15–150: Principles of Functional Programming
Evaluation Trace for `csum`

Michael Erdmann

Spring 2024

In lecture, we saw the following code:

```
(* sum : int list -> int
   REQUIRES: true
   ENSURES:  sum(L) evaluates to the sum of all the elements in L.
*)
fun sum ([] : int list) : int = 0
  | sum (x::xs) = x + sum xs

(* csum : int list -> (int -> ’a) -> ’a
   REQUIRES: true
   ENSURES:  csum L k  ≅ k (sum L).
*)
fun csum ([] : int list) (k: int -> ’a) : ’a = k(0)
  | csum (x::xs) k = csum xs (fn s => k (x + s))
```

When writing CPS code, you should use the techniques you have already mastered: Think recursively/inductively, using the specs. Evaluation traces with continuations are generally too involved to write out in detail for long lists or large trees. Nonetheless, it is useful to know how to unravel function calls involving continuations, as a technique for debugging one’s code or proving correctness. Below is a trace of `csum` called on a very short list of integers, with a top-level continuation that turns its integer argument into a string.

```
  csum [2,3] Int.toString

≅ csum [3] (fn s => Int.toString (2 + s))

≅ csum [] (fn s’ => (fn s => Int.toString (2 + s)) (3 + s’))

≅ (fn s’ => (fn s => Int.toString (2 + s)) (3 + s’)) 0

≅ (fn s => Int.toString (2 + s)) (3 + 0)

≅ Int.toString (2 + 3)

≅ Int.toString 5

≅ "5"
```

So `csum [2,3] Int.toString` ↪ `"5"`, since `"5"` is an observable value.