Tail Recursion

COMP 320

12 TAIL RECURSION

Recursion and Tail Recursion

Recursion has been used in our programs in place of loops

It is much more elegant when processing recursive datatypes (lists, trees, expressions)

There may be efficiency considerations when using recursion.

These can generally be overcome by using tail recursion

We will look at the use of an accumulator to achieve tail recursion

This idea generalizes and leads to an continuation passing style of programming

Factorial – Recursive Version

A standard recursive definition of factorial would be:

```
fun fact n = if n=0 then 1 else n * fact(n-1)
```

If we evaluate this using substitution, we see the following pattern:

Factorial – Recursive Version

```
fact 4
      |->* 4 * fact(3)
      |->* 4 * 3 * fact(2)
      |->* 4 * 3 * 2 * fact(1)
      |->* 4 * 3 * 2 * 1 * fact(0)
      |->* 4 * 3 * 2 * 1 * 1
      |->* ..
      |->* 24
```

Evaluation Stack

In order to get a better understanding of how to perform this evaluation, we have to think about how function calls are implemented

Conceptually, there is a stack in which each object stores information about the function that has been called but not completed

The objects on the stack don't concern us in this course but they would typically store things like the values of parameters and local variable, the point at which the function was called and must return to, etc

When a function is called an object (a "stack frame") is pushed onto the stack

When the function returns the stack is popped

For the factorial function the maximum size of the stack is linear in the argument n

Another Version of Factorial

Another version of factorial which seems more complicated:

```
fun fact n =
    let fun factTC (n, acc) =
                              if n=0 then acc else factTC(n-1,acc*n)
    in
      factTC (n, 1)
    end
```

This version is tail-recursive.

The recursive call occurs at the end of each invocation and nothing is done afterwards.

Local Definitions (an aside)

Let expression have the form `let decl in exp end`

If we just want to have local declarations in a declaration we can use

```
          local decl in decl end
```

```
local fact n =
    fun factTC (n, acc) = if n=0 then acc else factTC(n-1,acc*n)
    in
     fun fact = factTC (n, 1)
    end
```

Evaluation with the tail recursive version

```
fact 4
      |->* factTR(4,1)
      |->* factTR(3,4)
      |->* factTR(2,12)
      |->* factTR(1,24)
      |->* factTR(0,24)
      |->* 24
```

There is no need to maintain a call stack

Why Tail Recursion?

Implementations pf SML and many other functional languages include this optimization for tail recursive functions

They pop the calling function before the call since it is finished when the called function is finished

This provided us with a methodology for improving efficiency – try and use tail recursive calls

Code may be a bit more more complicated but much more efficient

How can we make a recursive function tail-recursive?

- Create a helper function that has an accumulator as argument
- the old base case is the initial value of the accumulator
- the new base case is the final value of the accumulator

What is tail recursion

Tail recursion occurs when a recursive call is made in **tail position**

Cases:

- `fun f(x) = e`, e is in tail position
- `if e1 then e2 else e3`, e2 and e3 are in tail position (not e1)
- `let b1, …, bn in e end`, only e is in tail position
- `f(e1, …, en)`, none of the expressions are in tail postion
- If an expression is not in tail position, then neither are its arguments

Sum the values in a list

```
(* not tail recursive *)
fun sum lst =
    case lst of
      [] => 0
    | x::xs => x + sum1 xs
```

Sum the values in a list

Both take linear time but this one takes less space

```
(* tail recursive *)
fun sum lst =
    let fun sumTR (lst, acc) =
          case lst of
             [] => acc
           | x::xs => sumTR(xs, x+acc)
    in
      sumTR(lst,0)
    end
```

Another Example

```
(* not tail recursive *)
fun rev lst =
   case lst of
        [] => []
     | x::xs => (rev xs) @ [x]
```

We have seen that this takes quadratic time

Another Example

Using tail recursion give a linear time algorithm

```
(* tail recursive  *)
fun rev lst =
    let fun revTR(lst,acc) =
          case lst of
             [] => acc
           | x::xs => revTR(xs, x::acc)
    in
      revTR(lst,[])
    end
```

Continuation Passing Style

CPS has several applications

One use is to reduce the stack space used by a program. This does not always reduce the total space requirements but in traditional implementations of many languages (C, Java, etc) there is a distinction made between stack space (used for function calls) and heap space (used for other data)

Too many function calls might cause a program to run out of stack space and crash

Continuations are also useful for backtracking algorithmsN

Sum With Accumulator

For sum we were able to use an accumulator to reduce the usage of the stack

```
fun sumTR (lst, acc) =
      case lst of
        [] => acc
      | x::xs => sumTR(xs, x+acc)
```

This took knowledge of the particular function

CPS gives us a generic way to achieve this effect

Continuations

In general, suppose we have a function `f: 'a -> 'b` and we would like to rewrite it as a function f' in tail-recursive form

We give f' an additional argument called a continuation

The continuation specifies what we want to do with the result of f once we get a result from the recursive call.

In the base case, instead of returning a result we call the continuation.

In the recursive case we tell the continuation what should be done to the result

The type of `f' : 'a -> ('c -> 'b) -> 'b`

When we use f' to compute f, we must specify an initial continuation which is often the identity function (meaning we don't want to process the final result any further)

Sum with continuation

We use an extra functional argument to represent the part of the computation that still has to be done

This extra argument, `k : int -> int`, is called a continuation

```
fun sumCPS lst k =
        case lst of
          [] => k 0
        | x::xs => sumCPS xs (fn s => k (x+s))
```

Evaluation

Calling sumCPS starting with the identity function:

```
sumCPS [1,2] (fn x => x)
|->* sumCPS [2] (fn s => (fn x => x) (1 + s))
|->* sumCPS [] (fn s' => (fn s => (fn x => x) (1 + s)) (2 + s'))
|->* (fn s' => (fn s => (fn x => x) (1 + s)) (2 + s')) 0
|->* (fn s => (fn x => x) (1 + s)) (2 + 0)
|->* (fn s => (fn x => x) (1 + s)) 2
|->* (fn x => x) (1 + 2)
|->* (fn x => x) 3
|->* 3
```

Simplification

First, note that in this evaluation we do not evaluate the continuation arguments until the end

Why? Because functions are values and only function application forces evaluation

It is easier to read the evaluation trace if we use function equivalence

Functions are equivalent if they give the same results for all arguments

Then by referential transparency we can replace a function by an equivalent function

An equivalent evaluation

```
sumCPS [1,2] (fn x => x)
== sumCPS [2] (fn s => (fn x => x) (1 + s))
== sumCPS [2] (fn s => (1 + s))
== sumCPS [] (fn s' => (fn s => (1 + s)) (2 + s'))
== sumCPS [] (fn s' => (1 + (2 + s')))
== (fn s' => (1 + (2 + s'))) 0
== (1 + (2 + 0))
== 3
```

Comments

Instead of a stack of recursive calls we have a stack of functions so we have not necessarily saved space but we have transferred the storage from the run-time stack to other storage for the closures of the continuations

In SML/NJ sum is automatically transformed into sumCPS to avoid distinguishing between different types of storage (stack vs heap)

We will look at more examples of this continuation passing style

It is useful for backtracking algorithms

It is useful (although we won't look at it here) in web programming. Programs that interact over HTTP are inherently written if CPS because the server program must terminate every time it passes control to the browser. If the compiler transforms a program into a CPS program simplifies web programming

Proving Correctness

Before we move onto more examples, we will prove the correctness of sumCPS

Claim: For every `lst : int lst, sumCPS lst (fn x => x) == sum lst`

Proof by induction

Proving Correctness

Before we move onto more examples, we will prove the correctness of sumCPS

Claim: For every `lst : int lst, sumCPS lst (fn x => x) == sum lst`

Proving Correctness

Proof failed.

What next? Generalize!

Claim: For every `lst : int lst, and for any k: int -> int,`

```
sumCPS lst k == k (sum lst)
```

This time, as long as we are careful to formulate the IH properly, the proof works