Constrained Functions

by Sven Nilsen, 2017

In path semantics it is normal to watch out for domain constraints because it changes function identity. Here I introduce a new concept called "constrained functions" that have built-in constraints with the property that function identity is preserved in path semantics. This gives rise to a new interpretation of functions that makes building theorem provers for path semantics more practical. I also show some properties of such functions and demonstrate the application of propagated function constraints to test whether a function can exist for some known path. These kind of propagations are directly applicable to complain-when-wrong type checking for path semantics.

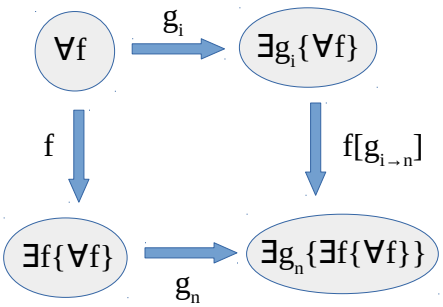
The major result of this paper is the following (trivial using the rule of composition, but most useful):

$$\exists f[g_{i\to n}]\{\exists g_i\{\forall f\}\} <=> \exists g_n\{\exists f\{\forall f\}\}$$

In the semantics of the new interpretation, one is allowed to write (notice `$\exists f\{\forall f\} <=> \exists f$`):

$$f[g_{i\to n}] : \exists g_i\{\forall f\} \to \exists g_n\{\exists f\}$$

As shown here in the diagram:



This provides a way to check code with path semantics using a complain-when-wrong type checker. A such type system will be accurate up to function types that belong to same output permutation group (ignoring cardinality swapping).

An old idea, that constraints propagates and composes has been around since the start of path semantics. However, the old idea was only considering sub-types defined by functions as first class citizens. The problem was that there is no natural choice of sub-types for intermediate steps without involving machine learning of maximizing predictability by probabilistic paths.

The new idea is to put domain constraints and existential paths inside functions as secrets, which results in a different dynamics. These sub-types are the largest sub-type that characterizes the set of variables and can be derived at compile time at higher order, but also satisfies the runtime properties required in dependently type systems.

The higher order representation corresponds to a hidden stack inside closures that is introspective using De Bruijn indices plus the identifier of the canonical form of the higher order function. In practice this corresponds precisely to the semantics of normal objects, which makes it practical to write libraries for path semantical theorem proving, e.g. for linear algebra, in Rust and other mainstream languages.

For example, consider the addition of natural numbers:

```
add : nat → nat → nat
∃add(k) <=> (>= k)
```

An existential path, notated with the symbol `∃` in front of the function `add(k)`, transforms to another function that tells which values are returned. The existential path of `add(k)` for natural numbers is a function that returns `true` if the input is greater or equal to `k` and false otherwise, written `(>= k)`.

Intuitively, if you add `0` and `k` you get `k`. If you add `1` and `k` you get `k + 1`. No matter what natural number you add with `k`, you get something greater than or equal to `k`. This means the function `(>= k)` characterizes the largest sub-type of the output:

```
nat ∧ (>= k)
```

In the traditional interpretation of functions, the problem is that `k` is captured and not available to the existential path at runtime. It seems obvious that the `k` in `add(k)` is transferred to `(>= k)` somehow, but exactly how to describe this in code has been a long term problem in path semantics.

Using a new interpretation of functions called “constrained functions”, one can think of `add` as an object `Add`. When applying it to `k` the result is `AddK`, another object with a family of existential paths over `k`. The `AddK` object captures `k` just like a closure, but makes it available for functions accepting a type of `AddK`.

For example, in Rust one could start coding like this:

```
pub struct Add<T> { t: PhantomData<T> }
pub struct AddK<T> { pub k: T }
pub struct GeK<T> { pub k: T }
```

These objects corresponds to higher order representations of closures under normal computing. In path semantics one would like to take existential path transforms, so two new members are added:

```
pub struct Add<T, I = (), O = ()> { t: PhantomData<T>, pub i: I, pub o: O }
pub struct AddK<T, I = (), O = ()> { pub k: T, pub i : I, pub o: O }
pub struct GeK<T, I = (), O = ()> { pub k: T, pub i: I, pub o: O }
```

This allows one to e.g. add constraints on the second parameter `add(2)(x: (>= 3)) : (>= 2 + 3)`:

```
let addk: AddK<u32, GeK<u32>, GeK<u32>> = AddK {
    k: 2,
    i: GeK { k: 3, i: (), o: () },
    o: GeK { k: 2 + 3, i: (), o: () }
};
```

Notice that this does not represent a final computed value, but the state of a closure while allowing introspection into the captured variables. The captured values of the sub-types of the input and output are available at runtime while the higher order sub-type is available at compile time.

As long the next function accepts a domain constraint `AddK` when computing its existential path, the captured variable `k` can be transmitted from `AddK` to the next function as a secret. The type system of the host language ensures that there is logical equivalence by normal type checking.

This means the traditional interpretation of functions is replaced by a new concept where functions have constraints, and these constrains for a function `f` are referred to as:

$f : A \to B$
$\forall f : A \to bool$ (input constraint)
$\exists f : B \to bool$ (output constraint)

In the full theory of path semantics, `∀` and `∃` are treated as higher order functions which can be used to define sub-types of functions. It allows expressing ideas in a concise form that otherwise would take long time to digest.

Because the new interpretation of functions, called “constrained functions”, deviates from the traditional interpretation (only according to path semantics, but for normal computation it looks the same), the notation is defined such that ideas from the new theory can be expressed in a compatible syntax with the old theory.

This allows people to use the traditional semantics of functions when they are uncomfortable or feel uncertain about whose constraints are referred to from which place in the proof. In implementations using a programming language with type classes, one can trust the type checker and use type annotations to clarify what is going on.

For normal paths, written e.g. `f[g]`, the semantics is the same as before.

Motivation

An existential path tells whether some function `f` returns an output or not:

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

A domain constraint before taking the existential path changes the function identity, because the existential path changes, and this would violate path semantics if the function identity did not change:

$$\exists f\{g\} : B \to bool$$
$$g : A \to bool$$

A universal existential path is a higher order path semantical function that can determine the existential path for all domain constraints of some function:

$$\exists f\{\} : (A \to bool) \to (B \to bool)$$

While universal existential paths are extremely powerful, it is possible to invent an even more powerful concept of that knows the universal existential path for any function and constraints:

$$\exists : \{a : type, b : type\} \to (a \to b) \to (a \to bool) \to (b \to bool)$$

This higher order path semantical function means the same as the concept of “existential path” in general, so it is not given its own name, but simply called “existential path”.

For example, for a function type `A → B` the existential has the type:

$$\exists : (A \to B) \to (A \to bool) \to (B \to bool)$$

The existential path is so complex that it can not be written down in code. To overcome this problem there is a simple trick: Instead of building an impractical theory out of the traditional concept of functions, one can create a new concept of functions for which the existential path has reduced type:

$$\exists : \{a: type, b : type\} \to (a \to b) \to (b \to bool)$$

This new concept for functions satisfying this criteria is called a “constrained function”.

For example, for a constrained function type `A → B` the existential path has the sub-type:

∃ : (A → B) → (B → bool)

Notice that this is the same type as before except the second argument has disappeared. So, where does the existential path get the second argument from? By building the domain constraint as a “secret” into the function itself!

To extract the secret from functions, a new higher order function is introduced called “the trivial path”:

∀ : {a : type, b : type} → (a → b) → (a → bool)

For example, for a constrained function type `A → B` the trivial path has the sub-type:

∀ : (A → B) → (A → bool)

To make this theory compatible with the theory using traditional interpretation of functions, one can specify new domain constraints that overrides the existing constraints. Overriding existing constraints with the existing constraints results in the exact same constrained function:

∃f{∀f} <=> ∃f

Old Theory	New Theory	Compatibility Syntax
f : A → B	f : A → B	-
g : A → bool	∀(f) : A → bool	∀f
∃f{g} : B → bool	∃(f) : B → bool	∃f{∀f}

One can think of a constrained function as making a specific choice of which secret to store inside it. For every function in the old theory, there is a family of functions in the new theory where each member behaves the exact same under computation in the old theory. This also means that the traditional interpretation of functions is a perfect predictor of the new interpretation, a kind of higher order path between the two theories.

Notice that since computations in the new theory are allowed to depend on values returned by `∀` and `∃`, it can perform computations that are not possible in the old theory without simulation.

The use of secrets also means that functions are introspective, like objects, so adding additional secrets is not a problem. It turns out that this technique is very useful to implement `∃` in code, because captured variables by closures can now be accessed in a consistent way by a function:

k : {a : type, b : type} → (a → b) → any

For example:

add(2) : nat → nat
k(add(2)) = 2

Two useful concepts for programming languages with constrained functions

To preserve soundness in programs using constrained functions, it is useful to introduce two concepts:

- Type check fails when a new existential path is not found when overriding a domain constraint
- Named finite sets of functions

By failing type checking when a new existential path is not found, it makes it easier to track down errors since a lot can happen in the meantime before calling `∃`.

```
f{g}            // ERROR: Could not find new existential path
```

Named finite sets of functions are useful because a set of functions can have some known properties, and the type checker can rely on looking up these properties by the name of the set. This is much easier than trying to derive the functional properties at runtime.

```
list_function {
        fn concat(a: [any], b: [any]) → [any] { … }
        fn len(a: [any]) → nat { … }
        print_list        // Imported from elsewhere.
}

fn foo(f : list_function) { … }

foo(concat)                 // OK
foo(len)                    // OK
foo(bar)                    // ERROR: Expected `list_function`, found `bar`
foo(list_function[0])       // OK
```

The compiler can use named finite sets of functions as a way to check the totality of self-knowledge for theorem proving. For example, it could answer queries like “can I derive all paths from this set of functions using compositions of functions from that set?”. If some knowledge is missing, one could add extra lines of code to instruct the type checker, and this information can be automatically extracted for improving the next version of the compiler.

```
∃(len{[foo] 3}) => true_1
```

Function currying

When evaluating the type of expressions using function currying, the boundary between trivial and existential path gets blurred. The reason for this is that function currying does not distinguish clearly between input and output:

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

$\forall(f) : A \to bool$	$\exists(f) : (B \to C) \to bool$	Alternative 1
$\forall(f) : A \to B \to bool$	$\exists(f) : C \to bool$	Alternative 2
$\forall(f) : A \to B \to C \to bool$	$\exists(f) : bool$	Does this make sense?
$\forall(f) : bool$	$\exists(f) : (A \to B \to C) \to bool$	Or, does this make sense?

The known approaches to solving this problem is to 1) infer from the return type 2) return a tuple of functions and 3) abandon function currying. In dynamically typed languages one can not infer from the return type, so the only possible way is to return a tuple.

In a dependently typed language under the traditional interpretation of functions, one could say that, if one knows the universal existential path, it is possible to transform the dependent function type:

$$f : (a : A) \to B(a) \to C$$
$$\exists(f) : (a : A \to bool) \to (B(a) \to bool) \to (C \to bool)$$

Notice how neat this fits together with function currying of domain constraints. Unfortunately, the universal existential path is a beast that is extremely hard to tame.

Inferring from return type

In a dependently typed language using constrained functions, perhaps one could do this:

$$f : (a : A) \to B(a) \to C$$

$$\forall(f) : (a : A) \to B(a) \to bool$$
$$\exists(f) : C \to bool$$

$$\forall(f) : (a : A) \to bool$$
$$\exists(f) : (\{a : A\} \to B(a) \to C) \to bool$$

The function return type must implicitly be able to determine the variable `a` which the second argument `B(a)` depends on. If there is no inverse `$B^{-1}$`, then type checking could fail. This is a place where named sets of functions might help the type checker.

Return a tuple of functions

For constrained functions, another way to work around the problem of ambiguity by function currying is to return a dependent tuple of the constraints. In the example above, this means the trivial path is obtained by accessing the first two components of the tuple:

$$f : (a : A) \to B(a) \to C$$
$$\exists(f) : (a : A \to \text{bool}, B(a) \to \text{bool}, C \to \text{bool})$$

This can be problematic because in a dependent tuple one need variables to determine the type for the rest of the tuple. Perhaps the tuple can have a dual representation that treats `a` with “for all” quantifier.

$$\exists(f) : C \to \text{bool} \qquad \text{for all } `a`$$

Still, even though the details are missing, the tuple picture can be quite useful to understand what happens under function currying:

$$\exists(f(k)) : (B(k) \to \text{bool}, C \to \text{bool})$$

Now that `a` take on a value `k` it is no longer “for all” `a`, but for a specific value. Notice that the constraints are changing when applying `f` to a value. They are not the same for dependently types:

$$\exists(f) <=> (g_0 : (a : A), g_1 : B(a), \exists f)$$
$$\exists(f(k)) <=> (g_2 : B(k), \exists f(k))$$

This means that constraints are propagating to the returned constrained function `f(k)`.

Properties of constrained functions

Now that `∀` and `∃` are functions accessible from within the program, it is possible to specify sub-types that uses these functions:

f : [∀] g ∧ [∃] h

If you restrict `g` and `h` when `∀f <=> g` and `∃f <=> h`, then you are changing the type of `f`. Even though you do not need to watch out for domain constraints on `f`, you still have to watch out for domain constraints on exposed functions.

Assume function `f` has type `A → B`, then we know:

∀ : (A → B) → (A → bool)
∃ : (A → B) → (B → bool)

Therefore:

g : A → bool
h : B → bool

As you might know from reduction of proofs with multiple constraints, this is equivalent to:

f : [∀{[∃] h}] g ∧ [∃{[∀] g}] h

To check for consistency it is sufficient to check one of the following:

g : [∃∀{[∃] h}] true
h : [∃∃{[∀] g}] true

Making an attempt to break down `g : [∃∀{[∃] h}] true` in order to understand it:

∀{[∃] h}	`∀` can be constrained by `[∃] h` because `∃` takes type `A → B`
∃∀{[∃] h}	This has type `(A → bool) → bool` which can take `g`
∃∀{[∃] ∃f}	Since `h <=> ∃f` this restricts `∀` to functions `x` where `∃(x) <=> ∃f`
[∃∀{[∃] h}] true	?

It is quite hard to wrap your head around the first version. Trying the other one `h : [∃∃{[∀] g}] true`:

∃{[∀] g}	`∃` can be constrained by `[∀] g` because `∀` takes type `A → B`
∃∃{[∀] g}	This has type `(B → bool) → bool` which can take `h`
∃∃{[∀] ∀f}	Since `g <=> ∀f` this restricts `∃` to functions `x` where `∀(x) <=> ∀f`
[∃∃{[∀] ∀f}] true	Since `h` is returned by `∃` when restricted to functions with `∀f`

Success! It is subtle, but at least no major obstacle in the second version, which is all we need.

One can think of `f` as a normal function that has two associated functions `∀f` and `∃f{∀f}`. When this is true by construction, it automatically satisfies the consistency criteria above. When these functions can be obtained from `f`, it suffices to check for logical equivalence with other functions.

Once you can talk about functions in this kind of third perspective, it is also true that:

$$f : [g]\ true \to [h]\ true$$

This can be shortened down to:

$$f : g \to h$$

Now, remember that this is a theory of constrained functions only. Since `g` and `h` are constrained functions too, they have their own trivial paths `∀g` and `∀h`.

$$\forall g : A \to bool$$
$$\forall h : B \to bool$$

The input to `f` is constrained by `g`, so `g` must at least take any value that one can pass to `f`. This means that `f` also has the sub-type:

$$f : [\forall g]\ true \to [\forall h]\ true$$

Which can be reduced to:

$$f : \forall g \to \forall h$$

In general:

$$(f : g \to h) \Rightarrow (f : \forall g \to \forall h)$$

This way of reasoning about functions allows new kinds of expressions. I can express this:

$$(f : \forall g \to \exists g) \Leftrightarrow (g : \forall f \to \exists f)$$

It means `f` and `g` has some overlap in their domains and maps to the same codomain.

The following can be true without `g` having the type `A → bool`:

$$f : [\forall]\ \forall g$$

Finally, constraints carry over for all compositions, so a path inherits constraints:

$$\exists f[g_{i \to n}]\{\exists g_i\{\forall f\}\} \Leftrightarrow \exists g_n\{\exists f\{\forall f\}\}$$

Written in the new syntax (remember `∃f{∀f} <=> ∃f`):

$$f[g_{i \to n}] : \exists g_i\{\forall f\} \to \exists g_n\{\exists f\}$$

Demonstration

A very simple example that previously was not understood how to solve:

f[len] <=> add

len : [any] → nat // Length of a list
add : nat × nat → nat // Adds two natural numbers

∀f <=> $true_1$
∃len <=> $true_1$
∃add <=> $true_1$

Can there exists some function `f` that has the path `add` by `len`?

It is already known that a such function exists because:

concat[len] <=> add

However, assuming that we did not know about `concat`, how do we prove that is “can exist”?

∃f[$g_{i \to n}$]{∃g_i{∀f}} <=> ∃g_n{∃f{∀f}}	
∃add{∃len{∀f}, ∃len{∀f}} <=> ∃len{∃f{∀f}}	Inserting
∃add{∃len{$true_1$}, ∃len{$true_1$}} <=> ∃len{∃f{$true_1$}}	∀f <=> $true_1$
∃add{∃len, ∃len} <=> ∃len{∃f}	Reducing
∃add{$true_1$, $true_1$} <=> ∃len{∃f}	Inserting
∃add <=> ∃len{∃f}	Reducing
$true_1$ <=> ∃len{∃f}	

This is the criteria of the path to exist. Making a new assumption:

∃f <=> $true_1$

$true_1$ <=> ∃len{∃f}	
$true_1$ <=> ∃len{$true_1$}	Inserting
$true_1$ <=> ∃len	Reducing
$true_1$ <=> $true_1$	Inserting
Qed.	

This does not prove that `f` exists, but if this new assumption is correct for `f`, then it “can exist”.

One can imagine situations where proving non-existence of a path is useful without testing all inputs. In theory it is possible to prove non-existence if the two objects are found that collapses on `$g_i$` but not on `$g_n$`. Sometimes such two objects can be very hard to find. Having a way to prove non-existence without any expensive search is much more efficient.

This means one major goal of path semantics, to find an efficient algorithm that exhausts a large portion of path semantical space, might… just perhaps… be within reach. :)