Natural Sum Sub-Type Factorization

by Sven Nilsen, 2019

To deal with sub-type theorem efficiently, one must make various assumptions and develop specific tools depending on the assumptions. One trick is to change the semantics of a data type such that theorem proving gets easier, while some information about the original data gets lost.

In the data structure I suggested for Efficient Type Checking[1], intended for dealing with sub-types, the following axiom gives natural behavior under abstract rewriting to normal form[2]:

$$a + a \cdot b = a$$

This is what I call Natural Sum Sub-Type Factorization.

Abstract rewriting is a way of simplifying or preparing expressions for more efficient theorem proving. The `+` operator means an analogue of `⩛` (XOR). The `·` operator means an analogue of `∧` (AND). This algebraic system exploits the fact that a sub-type is isomorphic to propositions in Logic about existence of symbols in a free form grammar sentence, joined by `∧` and `⩛`[3]. In Logic, the order of symbols is lost, but here, in the algebraic form, the order is preserved. A product `a · b` is interpreted as first matching against some symbol `a` and then `b`.

One can think about this law as a special case of sum[5] sub-type factorization:

$$a \cdot 1 + a \cdot b = a \cdot (1 + b)$$

The intuition behind the element `1` in this algebra is like a wildcard symbol: It matches all symbols, such that `1 + b = 1`. Since `a · 1 = a`, the result is just `a`. The data structure I suggested has no element for `1`, so the axiom must be used instead.

One benefit of having two interpretations, one in Logic and one in the algebraic form, is that our intuition about correct behavior can be checked. This is the case here. While the axiom might seem fine on the surface, a problem occurs when you try translating into Logic. The problem is that `b` collides with the constraints of XOR and is set to `false`. Our intuition about `b` says that it can be `false` or it can be `true`: It does not matter whether this symbols occurs. A proof of a second solution is needed.

To solve this issue, one must use Natural Implication[2]. Natural Implication is used when struggling with false assumptions, and there is no easy way to update them. An extra proposition is introduced in the proof to keep track of soundness. The other propositions can then be equal to `true`, or equal to `sound`. This is sufficient to prove[6] that `b` is `true`, but also `unsound`:

$$\frac{(A \veebar A \wedge B) = \text{sound} \qquad \wedge \qquad A}{(B = \text{true}) \vee (B = \text{sound}) \quad = \quad B}$$

Natural Implication supports the intuition of sum sub-type factorization such that `B` has two solutions.

References:

[1] “Efficient Type Checking”
Sven Nilsen, 2019
https://github.com/advancedresearch/path_semantics/blob/master/papers-wip/efficient-type-checking.pdf

[2] “Normal form (abstract rewriting)”
Wikipedia
https://en.wikipedia.org/wiki/Normal_form_(abstract_rewriting)

[3] “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

[4] “Natural Implication”
Sven Nilsen, 2019
https://github.com/advancedresearch/path_semantics/blob/master/papers-wip/natural-implication.pdf

[5] “Tagged union”
Wikipedia
https://en.wikipedia.org/wiki/Tagged_union

[6] “Pocket-Prover”
AdvancedResearch, Sven Nilsen
https://github.com/advancedresearch/pocket_prover