Existential Paths

by Sven Nilsen, 2017

In this paper I show that path semantics requires a kind of source code transformation that gives a natural semantics of set theory grounded in functions. Furthermore, one can use this kind transformation to directly describe and reason about empty vs non-empty types, halting programs and surjective properties of functions in path semantical notation. This is directly applicable for reasoning about pre- and post-conditions in programming. I also derive the transformations for basic arithmetic. Finally, I show that existential paths might be used to make a working sub type checker using path semantics!

In path semantics, when a variable is a sub type defined by some function \`g\`, the output of \`g\` must automatically satisfy the requirement that some input exists:

a : [g] b
b : [∃g] true

g : A → B
∃g : B → bool

The function \`∃g\` is total and called an "existential path" of \`g\`, which determines the truth value:

∃x { g(x) = b }

An existential paths satisfies an asymmetric path such that for any input, the output is output:

g[id → ∃g] <=> $true_1$

$true_1$: A → bool
$true_1$ = \(_) = true

This proves that the existential path is unique for the output of \`g\`, because asymmetric paths map deterministically from Nilsen cartesian products. For the non-output of \`g\` we return \`false\` by definition, which makes the existential path unique for all input and possible to construct automatically for functions with finite output.

By looking at the following:

a : [g] b
b : [∃g] true

We see that there exists a second order existential path:

true : [∃∃g] true

∃∃g : bool → bool

The second order existential path is one of four possible boolean functions:

$\exists\exists g \iff false_1$	The type `B` is empty
$\exists\exists g \iff not$	The function `g` never halts
$\exists\exists g \iff id$	The function `g` is surjective (outputs all)
$\exists\exists g \iff true_1$	The function `g` is non-surjective (outputs some)

Since every function has one of the four second order existential paths, but those functions themselves have existential paths, we can study the “end game” of all existential paths:

f	**∃f**	**$(\exists\exists)^n f$, n > 1**	**$\exists(\exists\exists)^n f$, n > 1**
$false_1$	not	$true_1$	id
not	$true_1$	id	$true_1$
id	$true_1$	id	$true_1$
$true_1$	id	$true_1$	id

Of course, all these functions halts and the boolean type is non-empty, so the only survivors are the `id` and `$true_1$` functions.

Surprisingly, a second order existential path `not` means that the function never stops running. This is because there is no output while the type has members. An existential path is total, so it can not be the case that the function has not yet been implemented, because all first order existential paths are implemented for their type by definition. Therefore, the only interpretation left is that the function never stops running, or that the language used to define functions is not Turing complete.

This means every halting function is either surjective or non-surjective and has two corresponding existential paths:

$\exists f \iff f'$
$\exists f' \iff \{ id? \mid true_1? \}$

$f : A \to B$
$f' : B \to bool$

The existential path determines whether some input exists for some output. One can also say that when a function defines the construction of a set, its existential path determines whether any member belongs to the set. You get set theory for free!

Yet, this is not like ordinary set theory where they are taken as primitive build blocks, but instead those sets are relative to some type `A` with a function that maps from `A` to the set `B`.

There are some known existential paths, but most of them are unknown, since there are infinitely many of them. One application of using known existential paths is to derive existential paths from compositions. In software verification this has great utility.

For example, for natural numbers the existential functions of addition and multiplication are known:

∃add(k) <=> ∃(+ k) <=> (>= k)
∃mul(k) <=> ∃(* k) <=> (= 0) || [% k] 0

∃(>= k) <=> if k == 0 {id} else {true_1}
∃(= 0) <=> true_1
∃{\(x) = (x % k) == 0} <=> if k == 1 {id} else {true_1}

add : nat → nat → nat
mul : nat → nat → nat

In natural language, one can say “adding some number gives a larger number than the number added”, and “multiplying some number is either 0 or divisible by the number multiplied with”. Multiplication has a lazy boolean OR (`||`) since you can not calculate with the reminder of a division of zero.

We can find the second order existential path of multiplication:

∃∃(* k)
∃{(= 0) || [% k] 0}
{∃(= 0) || ∃{\(x) = (x % k) == 0}}
if k == 0 {true_1} else if k == 1 {id} else {true_1}

This tells us that when multiplying with `1`, it is surjective, but otherwise it is non-surjective.

Notice that the existential paths corresponds to post-conditions in programming. Since the reverse operations of `add` and `mul` are `sub` and `div`, they have similar pre-conditions:

sub(a : (>= b), b) { … }
div(a : [% b] 0, b : (!= 0)) { … }

∃∃(- k) <=> true_1
∃∃(/ k) <=> id

sub : nat → nat → nat
div : nat → nat → nat

Addition has an existential path with greater or equal, so we can figure out the other comparisons:

(>= a) <=> ∃add(a)
(> b) <=> ∃add(b + 1)
    a : (> b)
    a : (>= (b + 1))
{a : (< b)} <=> {b : ∃add(a + 1)}
    a : (< b)   &   b : (> 0)
    b : (> a)
{a : (<= b)} <=> {b : ∃add(a)}
    a : (<= b)
    b : (>= a)

The existential paths of all comparisons:

∃(>= k) <=> if k == 0 {id} else {true$_1$}
∃(> k) <=> true$_1$
∃(< k) <=> if k == 0 {id} else {true$_1$}
∃(<= k) <=> true$_1$

The modulus operation (division reminder) returns a number starting at `0` and counting up to `k-1` before it starts again at `0`. This means it only returns values less than `k`:

∃(% k) <=> (< k)

So, if `a` is some output of modulus of `k`, then it must be less than `k`:

a : ∃(% k)
a : (< k)
a < k

This concludes the first and second order existential paths of basic arithmetic.

Functions with pre-conditions are partial, so they propagate constraints to the left side:

g(_ : [h] x) = { … }	`g` has a pre-condition defined by `h` and `x`
a : [g] b	`a` has a sub type defined by `g` and `b`
b : [∃g] true	`b` inherits post-condition from `g`
a : [h] x	`a` inherits pre-condition from `g`, a sub type defined by `h`
x : [∃h] true	`x` inherits post-condition from `h`
…	`a` inherits pre-condition from `h` etc.

When a variable is set to a constant on the left side, we can just evaluate the right side to check whether the condition of the sub type holds. When we have an unknown on the left side, we must expand the conditions on the right side and repeat the process. Using existential paths alone, we can not ensure correctness of the program, but if we enumerate/prove using sets that the right side is non-empty, then the program is correct. The challenge is finding enough existential paths for this to be useful.

Therefore, existential paths might be used in a complain-when-wrong sub type checker!