PLD Assignment 2

Ask

4. marts 2022

Indhold

1 Garbage collection

So the idea would be to use linked lists in order to allocate the numbers. One of the reason to use linked lists, is that "arrays"in my case the heap has a fized size in C. Linked lists also have the benefits of dynamic size and the ease of inserting/delition from compared to arrays.

Function to insert a node at the end of a linked list.

```
void addLast(struct node **head, int val)
{
    //create a new node
    struct node *newNode = malloc(sizeof(struct node));
    newNode->data = val;
    newNode->next     = NULL;

    //if head is NULL, it is an empty list
    if(*head == NULL)
            *head = newNode;
    //Otherwise, find the last node and add the newNode
    else
    {
        struct node *lastNode = *head;

        //last node's next address will be NULL.
        while(lastNode->next != NULL)
        {
            lastNode = lastNode->next;
        }

        //add the newNode at the end of the linked list
        lastNode->next = newNode;
    }

}
```

Function to iterate the linked list and print the data.

```
void printList(struct node *head)
{
    struct node *temp = head;

    //iterate the entire linked list and print the data
    while(temp != NULL)
    {
```

```
            printf("%d->", temp->data);
            temp = temp->next;
    }
    printf("NULL\n");
}
```

This is a function that could be used to remove a node when it is no longer in use in order to free up space in the heap.

```
Node* removeNode(int value, Node* head) {
    Node* newHead = head;
    if (head->value == value) {
        newHead = head->next;
        delete head;
    }

    while(head->next != nullptr && head->next->value != value)
    head = head->next;
    if(head->next) {
    Node* temp = head->next;
    head->next = head->next->next;
    delete temp;
    }
    return newHead;
}
```

2 Parameter passing mechanisms

2.1 a)

Since we're only asked about what happens in this general case, my answer to this question is a more general answer. I'm not doing the actual calculations of transpose(q,q) In this case, the function is given the same parameter of the matrix q, hence "transpose(q,q). This means, that the parameters are stored in two difference local variables in the callee (transpose). If these local variables have different values when the call returns, there can be an ambiguity in which value is copied back to the variable. However in this case, I know that it is copied (written) back from left to right. So the "a"parameter will be copied back first. Overwriting the original value of the variable.

2.2 b)

Call-by-Reference can introduce "aliasing". Meaning that if a function is given the same variable to/as two different reference parameters. Those two parameters (variables in the callee) can refer to the same location. Meaning that updating one of the variable also updates the other one.
Aliasing hinders optimisation.

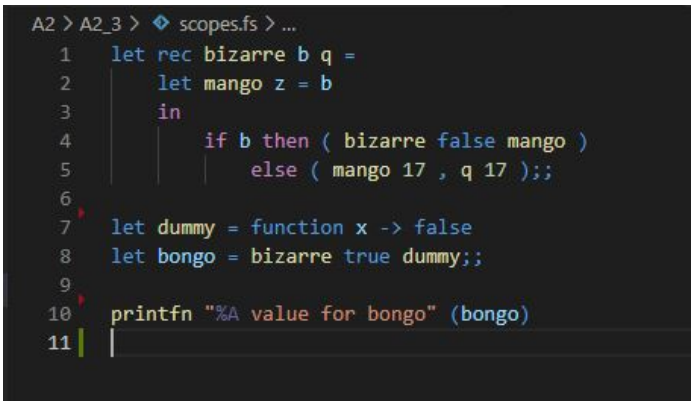
2.3 c)

If a compiler fetches the content of a reference parameter into a register, and then write to another reference parameter. The programmer should make sure that the compiler re-fetch the first parameter, because the write might have affected this. This would occur if the parameters are aliases of the same location.

essentially when using pass-by-reference parameters, if a function causes changes to the variables in the caller these changes would be reflected in the callee, since original values would have been overwritten.

3 Scope sules

Yes it is possible to describe it this way. I've modified the program slightly to display the result of



Figur 1: Modified F# code from assignment text

The result I get in my terminal when running and printing the value of "bongo"is bongo is = (false, true).

in the body of bongo we have the recursive function bizarre, which get instansiated with the parameters "true"and "false"since the function dummy will always set the other parameter to false.

going through the function "bizarre"we can see that the value of b is passed down until we enter the if statement, where bizarre will be called recursivly and the values for b and q will be updated with the new value for "b"and the old value for mango which is "true".

4 Program transformations

I'll be using the algorithm from wikipedia page on the topic λ-lifting. I personally just find it easier to follow since the steps are more clearly discribed compared to how it was described in our slides.

The algorithm is as following on the wikipedia page:

1. Rename the functions so that each function has a unique name.
2. Replace each free variable with an additional argument to the enclosing function, and pass that argument to every use of the function.
3. Replace every local function definition that has no free variables with an identical global function.
4. Repeat steps 2 and 3 until all free variables and local functions are eliminated.

ORIGINAL CODE SNIPPET:

```
let rec bizarre b q =
    let mango z = b
    in
        if b then ( bizarre false mango )
            else ( mango 17 , q 17 );;

let dummy = function x -> false
let bongo = bizarre true dummy;;
```

STEP 1, it is possible to skip step one of the algorithm, since all functions are "unique"in the sence that there's no dublicates of functionnames that are considered another function than the original.
STEP 2) the function named "mango"has a free variable b. So this variable has to be replaced and passed to every use of the function

I've chosen to replace variable name "b"with "mango_b"and passed that argument to every other use of the function "mango".

```
let rec bizarre b q =
    let mango z mango_b = mango_b
    in
        if b then ( bizarre false (mango z mango_b) )
            else ( mango 17 mango_b , q 17 );;

let dummy = function x -> false
let bongo = bizarre true dummy;;
```

STEP 3) Is to lift the function(s) to a global scope.
REPLACING mango to mango'

```
let rec mango z mango_b = mango_b
and bizarre b q =
    if b then ( bizarre false (mango' z b))
        else ( mango' 17 b, q 17 );;

let dummy = function x -> false
let bongo = bizarre true dummy;;

mango' z mango_b = mango_b
```

FINAL Program after lambda lifting

```
let rec bizarre' b q =
    if b then ( bizarre false (mango' z b))
        else ( mango' 17 b, q 17 );;

let dummy = function x -> false
let bongo = bizarre true dummy;;
mango' z mango_b = mango_b
```

5 Control structures

In our book, Prolang theres a specific line that I've been thinking about and trying to find more information about. It says *The ALGOL 60 report does not forbid the loop body to modify i*[1] If it is possible for the body of a loop to modify the counter, it could be possible to end ud with an infinite loop. By either resetting the value of the counter inside the body of the loop, in which case the loop would never terminate since a desired condition might never be reached.

[1] p. 138, Prolang