CPSIterate Module

`smlhelp.github.io` – Auxiliary Library

Jacob Neumann

June/July 2021

Contents

1 Type Declarations

Type Spec

aux-library: CPSIterate.sml

```
datatype result = Accept
                | Keep
                | Discard
                | Break of string
```

The `result` type encodes possible instructions for how to proceed with a list iteration: either (1) terminate successfully with the current element, (2) keep the current element as part of ongoing accumulation, (3) discard the current element and proceed with iteration, or (4) terminate unsuccessfully with some error message.

Type Spec

aux-library: CPSIterate.sml

```
datatype command = Stop
                 | Continue
                 | Crash of string
```

The `command` type encodes possible instructions for how to proceed with an iterative process: either (1) terminate the iteration, (2) continue on to the next iteration, or (3) crash the process with some error message.

2 Main Definitions

Note 1

When we write a natural number (or integer) in math font (e.g. n), this refers to the mathematical integer n. When we use teletype font (e.g. `n`), this refers to the corresponding value of the SML type `int`. Likewise with the difference between, for instance, 3 and `3`.

Defn. 2 (Index)

Given a type `t` and a value `x : t`,

- For any value `xs : t list`, we say that `x` is **at index** 0 of `x::xs`
- For any values `xs : t list` and `x' : t`, if `x` is at index n of `xs`, then `x` is at index $n+1$ of `x'::xs`

Defn. 3 (List Iteration)

Given a type `t`, a total function `check : t -> result`, and a value `L : t list`, the **outcome of iterating `L` using `check`** is defined to be:

- **A success with element** `x` if `x:t` is an element of `L` such that `check x` $\Longrightarrow$ `Accept`, and, for any `x'` occurring before `x` in `L`, either `check x'` $\Longrightarrow$ `Keep` or `check x'` $\Longrightarrow$ `Discard`
- **A failure with message** `s` if `x:t` is an element of `L` such that `check x` $\Longrightarrow$ `Break s`, and, for any `x'` occurring before `x` in `L`, either `check x'` $\Longrightarrow$ `Keep` or `check x'` $\Longrightarrow$ `Discard`
- **An accumulation with sublist** `L'` if, for all `x` in `L`, either `check x` $\Longrightarrow$ `Keep` or `check x` $\Longrightarrow$ `Discard`, and

  `L'` $\cong$ `filter (fn x => (check x)=Keep) L`

3 Value Specifications

Value Spec

```
For : ('a -> result)
   -> ('a list)
   -> ('a -> 'b -> 'b)
   -> 'b
   -> ('a -> 'c)
   -> (string -> 'c)
   -> ('b -> 'c)
   -> 'c
```

REQUIRES: `check` is total, `combine x` is total for every `x`

ENSURES: `For check L combine base success panic return` ≅

`success(y)`	if the outcome of iterating `L` using `check` is a success with element `y`
`panic(s)`	if the outcome of iterating `L` using `check` is a failure with message `s`
`foldr (Fn.uncurry combine) base L'`	if the outcome of iterating `L` using `check` is an accumulation with sublist `L'`

4 `For` Correctness

4.1 Preliminaries

Defn. 4 (Terminator)

The values `Accept` and `Break(s)` (for any `s:string`) are called the **terminator** values of type `result`.

`Keep` and `Discard` are the non-terminator values.

Defn. 5

For the purposes of this document, we define the following specification function:

Value Spec

```
firstTerminator : ('a -> result)
                -> 'a list
                -> (int * 'a * result) option
```

REQUIRES: `check` is total

ENSURES: `firstTerminator check L` $\cong$

`NONE`	if `map check L` contains no terminators
`SOME(n,y,R)`	if `y` is the first element in `L` such that `check y` evaluates to some terminator `R`, and moreover `y` is at index `n` of `L`

```
fun firstTerminator check [] = NONE
  | firstTerminator check (x::xs) =
      case (check x) of
        Accept => SOME(0,x,Accept)
      | (Break s) => SOME(0,x,Break s)
      | _ => (case (firstTerminator check xs) of
                (SOME(n,y,R)) => SOME(n+1,y,R)
              | NONE => NONE)
```

Claim 6

Given `check` and `L` as in the spec of `For`:

- The outcome of iterating `L` using `check` is a success with element `y` if and only if

  `firstTerminator check L` $\cong$ `SOME(n,y,Accept)`

  for some `n`

- The outcome of iterating `L` using `check` is a failure with message `s` if and only if

  `firstTerminator check L` $\cong$ `SOME(n,y,Break s)`

  for some `y` and `n`.

- The outcome of iterating `L` using `check` is an accumulation with sublist `L'` (for some `L'`) if and only if

  `firstTerminator check L` $\cong$ `NONE`

Claim 7

If `firstTerminator check L` $\cong$ `SOME(n,y,R)`, then `y` is at index n of `L` and `check y` $\cong$ `R`

Claim 8

If `firstTerminator check (x::xs)` $\cong$ `SOME(n,y,R)` for some $n > 0$, then

`check x` is a non-terminator

Proof. If `check x` were a terminator `R`, then observe that `firstTerminator check (x::xs)` would evaluate to `SOME(0,y,R)`, contrary to our assumption that $n > 0$. ☐

Note 9

Henceforth, we refer to the helper function `run : t1 list -> (t2 -> t3) -> t3`, defined inside a `let` in the body of `For`. The types `t1,t2,t3` are determined by the arguments passed to `For`, which are in scope when evaluating `run`.

- `check : t1 -> result`
- `combine : t1 -> t2 -> t2`
- `base : t2`
- `success : t1 -> t3`
- `panic : string -> t3`

Whenever we state results about `run`, these are fixed in the background; any of these values not mentioned in the statement can be assumed to be arbitrary (though, following the spec of `For`, we assume `check` is total and `combine x` is total for all `x`). Note that, in addition to these, `For` also takes in `L : t1 list` and `return : t2 -> t3`. These values don't occur in the body of `run` (they're the arguments `run` is applied to), but they are technically in scope whenever `run` is being evaluated.

aux-library: CPSIterate.sml

```
    fun run ([] : 'a list) (k:'b -> 'c) : 'c =
          k base
      | run (x::xs) k =
          (case (check x) of
              Accept => success x
          |      Keep => run xs (k o (combine x))
          |   Discard => run xs k
          | (Break s)=> panic s)
```

4.2 `Accept`-Correctness of `For`

Lemma 10

If `firstTerminator check L` $\cong$ `SOME(n,y,Accept)`, then for all appropriately-typed `k`,

$$\texttt{run L k} \quad \cong \quad \texttt{success y}$$

Proof. We proceed by weak induction on n.

BC If $n = 0$, then by Claim 7, `y` is at index 0 of `L`, i.e. `L=y::xs` for some `xs`, and moreover

$$\texttt{check y} \Longrightarrow \texttt{Accept.}$$

So then, for any `k`,

$$\begin{aligned} \texttt{run (y::xs) k} &\Longrightarrow \texttt{case (check y) of ...} && \text{(defn run)} \\ &\Longrightarrow \texttt{case (Accept) of ...} && \text{(above)} \\ &\Longrightarrow \texttt{success y} && \text{(defn run)} \end{aligned}$$

Where the `...`'s indicate the rest of the body of `run`. So `run L k` $\cong$ `success y` as desired.

IH Suppose for some n that: if `firstTerminator check xs` $\cong$ `SOME(n,y,Accept)`, then for all appropriately-typed `g`,

$$\texttt{run xs g} \quad \cong \quad \texttt{success y}$$

Assume `firstTerminator check L` $\cong$ `SOME(n+1,y,Accept)`. Then `L` must be nonempty, so let `L=x::xs`. Then, since $n + 1 > 0$, by Claim 8 we must have that

`check x` is a non-terminator.

So either `check(x)` is `Keep` or `Discard`. The proofs of the claim in these two cases are almost identical, but we'll do the `Keep` case since it's a little bit more tricky. For arbitrary `k`,

$$\begin{aligned} \texttt{run (x::xs) k} &\Longrightarrow \texttt{case (check x) of ...} && \text{(defn run)} \\ &\Longrightarrow \texttt{case (Keep) of ...} && \text{(assumption)} \\ &\Longrightarrow \texttt{run xs (k o (combine x))} && \text{(defn run)} \\ &\cong \texttt{success y} && \textbf{IH} \end{aligned}$$

as desired. To see why **IH** is applicable in the last step, recall that:

(A) `check x` is a non-terminator

(B) `firstTerminator check (x::xs)` $\cong$ `SOME(n+1,y,Accept)`.

By (A) and the definition of `firstTerminator`,

```
firstTerminator check (x::xs)
⟹
(case (firstTerminator check xs) of
  (SOME(n,y,R)) => SOME(n+1,y,R)
 | NONE => NONE)
```

So we can see that if (B) is true, it must be the case that

`firstTerminator check xs` ≅ `SOME(n,y,Accept)`.

And thus the antecedent of the inductive hypothesis is satisfied, so, taking `g` to be `(k o (combine x))`, we have

`run xs (k o (combine x))` ≅ `success y`.

The case where `check(x)` ⟹ `Discard` is similar. □

Cor. 11 (Accept-correctness)

If the outcome of iterating `L` using `check` is a success with element `y`, then for all appropriately-typed `combine`, `base`, `success`, `panic`, and `return`,

`For check L combine base success panic return` ≅ `success y`

Proof. Use the first bullet point of Claim 6, and then by Lemma 10 get that

`run L return` ≅ `success y`

proving the claim. □

4.3 Break-Correctness of `For`

4.4 Accumulation-Correctness of `For`