Another Example of Futures

The following function, *partition*, will take a list and a data value (called *pivot*). *partition* will partition the list into two sublists:

(a) Those elements \( \leq \text{pivot} \)

(b) Those elements \( > \text{pivot} \)

(define (partition pivot L)
  (if (null? L)
      (cons () () )
    (let ((tail-part
            (partition pivot (cdr L))))
      (if (\(\leq\) (car L) pivot)
          (cons
           (cons (car L) (car tail-part))
           (cdr tail-part))
          (cons
           (car tail-part))
           (cons (car L) (cdr tail-part))))
    ) ) )
)
We want to add futures to partition, but where?

It makes sense to use a future when a computation may be lengthy and we may not need to use the value computed immediately.

What computation fits that pattern? The computation of `tail-part`. We’ll mark it in a blue box to show we plan to evaluate it using a future:
(define (partition pivot L)
  (if (null? L)
      (cons () () )
      (let ((tail-part
              (partition pivot (cdr L))))
        (if (<= (car L) pivot)
            (cons
             (cons (car L) (car tail-part))
             (cdr tail-part))
            (cons
             (car tail-part)
             (cons (car L) (cdr tail-part))))))
)

But this one change isn’t enough! We soon access the car and cdr of tail-part, which forces us to wait for its computation to complete. To avoid this delay, we can place the four references to car or cdr of tail-part into futures too:
(define (partition pivot L)
  (if (null? L)
      (cons () () )
      (let ((tail-part
          (partition pivot (cdr L))))
        (if (<= (car L) pivot)
            (cons
              (cons (car L) (car tail-part))
              (cdr tail-part))
            (cons
              (car tail-part))
            (cons (car L) (cdr tail-part))
          )
        )
      )
  ) ) ) )
Now we can build the initial part of the partitioned list (that involving pivot and (car L) independently of the recursive call of partition, which completes the rest of the list.

For example,

\[(\text{partition } 17 \ (5 \ 3 \ 8 \ ...))\]

creates a future (call it \text{future1}) to compute

\[(\text{partition } 17 \ (3 \ 8 \ ...))\]

It also creates \text{future2} to compute (car tail-part) and \text{future3} to compute (cdr tail-part). The call builds

\[\text{future2} \rightarrow \text{future3} \rightarrow 5\]
Reading Assignment

- Roosta: Section 13.3
- Introduction to Standard ML (linked from class web page)
- Webber: Chapters 5, 7, 9, 11
ML — Meta Language

SML is Standard ML, a popular ML variant.

ML is a functional language that is designed to be efficient and type-safe. It demonstrates that a functional language need not use Scheme’s odd syntax and need not bear the overhead of dynamic typing.

SML’s features and innovations include:

1. Strong, compile-time typing.
2. Automatic type inference rather than user-supplied type declarations.
3. Polymorphism, including “type variables.”
4. Pattern-directed Programming

```ocaml
fun len([]) = 0
| len(a::b) = 1+len(b);
```

5. Exceptions

6. First-class functions

7. Abstract Data Types

```ocaml
coin of int | bill of int |
check of string*real;
val dime = coin(10);
```

A good ML reference is

“Elements of ML Programming,”

by Jeffrey Ullman
(Prentice Hall, 1998)