Efficient Type Checking

by Sven Nilsen, 2019

A sub-type is isomorphic to propositions about existence of symbols in a free form grammar sentence, joined by `∧` and `⊻` [1]. Since brute-force theorem proving in propositional logic has runtime complexity `O($2^N$)` for `N` propositions, I suggest using the following data structure for type checking (in Rust):

```
enum Adt {
    Leaf(usize),
    Prod(Vec<Adt>),
    Sum(Vec<Adt>),
}
```

The name `Adt` means “Abstract Data Type”.

Here, a leaf refers to some external symbol which operations might depend on context.

A product is ordered, such that the commutative property[2] does not hold:

$a \cdot b = b \cdot a$: false

A sum is unordered, such that the commutative property[2] does hold:

$a + b = b + a$: true

This means that one can prove things like:

$a \cdot (b + c) = a \cdot b + a \cdot c$

Since the product is ordered, one can reuse leaf indices, for example:

```
false := Leaf(0)
true := Leaf(1)
bool := Sum(false, true)

none := Leaf(0)
some(x) := Prod(Leaf(1), x)
opt(x) := Sum(none, some(x))
```

The `none` leaf never collides with `false` leaf, because whenever `none` is used, it is known that the type is `opt[T]` instead of `bool` from the context.

For example, a union sub-type `bool | opt[T]` is defined as following:

```
bool | opt(x) := Sum(Prod(Leaf(0), bool), Prod(Leaf(1), opt(x)))
```

References:

[1] “Propositional Logic as Symbolic Free Form Grammar”
Sven Nilsen, 2019
https://github.com/advancedresearch/path_semantics/blob/master/papers-wip/propositional-logic-as-symbolic-free-form-grammar.pdf

[2] “Commutative property”
Wikipedia
https://en.wikipedia.org/wiki/Commutative_property