Homework 5

Handout 15
CS242: Autumn 2002
5 November

Solutions

1. Exceptions

(a) There are two cases to consider. If `x` does not appear in `t`, then it is easy to see that both functions return the same result. On the other hand, if `x` is the value of one or more leaves of `t`, then the first function will return `x`, since this will be the closest value to `x` in the tree. Since the second function will return `x` by raising an exception, the two functions return the same result.

(b) If some leaf has value `x`, then the second function will raise an exception, transferring control to the handler and aborting the computation of `cls`. As a consequence, the function will only search the tree until the leaf with value `x` is found. The first function, on the other hand, computes the distance from each leaf to `x`. If the leaf with value `x` appears early in the depth-first traversal of the tree, then the second version will traverse much less of the tree and may therefore be more efficient. This efficiency must be balanced, of course, against the cost of testing each leaf to see if the exception should be raised. In other words, when the value x does not occur in t, the second function will be less efficient since it does an extra test on each leaf. Therefore, the choice between these two functions will depend on whether we expect the value we are searching for to appear or not. The point of the example, however, is to illustrate how exceptions could be used to terminate a computation for efficiency. If we were doing more computation at each leaf in the tree, such as might be necessary when searching for a value with more complicated property, then the cost of one equality test per leaf would be negligible and we would generally prefer the second form of search.

2. Exceptions and Recursion

call f(7)
call f(5)
call f(3)
call f(1)
raise exception Odd
pop activation record of f(1) off stack
pop activation record of f(3) off stack without returning control to f(3)
handle exception Odd in f(5)
return -5 from f(5)
return -5 from f(7)
return -5 from f(9)
return -5 from f(11)

3. Tail Recursion and Exception Handling

Tail-recursion elimination depends on being able to reuse the same activation record for each call to the function. However, if an exception-handling mechanism expects to be able to pop activation records off the stack until a handler is found, ordinary tail-recursion elimination may not work for functions that establish handlers.

For this function in particular, a smart compiler could apply tail-recursion-elimination to `f(x,count)` because every call will use the same handler and the handler does not access global variables. In

this case, there would be two forms of activiation record, one for calls that establish a handler and one for calls that do not. In other words, on a call to `f`, we would use unoptimized calls for `f(1,count)` and `f(0,count)`, but optimize tail calls from `f(x,count)` to `f(x-1,count+1)` when $x - 2 > 1$.

Another way to enable tail-recursion elimination is to store pointers to exception handlers in some data structure separate from the run-time stack. This is actually the approach used in many compilers, independent of whether tail-recursion elimination is performed.

4. ..... Evaluation Order and Exceptions

```
let fun f(x,y) = x+y in f(raise E(1), raise E(2)) end handle E(x) => x;
```

5. ..... Control Flow and Memory Management

If a function `main` calls a function `savealldata`, and `savealldata` allocates memory for local use and then calls `openfile`, and after `openfile` returns, `savealldata` frees the memory and returns to `main`, then we may see a memory leak in the presence of exceptions. Suppose that `openfile` raises an exception, and `main` handles it. When `openfile` raises the exception, control is transferred to `main`, and `savealldata` does not have an opportunity to free the memory. Since the memory was for local use by `savealldata`, it was referenced only by `savealldata`'s local variables. These variables are no longer available, so the memory cannot be reached by the program. Thus we have a memory leak.

6. ..... Tail Recursion and Continuations

(a) In a tail recursive function, the return is the same as the value of the recursive call, meaning that no work is done after the recursive call is invoked. We do not need to keep around the information in the activation record of the function once the recursive call is made if we will not use that information again. An optimizing compiler can reuse the space allocated for the activation record for the next recursive call. This means that recursive function will always use a constant amount of space

(b) The closure for the function created in f has an environment pointer that points to the activation record of the current call to f. Therefore, the activation record for f cannot be reused for the recursive call since the information stored in that activation record may be used in the future by the newly created function. Also, at the end, $g(1)$ is called, and g will be an $n-$level function built up during the calls to f. This function isn't tail recursive, so it cannot be optimized, and it also uses n activation records. Thus, the amount of space needed for the recursion is $O(n)$.

7. ..... Continuations

(a) Looking at the two program fragments, 2 cases need to be considered, one when the exception is raised and one when the exception is not raised:

- `y < 0`: For the exception style program, the exception `Too_Small` is raised. This exception is handled by the handler defined at the last line and the result is 0. Notice that in that case, `f` will never return and the addition is never be performed. For the continuation passing style program, `f` returns the result of applying `(fn () => 0)` to `()`, namely 0.
- `y > 0`: For the exception style program, `f` returns `y/2`, which is added to 1 at the last line. The result of the program is `1 + y/2`. For the continuation passing style program, `f` returns the result of the application of `fn z => 1+z` to `y/2` namely `1 + y/2`.

In both cases these two programs return the same value.

(b) In continuation passing style, functions don't return before the program ends. If tail call optimization is not performed, activation records will pile up on the stack, and it is more likely that the program will run out of stack space.