15-150 Assignment 10

Karan Sikka
ksikka@andrew.cmu.edu
G
May 2, 2012

2.1: Pebbling 2DFS

Time Step	Pebble 1	Pebble 2
1	A	
2	B	C
3	D	E
4	H	I
5	K	L
6	N	F
7	Q	G
8	J	
9	M	
10	O	P
11	R	
12	S	

2.2: Pebbling 2BFS

Time Step	Pebble 1	Pebble 2
1	A	
2	B	C
3	D	E
4	F	G
5	H	I
6	J	K
7	L	M
8	N	O
9	P	Q
10	R	
11	S	

3.1:

Since `map` does stuff in parallel, the processor gets to decide the schedule in which it performs the operations. `incr` incorporates the following operations: (1) getting the value stored in `terminals` (2) adding 1 to the value and (3) storing the value back in the ref.

Say the ref has a value of 4, and two calls to `incr` need to be done. The processor may choose to do the following:

1. get the value from terminals (4)
2. get the value from terminals (4)
3. increment 1 to the one of the values (4 to 5)
4. increment 1 to the other value (4 to 5)
5. store the value in the ref (4 to 5)
6. store the other value in the ref (5 to 5)

At the end of these two parallel `incr` operations, the value of terminals has changed from 4 to 5. However, by the spec of incr, the value should have changed from 4 to 5 to 6. This is why different schedules will lead to different values for the integer.

3.2:

Yes it is benign in sequential contexts, because at the beginning of each sequential call, the value of 0 is stored in the ref, and during the call, only that call will affect the ref.

3.3:

No, it is not benign in parallel contexts. Multiple calls to `next_move` will alter the same reference `terminals`. Therefore the result of the value in `terminals` is affected by the other instances of `next_move`.

3.4:

Yes it is benign in sequential contexts, because each call has it's own ref, which is initialized to zero to start and is not affected by other sequential calls to `next_move`.

3.5:

Yes it is benign in parallel contexts. Parallel calls to `next_move` do not affect each other since each one has it's own copy of the ref `terminals`.

4.1:

Lemma 3:
For all `l : 'a list`, `r : 'a list`

```
revTwoPiles(l, r) = (rev l) @ r
```

The proof is by induction on `l`:

Base Case:
`l` ≅ `[]`

```
  revTwoPiles([], r)
≅case [] of [] => r | ...                          step
≅r                                                 step
≅[] @ r                                            Lemma 1
≅(case [] of [] => [] | ...)  @ r                  rev step
≅(rev l) @ [z]                                     rev step
```

Inductive Hypothesis:
`l` ≅ `x::xs`
Assume `revTwoPiles(xs, r)` ≅ `(rev xs) @ r`

Inductive Step:
`l` ≅ `x::xs`
WTS: `revTwoPiles(x::xs, r)` ≅ `(rev x::xs) @ r`

```
  revTwoPiles(x::xs, r)
≅case ...  | (x::xs) => revTwoPiles(xs , x::r)                    step
≅revTwoPiles(xs , x::r)                                           step
≅(rev xs) @ (x::r)                                                by the IH
≅(rev xs) @ (x::([] @ r))                                         by Lemma 1
≅(rev xs) @ (case x::[] of ...  | x::[] => x::([] @ r))           rev step from @
≅(rev xs) @ (x::[] @ r)                                           rev step from @
≅(rev xs) @ ([x] @ r)                                             x cons nil is [x]
≅((rev xs) @ [x]) @ r                                             Lemma 1
≅(case x::xs of ...  | x ::  xs => (rev xs) @ [x]) @ r            rev step from rev
≅(rev x::xs) @ r                                                  rev step from rev
```

4.2:

Theorem 1: For all values `l : 'a list`

```
rev l = rev2 l
```

Proof:
Consider:

```
rev2 l
≅let ...  in revTwoPiles(l , []) end          step
≅revTwoPiles(l, [])                           step
```

≅`(rev l) @ []` Lemma 3
≅`rev l` Lemma 1

5.3:

In the case that the key is not in the dictionary, PoorMemoizer will use the function passed in to compute the result. However, the function passed in makes recursive calls, those calls will not be memoized. In the case of computing the nth fibonacci number where n is not in the dictionary, the recursive function for fibonacci will call f(n-1) and f(n-2), without checking if (n-2) or (n-1) are in the dictionary.

5.5:

The effects will not occur if arguments passed to the function are already in the dictionary. If you try to memoize a function which utilizes the print function, effects will only occur if you pass in a key which is not already in the dictionary. If you pass a value which is already in the dictionary (or if the function does so recursively), the memoized function will recognize that and will immediately return the function's previously returned value. In the case of the print function, the printing will not occur but the function will still return `()`.