Checking Tail Recursion in PicoML

Umang Mathur
umathur3@illinois.edu

Chia-Hao Hsieh
chsieh17@illinois.edu

December 16, 2015

1 Overview

1.1 Recursion

Allowing procedures to be recursive allows the programmer to write more readable and intuitive/natural programs. Recursive programs, at times, tend to be more efficient than a naive program with loops and no recursive calls. Recursion, thus is a handy tool for programmers.

1.2 Checking Tail Reclusion : Motivation

The convenience offered comes at a cost. Recursive programs are generally modelled by the use of stack frames. This means that recursive programs tend to consume extra space (stack) for every recursive call they make.

However, the extra space consumed can be overcome when the recursive call is the last thing the function does. In this case, the contents of the stack can be replaced by the new frame, and there is no need to push an additional frame.

The idea behind tail call optimization is essentially the same. Informally, a recursive function is tail recursive when the recursive call is the last thing executed by the function. Thus, if the compiler can detect if a function is tail-recursive, it can convert the function to an equivalent while-loop, thus avoiding an additional call that consumes extra stack space by virtue of the new frame added.

1.3 Goal of the Project

In this project, we implement a tool that checks if an procedure is tail recursive or not. Specifically, we wish to analyze code written in PicoML. PicoML is a restricted form of OCaml. As part of the assignments in the course, we have built an interpreter for this language. We aim to integrate the tool with the interpreter. That is, we would use the parsing and the type checking functionality written in older assignments. This would enable use to directly use the functionality for implementing our tool, and would save some effort, as compared to the scenario where we had to re-invent the wheel.

2 Implementation

We built two tail recursive checking programs. One for normal functions is based on MP6 while the other for CPS functions is based on MP7.

2.1 Tail Recursive Checking Based on MP6

Here is our proposal:

2.2 Tail Recursive Checking for CPS Based on MP7

Here is our proposal:

2.3 Code Structure

Describe the code structure. Justify that the code is modular

3 Tests

[TODO]We put our test cases into grader of MP6 and MP7. So to test our programs, just run './grader' after 'make', as in what we have to do in the assignments.

4 Listing

4.1 Direct stype PicoML expressions

```
open Definitions;;

let rec check_let_in_meaningful x e =
    match e
    with ConstExp c -> false
    | VarExp v -> if (v = x) then true else false
    | MonOpAppExp (mon_op, e1) -> check_let_in_meaningful x e1
    | BinOpAppExp (bin_op, e1, e2) -> (check_let_in_meaningful x e1) || (check_let_in_meaningful
    x e2)
    | IfExp (e1, e2, e3) ->
        (check_let_in_meaningful x e1) || (check_let_in_meaningful x e2)  || (
    check_let_in_meaningful x e3)
    | LetInExp (s, e1, e2) ->
        if (check_let_in_meaningful x e1)
            then (check_let_in_meaningful s e2)
            else (
                if (x=s)
                    then false
                    else check_let_in_meaningful x e2
                )
    | FunExp (s, e1) -> if (s=x) then false else (check_let_in_meaningful x e1)
    | AppExp (e1, e2) ->
        (check_let_in_meaningful x e1) || (check_let_in_meaningful x e2)
    | LetRecInExp (g, y, e1, e2) ->
        if ((g=x) || (y=x))
            then false
            else if (check_let_in_meaningful x e1)
                then (check_rec_f g e2)
                else (check_let_in_meaningful x e2)
    | RaiseExp e1 -> (check_let_in_meaningful x e1)
    | TryWithExp (e0, n1opt, e1, nopt_e_lst) ->
        (check_let_in_meaningful x e0) || (check_let_in_meaningful_lst x ((n1opt,e1)::nopt_e_lst)
     )
and check_let_in_meaningful_lst x nopt_e_lst =
    match nopt_e_lst
    with [] -> false
    | (nopt, en)::rest -> (check_let_in_meaningful x en) || (check_let_in_meaningful_lst x rest)
and check_rec_f f e =
    match e
    with ConstExp c -> false
    | VarExp v -> false
    | MonOpAppExp (mon_op, e1) -> check_rec_f f e1
    | BinOpAppExp (bin_op, e1, e2) -> (check_rec_f f e1) || (check_rec_f f e2)
    | IfExp (e1, e2, e3) ->
        (check_rec_f f e1) || (check_rec_f f e2) || (check_rec_f f e3)
    | LetInExp (s, e1, e2) ->
        if (check_rec_f f e1)
            then (check_let_in_meaningful s e2)
            else
                (
                if (s=f) then false else ( (check_rec_f f e1) || (check_rec_f f e2) )
                )
    | FunExp (s, e1) -> if (s=f) then false else (check_rec_f f e1)
    | AppExp (e1, e2) ->
        (
        match e1
        with VarExp v -> if (v=f) then true else false
        | _ -> (check_rec_f f e1) || (check_rec_f f e2)
        )
    | LetRecInExp (g, x, e1, e2) ->
        if ( (g=f) || (x=f))
```

```
            then false
            else if (check_rec_f f e1)
                then (check_rec_f g e2)
                else (check_rec_f f e2)
    | RaiseExp e1 -> (check_rec_f f e1)
    | TryWithExp (e0, n1opt, e1, nopt_e_lst) ->
       (check_rec_f f e0) || (check_rec_f_lst f ((n1opt, e1) :: nopt_e_lst) )

and check_rec_f_lst f nopt_e_lst =
    match nopt_e_lst
    with [] -> true
    | ((nnopt, en)::rest) -> ((check_rec_f f en) || (check_rec_f_lst f rest));;

let rec check_tail_rec_f f e =
    match e
    with ConstExp c -> true
    | VarExp v -> true
    | MonOpAppExp (mon_op, e1) -> not (check_rec_f f e1)
    | BinOpAppExp (bin_op, e1, e2) -> (not (check_rec_f f e1)) && (not (check_rec_f f e2))
    | AppExp(e1, e2) -> if (check_rec_f f e2)
       then false
       else check_tail_rec_f f e1
    | IfExp (e1, e2, e3) ->
            (not (check_rec_f f e1)) &&
            (check_tail_rec_f f e2) &&
            (check_tail_rec_f f e3)
    | FunExp (x, e1) -> true
    | LetInExp (x, e1, e2) ->
       if (x = f)
           then (not (check_rec_f f e1))
           else ( (not (check_rec_f f e1)) && (check_tail_rec_f f e2))
    | LetRecInExp (g, x, e1, e2) ->
       if (g = f) then true
       else if (not (g=f) && (x=f)) then (check_tail_rec_f f e2)
       else ( (check_tail_rec_f f e2))
    (* Before fix:
       if (g = f) then true
       else if (not (g=f) && (x=f)) then (check_tail_rec_f f e2)
       else ( (not (check_rec_f f e1)) && (check_tail_rec_f f e2))
    *)
    | TryWithExp (e', n1opt, e1, nopt_e_lst) ->
       (
       if (check_rec_f f e')
           then false
           else let lst = ((n1opt, e1)::nopt_e_lst)
               in
               (List.fold_right (fun (intop, h) -> fun t -> (check_tail_rec_f f h) && t) lst
    true)
       )
    | _ -> false ;;


let check_tail_recursion dec =
    match dec
    with (Anon e) -> true
    | Let (s, e) -> true
    | LetRec (f, x, e) ->
       check_tail_rec_f f e ;;
```

Listing 1: Tail recursion for PicoML expressions

4.2 CPS transformed PicoML expressions

```
1
2  open Definitions
3
4  let rec convert_f_exp e name_of_f =
5      (
6      match e
7      with  ConstCPS (k, c) ->
8          (*
9           print_string "\nConstCPS \n";
10         *)
11         convert_f_cont k name_of_f
12     | VarCPS (k, g) ->
13         (*
14         print_string "\nVarCPS \n";
15         *)
16             (
17             match k
18             with FnContCPS (number_of_f, e') ->
19                 let rec_list = convert_f_exp e' name_of_f
20                 in
21                 if (g = name_of_f)
22                         then (number_of_f :: rec_list)
23                         else rec_list
24             | _ -> []
25             )
26     | MonOpAppCPS (k, _, _, _) ->
27         (*
28         print_string "\nMonOpAppCPS \n";
29         *)
30         convert_f_cont k name_of_f
31     | BinOpAppCPS (k , _, _, _, _) ->
32         (*
33         print_string "\nBinOpAppCPS \n";
34         *)
35         convert_f_cont k name_of_f
36     | IfCPS (b, e1, e2) ->
37         (*
38         print_string "\nIfCPS \n";
39         *)
40         (convert_f_exp e1 name_of_f)@(convert_f_exp e2 name_of_f)
41     | AppCPS (k, _, _, _) ->
42         (*
43         print_string "\nAppCPS \n";
44         *)
45         convert_f_cont k name_of_f
46     | FunCPS (k, _, _, _, _) ->
47         (*
48         print_string "\nFunCPS \n";
49         *)
50         convert_f_cont k name_of_f
51     | FixCPS (k, _, _, _, _, _) ->
52         (*
53         print_string "\nFixCPS \n";
54         *)
55         convert_f_cont k name_of_f
56     )
57
58 and
59
60 convert_f_cont k name_of_f =
61     (
62     match k
63     with FnContCPS (_, e') ->
```

```
        convert_f_exp e' name_of_f
    | _ -> []
    ) ;;


let rec check_cps_tail_rec_f flist original_k x k e =
    match e
    with ConstCPS (k', c) ->
        cont_tail_recursive original_k k' flist
    | VarCPS (k', v) ->
        (*
        print_string "\nVarCPS\n";
        *)
        cont_tail_recursive original_k k' flist
    | MonOpAppCPS (k', mono_op, o1, exk) ->
        (*
        print_string "\nMonOpAppCPS\n";
        *)
        cont_tail_recursive original_k k' flist
    | BinOpAppCPS (k', bin_op, o1, o2, exk) ->
        (*
        print_string "BinOp\n";
        *)
        cont_tail_recursive original_k k' flist
    | IfCPS (b, e1, e2) ->
        (check_cps_tail_rec_f flist original_k x k e1) && (check_cps_tail_rec_f flist original_k
    x k e2)
    | AppCPS (k', e1, e2, exk) ->
        (
        (*
        print_string ("AppCPS:\n"^e1^", "^e2^", f: "^f^"\n");
        *)
        if ( List.exists (fun x -> x = e1) flist)
            then
                (
                if (k'=original_k)
                    then
                        (*
                        print_string "true\n";
                        *)
                        true
                     else
                         (*
                         print_string "false\n";
                         *)
                         false
                )
            else
                cont_tail_recursive original_k k' flist
        )
    | FunCPS (kappa, x, k, ek, e) -> true
    | FixCPS (kappa, f, x, k, ek, e) -> true

and cont_tail_recursive original_k k flist =
    match k
    with ContVarCPS i -> true
    | External -> true
    | FnContCPS (x, e) ->
        check_cps_tail_rec_f flist original_k x k e (*TODO x ?*)
    | ExnMatch ek -> true ;;


let check_tail_recursion dec =
    match dec
    with Anon e -> true
```

```
    | Let (x,e) -> true
    | LetRec (f,x,e) ->
        let (i,j) = (next_index(),next_index())
        in
            let ecps2 = cps_exp e (ContVarCPS i) (ExnContVarCPS j)
            in
            (*
            print_string ((string_of_exp_cps ecps2)^"\n");
            *)
            let flist = convert_f_exp ecps2 f
                in
                check_cps_tail_rec_f flist (ContVarCPS i) x (ContVarCPS i) ecps2 ;;
```

Listing 2: Tail recursion for CPS transformed expressions

References

[1] Leslie Lamport, *LaTeX: a document preparation system*, Addison Wesley, Massachusetts, 2nd edition, 1994.