Reduction of Proofs
With Multiple Constraints

by Sven Nilsen, 2017

When you have a variable \`a\` with multiple constraints:

$$a : [f]\ b \wedge [g]\ c$$

$$f : A \to B$$
$$g : A \to C$$

This is equivalent to:

$$a : [f\{[g]\ c\}]\ b \wedge [g\{[f]\ b\}]\ c$$

One could reason that type checking requires only proving that the existential path of either sub-type returns \`true\`. If one of the existential paths returns \`true\`, then the other must also return \`true\`.

$$b : [\exists f\{[g]\ c\}]\ \text{true} \quad <=> \quad c : [\exists g\{[f]\ b\}]\ \text{true}$$

$$\exists f\{[g]\ c\} : B \to \text{bool}$$
$$\exists g\{[f]\ b\} : C \to \text{bool}$$

This is a way to reduce the amount of proof burden required to check for consistency. It works even when the sub-types are defined by two different types.

For example:

$$a : (< 100) \wedge \text{even}$$
$$a : [(< 100)]\ \text{true} \wedge [\text{even}]\ \text{true}$$
$$a : [(< 100)\{[\text{even}]\ \text{true}\}]\ \text{true} \wedge [\text{even}\{[(< 100)]\ \text{true}\}]\ \text{true}$$

$$(< 100)\{[\text{even}]\ \text{true}\} : \text{nat} \to \text{bool}$$
$$\text{even}\{[(< 100)]\ \text{true}\} : \text{nat} \to \text{bool}$$

If you take the existential path of numbers that are less than 100, constrained by the even numbers, then we know it returns \`true\` for even numbers less than 100 and \`false\` otherwise. Likewise, if you take the existential path of even numbers, constrained by numbers less than 100, then we know it returns \`true\` for even numbers less than 100 and \`false\` otherwise.

$$\text{true} : [\exists(< 100)\{[\text{even}]\ \text{true}\}]\ \text{true} \quad <=> \quad \text{true} : [\exists \text{even}\{[(< 100)]\ \text{true}\}]\ \text{true}$$

$$\exists(< 100)\{[\text{even}]\ \text{true}\} <=> \text{true}_1$$
$$\exists \text{even}\{[(< 100)]\ \text{true}\} <=> \text{true}_1$$

∃(< 100){[even] true} : bool → bool
∃even{[(< 100)] true} : bool → bool

Another example:

a : [(% 3)] 0 ∧ [even] true
a : [(% 3){[even] true}] 0 ∧ [even{[(% 3)] 0}] true

(% 3){[even] true} : nat → nat
even{[(% 3)] 0} : nat → bool

In this case the sub-types are defined by different types, and the existential paths are of different types:

0 : [∃(% 3){[even] true}] true <=> true : [∃even{[(% 3)] 0}] true

∃(% 3){[even] true} : nat → bool
∃even{[(% 3)] 0} : bool → bool

There are both even and odd numbers which are divisible by 3, and therefore the domain constraint has no effect. Still, the sub-type is only consistent if the existential path returns `true`, which it does on `0`.

∃(% 3){[even] true} => ∃(% 3)

∃(% 3){[even] true} := \(x : nat) = x < 3
∃even{[(% 3)] 0} <=> true[1]

(∃(% 3){[even] true})(0) = 0 < 3 = true
(∃even{[(% 3)] 0})(true) = true

These two types of existential paths might seem different, but they are secretly connected to each other. If one of them returns `true`, then the other must return `true`. If one returns `false`, then the other can not return `true`, so it must return `false`.

Here is another example:

a : [(% k)] 1 ∧ [even] true

For which values of `k` is this sound? If `k` is even, the modulus lines up with even numbers:

∃(% k){[even] true} := \(x : nat) = x < k && (¬even(k) || even(x))

(∃(% k){[even] true})(1) = 1 < k && (¬even(k) || even(1))
(∃(% k){[even] true})(1) = 1 < k && (¬even(k) || false)
(∃(% k){[even] true})(1) = 1 < k && ¬even(k)

Therefore, it is only sound when `k` is greater than `1` and `k` is not even. One could also say:

k : [linear(3, 2)] true