Simple Structure Logic

by Sven Nilsen, 2020

In this paper I represent a logic for simple data structures as a small subset of path semantics.

A simple structure logic is a logic of some type \`T\` with some list of ordered properties \`$p_i$\`:

$$p_i : T \to U_i$$

Together with Boolean Algebra of sub-types over \`$p_i$\` using Higher Order Operator Overloading.

The properties are ordered to permit transformation of expressions into canonical form.
This can be used to prove whether two sub-types are equal.

Every property can be expressed with the following sub-types:

$$x : [p_i]\ (= y) \qquad\qquad x : [p_i]\ (\neg= y)$$

$$x : T$$
$$y : U_i$$

When \`$U_i$\` is ordered, the property can also be expressed with the following sub-types:

$$x : [p_i]\ (< y) \qquad x : [p_i]\ (<= y) \qquad x : [p_i]\ (>= y) \qquad x : [p_i]\ (> y)$$

The order of comparison operators is \`<, <=, =, ¬=, >=, >\`.

In simple structure logic, every data record can be translated into a sub-type such that:

$$|\cap\ i\ \{\ [p_i]\ (= y_i)\ \}| = 1$$

For example, if a \`person\` is uniquely determined by a \`name\` and \`age\` property:

$$|[age]\ (= 20) \wedge [name]\ (= \text{“Hans”})| = 1$$

$$age : person \to nat$$
$$name : person \to str$$

The properties are ordered such that for some finite \`n\`, the properties \`$p_0, p_1, \ldots, p_n$\` constructs every possible data record. Generically, this form can be written in a short hand syntax:

$$[p_{in}]\ y_{in} \qquad <=> \qquad \cap\ i\ n\ \{\ [p_i]\ (= y_i)\ \}$$

$$|[p_{in}]\ y_{in}| = 1$$

Sometimes the list \`$p_i$\` is infinite, for example by composing functions within a programming language.

A canonical form can be chosen e.g. by using Conjuctive Normal Form (CNF).

For example, it is known from the law of distribution that the following expressions are the same:

$$a \wedge (b \vee c) \quad <=> \quad (a \wedge b) \vee (a \wedge c)$$

The expression `a ∧ (b ∨ c)` is in CNF, therefore one can apply the rule of transformation:

$$(a \wedge b) \vee (a \wedge c) \quad => \quad a \wedge (b \vee c)$$

CNF uses these connectives of Boolean Algebra:

$\wedge$	AND	Commutative: $a \wedge b = b \wedge a$	Associative: $(a \wedge b) \wedge c = a \wedge (b \wedge c)$
$\vee$	OR	Commutative: $a \vee b = b \vee a$	Associative: $(a \vee b) \vee c = a \vee (b \vee c)$
$\neg$	NOT	Single-argument	

Transformation into CNF uses Negation Normal Form, such that negation is only applied to variables.
This means that in simple structure logic, negation is only applied to sub-type properties.

However, simple structure logic can eliminate any negation applied to a sub-type property:

$$\neg[p]\ (< y) \quad <=> \quad [p]\ (>= y)$$
$$\neg[p]\ (<= y) \quad <=> \quad [p]\ (> y)$$
$$\neg[p]\ (= y) \quad <=> \quad [p]\ (\neg= y)$$

Therefore, the canonical form based on CNF does not require `¬`. Only `∧` and `∨` are needed.

Yet, there is one edge case: Boolean properties are ambiguous since the value can be inverted:

$$[p]\ (= \text{false}) \quad <=> \quad [p]\ (\neg= \text{true})$$
$$[p]\ (= \text{true}) \quad <=> \quad [p]\ (\neg= \text{false})$$

This means that one can choose to express `false` as one of the following:

$$[p]\ (= \text{false}) \qquad [p]\ (\neg= \text{true})$$

The standard convention is to use `[p] (= false)`, due to construction of data records.

AND and OR might be represented internally as multi-argument functions.
This semantics is relative to representation with binary functions, where the following order holds:

$$\text{order}(a \wedge (b \wedge c)) < \text{order}((a \wedge b) \wedge c)$$
$$\text{order}(a \vee (b \vee c)) < \text{order}((a \vee b) \vee c)$$

The semantics of AND and OR as multi-argument functions is the least order:

$$\text{and}(a, b, c) \quad <=> \quad a \wedge (b \wedge c)$$
$$\text{or}(a, b, c) \quad <=> \quad a \vee (b \vee c)$$