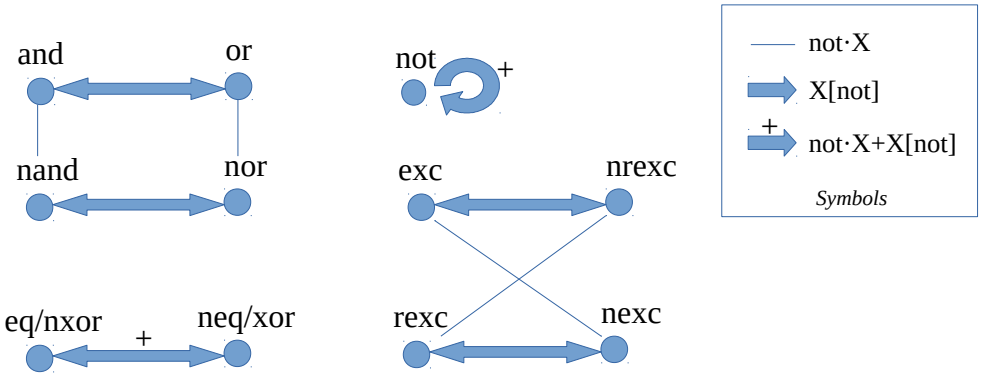
Symmetric paths in Boolean algebra

by Sven Nilsen, 2016



Patterns of Boolean operations by paths and operations of `not`.

Introduction

Path Semantics (started summer 2014) was inspired by homotopy type theory and contextual memory combinatorics (currently unpublished). It is a natural way to express partial knowledge about functions. After two years of research and experiments with interpreters, I believe it will be a promising area for machine learning.

One long term goal of path semantics is to develop efficient algorithms to explore the space of functions. Paths are kind of like prime numbers, which seem to have some sort of system in them, but difficult to tame. This paper is about the simplest collection of functions: Boolean algebra.

Complete list of symmetric paths

The knowledge about Boolean logical gates is derived by inferring symmetric paths, automated by a computer program. It is a basic and important first step for the goal of exploring the space of path semantics efficiently. Here is the complete list:

not[not] <=> not
and[not] <=> or
or[not] <=> and
nand[not] <=> nor
nor[not] <=> nand
eq[not] <=> neq
neq[not] <=> eq
exc[not] <=> nrexc
nrexc[not] <=> exc
rexc[not] <=> nexc
nexc[not] <=> rexc

The meaning of a statement like:

    and[not] <=> or

Is a shorthand for:

    fn and(a: not, b: not) → not { or(a, b) }

This is equivalent to the law:

    forall a, b: bool { not(and(a, b)) = or(not(a), not(b)) }

For Boolean values, one can use equality and inequality instead of `nxor` and `xor`. This is used in the list to avoid duplicates:

    eq <=> nxor
    neq <=> xor

The operation `exc` is exclusion and `rexc` is right exclusion:

    exc(a, b) = a – b = and(a, not(b))
    rexc(b, a) = b – a = and(not(a), b)

Exclusion operators are useful in some Boolean algebras, for example in non-invertible sets where `not` is impossible.

Search algorithm

The pseudo algorithm is the following:

1. Input data to use for search (in this case single and binary Boolean values)
2. For each function `X` and `Y`, find symmetric path candidates `X[Y]` in context of data
3. For each candidate `X[Y]` and function `Z` look for relation `X[Y] <=> Z`
4. Print out relations

Finding symmetric path candidates (Dyon code):

```
/// Infers whether a path can be used for a function
/// in a symmetric way for both arguments and result.
sym(f: \([]) -> any, p: \([]) -> any, args: []) =
    all i {is_ok(try \p([args[i]]))} &&
    is_ok(try \p([\f(args)]))

// Find symmetric path candidates.
find_sym_candidates__functions_data(fs, data) = {
    fs_keys := keys(fs)
    n := len(fs_keys)
    sym_candidates := []
    for i, j {
        a := fs_keys[i]
        b := fs_keys[j]
        if !all h {sym(fs[a], fs[b], data[h])} {continue}
        push(mut sym_candidates, [a, b])
    }
    clone(sym_candidates)
}
```

Finding symmetric paths among candidates (Dyon code):

```
/// Finds symmetric paths.
find_sym_paths__functions_candidates_data(fs, sym_candidates, data) = {
    fs_keys := keys(fs)
    sym_paths := []
    for i {
        a := sym_candidates[i][0]
        b := sym_candidates[i][1]
        fa := fs[a]
        fb := fs[b]
        for j {
            c := fs_keys[j]
            fc := fs[c]
            if all h {
                ls := try \fb([\fa(data[h])])
                rs := try \fc(sift k {\fb([data[h][k]])})
                if is_err(ls) || is_err(rs) {false}
                else {
                    cmp := try unwrap(ls) == unwrap(rs)
                    if is_err(cmp) {false}
                    else {unwrap(cmp)}
                }
            } {
                push(mut sym_paths, [a, b, c])
            }
        }
    }
    clone(sym_paths)
}
```

Full source code:
https://github.com/bvssvni/path_semantics/tree/master/dyon_experiments/sym_extract

Further research

The algorithm used for symmetric path search is very simple, but has a cool property: It finds paths that are consistent with the input. For example:

- When you search with booleans, you get conjectures about booleans
- When you search with numbers, you get conjectures about numbers
- When you search with integers, you get conjectures about integers
- When you search with positive integers, you get conjectures about positive integers
- When you search with lists, you get conjectures about lists

The reason this works is that paths are functions themselves, so the algorithm needs no pre-knowledge. It simply tries everything, but only functions that accepts the input get listed as path candidates. As a result the conjectures it finds are actually about the unknown type and constraints of the data you put in.

For example, the set of positive integers is a subset of the set of integers, so the paths that hold for positive integers includes all paths that holds for integers.

This might be used to learn types of data using only examples and a knowledge base of functions.