Encoding Knowledge About Existential Paths as Functions

by Sven Nilsen, 2017

In this paper I represent an idea for dependently typed languages to reason about existential paths.

In path semantics the concept of existential paths is very important for type checking and theorem proving. Unfortunately it is very hard to find all existential paths, so any practical language using path semantics requires a way to supplement the compiler with new knowledge. Another problem is that one would like to write reusable code that works for future usage, given the necessary knowledge is acquired.

One idea of how to solve this problem, is to add a dependent type that contains the `∃` operator for functions. It is then possible to prove something assuming some new knowledge:

```
add : nat → nat → nat

check_addk : {G : nat → (nat → bool)} → ((k : nat) → ∃add(k) → G(k)) → ∃add(a) → G(a)
check_addk := \(f, f2) {a} = f(a, f2)

\(k : nat) → ∃add(k) → (>= k)              // Added knowledge

check_addk(\, ∃add(2)) : (>= 2)            // Looks for any function when using `\`
```

The program above type check because there exists some function that provides the proof. One way to type check is to insert every function by turn and see which fits. Notice that one does not need to insert `2` as an isolated argument, because it is extracted from `∃add(2)`.

Also, notice that function sub-types are objects on the type level and `:` is used instead of `==` to check for equality. All functions that use currying are introspective. When calling `check_addk(\, ∃add(2))` it is not possible to evaluate as a normal function, but it is possible to type check.

The `(>= k)` notation is shorthand for:

```
(>= k) := \(x) = x >= k
```

It is a way to declare a higher order function that binds a variable to the syntax.

Perhaps it is possible to generate such proofs automatically, such that one can just type:

```
∃add(2) : (>= 2)
```

This technique can also be used on problems that do not involve `∃`:

```
h : {G : nat → (nat → bool)} → ((k : nat) → (l : G(k)) → (>= k) → (>= x)) → (>= a)
        → (>= b : G(a))
h := \(f, f2) {a, b} = f(a, b, f2)

\(k : nat) → (x : (< k)) → (>= k) → (>= x)

h(\, (>= 2)) : (>= 1)
```