Existential Path of `add` with Domain Constraint `even`

by Sven Nilsen, 2017

Every natural number `a` can be written as a sum of two natural numbers, because:

a + 0 = a

In path semantics, the idea that every natural number can be written as a sum of two natural numbers, can be expressed in the following way:

∃add <=> $true_1$

add := \(a: nat, b: nat) = a + b
$true_1$:= \(_) = true
∃add : nat → bool

The function `∃add` tells which numbers that are returned by the function `add`.
The constraint `[∃add] true` defines the codomain of the function `add`.
Since `add` can return any number, the function `∃add` always return true, so `[∃add] true` is equal to the set of all natural numbers.

Now, I will add a domain constraint `even` on the arguments of `add`:

∃add{[even] a, [even] b} := \(x) = if x == 0 { a && b } else { (a == b) == even(x) }

even := \(a: nat) = (a % 2) == 0

This adds two parameters that tell whether the arguments should be even or odd numbers.
The constraints on the input are different, so telling which numbers that are returned by `add` becomes more complex.

Intuitively, one can understand it as a compact representation as a collection of knowledge:

For any natural number, can you tell me whether it is possible to construct it by
adding two <even/odd/one even and one odd> natural number(s)?

When I have two odd numbers, I can reduce the formula to construct any `x` with the property:

if x == 0 { false && false } else { (false == false) == even(x) }
if x == 0 { false } else { true == even(x) }
if x == 0 { false } else { even(x) }

I will now prove the algorithm above.
The strategy used is to break down the constraints in cases and then unify the results.

add[even] <=> eq
This symmetric path is well known in path semantics.
It says how to predict the even property of the output from the even property of the inputs.

f_{00} = ∃add{[even] false, [even] false}
f_{00}(_: [even] false) = false
Since `add[even](false, false) = true`, it returns no odd numbers.
f_{00}(x: [even] true) = x ¬= 0
All even numbers can be constructed by adding two numbers, except 0.
This is because the smallest odd number is 1, so the smallest even number, that can be constructed by adding two odd numbers, is 2.

f_{01} = ∃add{[even] false, [even] true}
f_{01}(_: [even] false) = true
Since `a + 0 = a` and `a` can be odd, all odd numbers are returned.
f_{00}(_: [even] true) = false
Since `add[even](false, true) = false`, it returns no even numbers.

f_{10} <=> f_{01}
The `add` function is associative, so proofs of `$f_{01}$` are reused for `$f_{10}$`.

f_{11} = ∃add{[even] true, [even] true}
f_{11}(_: [even] false) = false
Since `add[even](true, true) = true`, it returns no odd numbers.
f_{11}(_: [even] true) = true
Since `a + 0 = a` and `a` can be even, all even numbers are returned.

(∃add{[even] a, [even] b})(x) = if a {if b {even(x)} else {¬even(x)}}
else {if b {¬even(x)} else {even(x) && x ¬= 0)}}
(∃add{[even] false, [even] false})(x) = even(x) && x ¬= 0
(∃add{[even] false, [even] false}{[even] false})(_) = false
(∃add{[even] false, [even] false}{[even] true})(x) = x ¬= 0
(∃add{[even] false, [even] true})(x) = ¬even(x)
(∃add{[even] false, [even] true}{[even] false})(x) = true
(∃add{[even] false, [even] true}{[even] true})(x) = false
(∃add{[even] true, [even] false})(x) = ¬even(x)
(∃add{[even] true, [even] false}{[even] false})(x) = true
(∃add{[even] true, [even] false}{[even] true})(x) = false
(∃add{[even] true, [even] true})(x) = even(x)
(∃add{[even] true, [even] true}{[even] false})(x) = false
(∃add{[even] true, [even] true}{[even] true})(x) = true

Notice the parentheses since the existential path operator `∃` has low precedence:

(∃add{[even] a, [even] b})(x) <=> (∃add{[even], [even]}(a, b))(x)

The two expressions to check for equality are:

A: if a {if b {even(x)} else {¬even(x)}}
else {if b {¬even(x)} else {even(x) && x ¬= 0)}}

B: if x == 0 { a && b } else { (a == b) == even(x) }

To check for equivalence, I use brute force because `even(x)` is always true when ` x == 0` is true.
The two expressions are allowed to return different values when violating this constraint.

a	b	even(x)	x == 0	A	B
0	0	0	0	0	0
0	0	0	1	0	0
0	0	1	0	1	1
0	0	1	1	0	0
0	1	0	0	1	1
0	1	0	1	1	0
0	1	1	0	0	0
0	1	1	1	0	0
1	0	0	0	1	1
1	0	0	1	1	0
1	0	1	0	0	0
1	0	1	1	0	0
1	1	0	0	0	0
1	1	0	1	0	1
1	1	1	0	1	1
1	1	1	1	1	1

One can generate a sequence from all natural numbers that are returned by `add`:

add_sequence := \(a, b, x) = if a ¬= b { 1 + 2·x } else if a && b { 2·x } else { 2 + 2·x }

∃add_sequence(a, b) <=> ∃add{[even] a, [even] b}
∃add_sequence(a, b) <=> ∃add{[even], [even]}(a, b)

Notice that the existential path operator `∃` has low precedence, such that:

∃add_sequence(a, b) <=> ∃(add_sequence(a, b))

The following is not equivalent to the above, because it returns `true` for all natural numbers:

∃add_sequence <=> ∃add{[even], [even]} <=> $true_1$