Detecting Existence of Asymmetric Boolean Paths

by Sven Nilsen, 2017

In this paper I represent an efficient algorithm that detects whether a boolean function has an asymmetric path (cross argument not supported). I also discuss an interpretation of a binary encoding for detecting repeating patterns in binary sequences, and a natural bias that can be assigned to shorter programs aka Occam's Razor when picking random boolean functions.

A working implementation is located under "rust_experiments/boolean_paths".

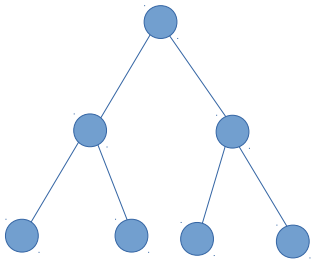
All boolean functions can be constructed using a higher order `if`:

```
if := \(a, b) = \(c) = if c { a } else { b }
```

For example:

```
if(false, false) <=> false_1
if(false, true) <=> not
if(true, false) <=> id
if(true, true) <=> true_1
if(id, false_1) <=> and
if(true_1, id) <=> or
```

The algorithm, that detects existence of asymmetric boolean paths, takes a binary tree description of the function using `if` for branches and `false/true` as leafs.



In normal programming it is common to shorten down the branches with early returns from a function. Here I use a complete tree, because it has the property that each layer of depth corresponds to the processing of an input boolean value.

It also proves that a binary function is determined only by the leaves of the tree, since the functionality of each branch depends on its sub nodes, which in turn depends on the leaves.

An asymmetric path is a list of functions that maps each input argument and the return value to a new boolean value:

$$f[g_0, g_1, \ldots, g_n] <=> \{h_0, h_1, \ldots, h_n\}$$

This corresponds to 3 possibilities:

- Non-existent path (empty function set)
- Total path (function set with one element)
- Partial function (function set with more than one element

For boolean functions, when cross argument paths are not supported, the only valid functions of `g` are the ones of the type `bool → bool`:

```
false_1
not
id
true_1
```

Two of these functions collapses the argument, and two preserves structure:

{false_1, true_1}	Collapses the argument
{not, id}	Preserves structure

When a parent node has identical children, its argument is irrelevant, because it does not matter whether you pass it `false` or `true` since the function has the same behavior anyway.

Collapsing functions of type `bool → bool` for `g` do the same as asking whether the argument is irrelevant for the function, or it creates a “tension” that must be resolved by collapsing later arguments.

Therefore, as the Rust code on the next page demonstrates, the algorithm to detect the existence of a path has very low complexity, because it only need to keep track of the tension and detect whether it has been solved for child nodes.

```
pub fn has_asympath(&self, ps: &[Expr], tension: bool) -> bool {
    // When both sub functions are the same,
    // the argument is irrelevant, so it does not matter
    // what the path does.
    if self.is_irrelevant() {
        if let Expr::If(ref a, _) = *self {
            // The tension is inherited.
            return a.has_asympath(&ps[1..], tension)
        } else {
            unreachable!()
        }
    }

    if ps[0].collapses() {
        // Collapse argument.
        match *self {
            // Last value, tension is resolved by collapsing.
            Expr::False | Expr::True => true,
            Expr::If(ref a, ref b) => {
                a.has_asympath(&ps[1..], true) &&
                b.has_asympath(&ps[1..], true)
            }
        }
    } else if ps[0].preserves_structure() {
        // Preserve structure.
        match *self {
            // Last value, path exists if there is no tension.
            Expr::False | Expr::True => !tension,
            Expr::If(ref a, ref b) => {
                a.has_asympath(&ps[1..], tension) &&
                b.has_asympath(&ps[1..], tension)
            }
        }
    } else {
        false
    }
}
```

A simple binary encoding of this algorithm can use a `$2^N$` sequence of bits to store values of the leaves, such that it corresponds to a truth table.

arg_0	arg_1	and	or
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	1

0001 and
0111 or

To encode the asymmetric path, one uses a `N` bit sequence where `0` is collapsing the argument and `1` is preserving structure, but the last bit for output is removed because collapsing the output leads to the same solution.

11 {[not → not → ?], [not → not → ?], …, [id → id → ?]}
01 {[false_1 → not → ?], [false_1 → not → ?], …, [true_1 → id → ?]}

A such encoding detects repeating patterns in a binary sequence:

```
path(0000, 00) => 1
path(0001, 00) => 0
path(0101, 01) => 1
path(0101, 00) => 0
path(01100110, 011) => 1
path(01100110, 010) => 0
path(01010101, 101) => 1
path(01010101, 001) => 1
path(0110011010011001, 1011) => 1
path(0011110000111100, 0110) => 1
```

The lowest encoding of the second argument collapses as many arguments as possible, and describes the characteristic repeating pattern for a sequence. Yet, since the left side is separated from the right side, this pattern can be non-trivial and difficult to spot in longer sequences:

```
 0110011010011001          1
01100110  10011001          0
   0110      1001           1
  01  10    10  01           1

 0011110000111100           0
      00111100              1
    0011    1100            1
   00 11 11  00              0
```

In a random sequence, a `0` in the characteristic repeating pattern has the same likelihood to occur anywhere, because the chance that an argument is irrelevant to a boolean function is the same no matter which order we put the arguments.

As a random bit sequence increases in size, it is intuitive to assume the chance that a `0` occurs in the characteristic repeating pattern goes down. This is based on the idea that both child nodes are the same for all parent nodes at the same level. At the top, you would have to pick two identical sub trees by random. At the bottom, each parent node at the same level, which are 2^N parent nodes, would have to pick same values of `bool` as leafs with 0.5 probability.

$$0.5\hat{}2^N = 1/(2\hat{}2^N)$$

Arguments	Probability	Fraction
1	0.5	$1/2^1$
2	0.25	$1/2^2$
3	0.175	$1/2^3$

Every irrelevant argument in a boolean function means the same computation could be performed in a lower function space, e.g. instead of `bool → bool → bool` it could be done in `bool → bool`. This means that a random function has a naturally bias to be logically equivalent to a shorter program. The bias can be represented as a conditional probability `P(M | N)` where `M` is number of arguments in the shorter program, and `N` is the number of arguments in the larger function space.

$$P(M \mid N) = (1/2^N)^{N-M}(1-1/2^N)^M$$