale till a possible limit is reached. This is significant and consequential because in the worst case scenario, the game will run indefinitely because both players could make the same moves over and over again. For example, if a player had 2 fingers on each hand and combined them into 4 fingers on one hand and split them back into 2 fingers on each hand over and over again, that would still be legal because there would be no rule preventing that. In the real world, when this scenario happens in the game, which player wins simply depends on whose patience runs out first. Thus, it may be potentially beneficial to define a new rule in which we restrict our version of "Chopsticks" to prevent cycles from occurring; "a player cannot make an infinitely repeating sequence of moves".

The function we use to test the cycles of game-states that represent a match should include a checker that takes in a list of game-states (the sequence used to get to the current point) and checks whether there exist two adjacent sublists at the very end of the list that are the same. In such a case, it would output "false"; i.e., that sequence of moves does not satisfy the question, as it is invalid.

Given the current progress, and our milestones for the coming week, it seems that we have been able to and will continue to reasonably handle determining boolean satisfiability for the game of "Chopsticks".