Checking Tail Recursion in PicoML

Umang Mathur
umathur3@illinois.edu

Chia-Hao Hsieh
chsieh17@illinois.edu

December 17, 2015

1 Overview

1.1 Recursion

The use of recursion dates back to the late 19^{th} century, when mathematicians Dedekind and Peano used induction to defined functions. The use of recursion played an important role in foundations of computer science, and was later referred to as 'primitive recursion' [1]

Use of recursion is not just exciting from the perspective of a Mathematician, but is also quite significant from the perspective of a developer. Allowing procedures to be recursive helps the programmer write more readable and intuitive/natural programs. A notable use of recursion is seen when dealing with inductive structures. Inductive definitions and inductive programs can be very naturally programmed as recursive functions. Besides, recursive functions, can be easier to debug, due to the same reason. Recursive programs, at times, tend to be more efficient than a naive program with loops and no recursive calls. Recursion, thus is a very handy tool for programmers.

1.2 Checking Tail Reclusion : Motivation

The convenience offered due to recursion, 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. Besides, the additional overhead of copying the variables and values to the new frame, also accounts for a non trivial overhead, which at times, is not desirable from the standpoint of efficiency.

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 **`TailRec`** that checks if a procedure is tail recursive or not. Specifically, we wish to analyze declarations written in PicoML. PicoML is a restricted form of OCaml, and supports simple expressions like `if then else`, `fun`, `letrec`. As part of the assignments in the course, we have built an interpreter for this language [2]. We aim to integrate the **`TailRec`** 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 **`TailRec`** , and would save some effort, as compared to the scenario where we had to re-invent the wheel.

2 Definitions

Before we describe our implementation, it would be useful to go through some notations and definitions. The purpose of the definitions is to give a nice characterization of the problem we wish to address.

Definition 2.1 (PicoML Expression)**.** A PicoML-style expression e is formally defined by the recursive grammar:

$$\begin{aligned} e := c \mid v \mid\ \odot e \mid e \oplus e \mid \texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e \mid e\ e \mid \texttt{fun}\ x\ \rightarrow\ e \\ \mid \texttt{let}\ x = e\ \texttt{in}\ e \mid \texttt{let rec}\ f\ x = x\ \texttt{in}\ e \mid \texttt{raise}\ e \\ \mid \texttt{try}\ e\ \texttt{with}\ \texttt{e\_int\_list} i \end{aligned}$$

where, c is a constant, v is a variable, and `e_int_list` is inductively defined as:

$$\texttt{e\_int\_list} := \texttt{i} \rightarrow e \mid \texttt{i} \rightarrow e, \texttt{e\_int\_list}$$

where i is an integer

Definition 2.2 (PicoML Declaration)**.** Declarations in PicoML are declarative statements that assign an expression to an identifier. Alternately, then can also be plain expressions.

$$\texttt{dec} := e \mid \texttt{let}\ x = e \mid \texttt{let rec}\ f\ x = e$$

Definition 2.3 (Recursive Expression)**.** An expression e defined in PicoML is defined to be recursive with respect to an dentifier f if there is a subexpression e' of e (that is not lambda lifted) and has the form $f\ e''$, where e'' is an expression, and no expression that contains e' redefines f

Let us take a look at a couple of examples to understand the above definition.

Example 2.1. Consider the PicoML declaration below:

$$\texttt{let rec}\ f\ x\ =\ \texttt{if}\ x = 0\ \texttt{then}\ 1\ \texttt{else}\ f\ (f\ x - 1);;$$

Note that, the body of the above decclaration is recursive in f, based on the above definition.

Example 2.2. Consider the expression defined below:

$$\texttt{let}\ g\ = f\ \texttt{in}\ g\ 3;;$$

Note that, based on the definition above, the above expression is not recursive in f

Definition 2.4. Tail Recursive Expression A PicoML expression e is said to be tail-recursive with respect to an identifier f if it is recursive in f and one of the following holds:

- e is a constant c or a variable v
- e is of the form $\odot e'$, and e' is not recursive in f
- e is of the form $e' \oplus e''$, and none of e' and e'' is recursive in f
- e is of the form $e'e''$ and e'' is not recursive in f and e' is tail-recursive in f
- e is of the form $\texttt{if}\ e'\ \texttt{then}\ e''\ \texttt{else}\ e'''$ if e' is not recursive in f, and both e'' and e''' are tail recursive in f

- e is of the form `fun` $x \rightarrow e'$
- e is of the form `let` $x = e'$ `in` e'' and, either
  - x is not f (that is, f has the older binding in e''), e' is not recursive in f, and e'' is tail recursive in f, or
  - x is f (that is, the binding of f is updated in e''), and e' is tail recursive in f
- e is of the form `let rec` $g\ x = x$ `in` e', and, one of the following is true
  - g is f (that is, f is given a new binding in e''), or
  - g is not f, (here, f retains its binding in e''), and e'' is tail recursive in f
- e is of the form `try` e' `with` $i1 \rightarrow e_{i1} \mid i2 \rightarrow e_{i2} \mid \ \ldots \mid ik \rightarrow e_{ik}$, and
  - e' is not tail recursive in f, and
  - Each of e_{i1} through e_{ik} is tail recursive in f

Definition 2.5 (Tail Recursive Declaration)**.** A PicoML declaration is said to be tail recursive if one of the following hold:

- It is of the form e
- It is of the form `let` $x = e$
- It is of the form `let rec` $f\ x = e$ and e is tail recursive in f

3 Implementation

For the purpose of the project, we check if a PicoML declaration is tai recursive or not, in the following two ways :

1. We check if the declaration satisfies Definition 2.5 given earlier
2. We transform the delaration into Continuation Passing style, and check for some conditions (covered later)

3.1 Checking Tail Recursion in PicoML

Here is our proposal:

3.2 Tail Recursive Checking for CPS Based on MP7

Here is our proposal:

3.3 Code Structure

Describe the code structure. Justify that the code is modular

4 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.

5 Listing

5.1 Direct stype PicoML expressions

```ocaml
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

5.2 CPS transformed PicoML expressions

```

open Definitions

let rec convert_f_exp e name_of_f =
    (
    match e
    with  ConstCPS (k, c) ->
        (*
         print_string "\nConstCPS \n";
        *)
        convert_f_cont k name_of_f
    | VarCPS (k, g) ->
        (*
        print_string "\nVarCPS \n";
        *)
            (
            match k
            with FnContCPS (number_of_f, e') ->
                let rec_list = convert_f_exp e' name_of_f
                in
                if (g = name_of_f)
                        then (number_of_f :: rec_list)
                        else rec_list
            | _ -> []
            )
    | MonOpAppCPS (k, _, _, _) ->
        (*
        print_string "\nMonOpAppCPS \n";
        *)
        convert_f_cont k name_of_f
    | BinOpAppCPS (k , _, _, _, _) ->
        (*
        print_string "\nBinOpAppCPS \n";
        *)
        convert_f_cont k name_of_f
    | IfCPS (b, e1, e2) ->
        (*
        print_string "\nIfCPS \n";
        *)
        (convert_f_exp e1 name_of_f)@(convert_f_exp e2 name_of_f)
    | AppCPS (k, _, _, _) ->
        (*
        print_string "\nAppCPS \n";
        *)
        convert_f_cont k name_of_f
    | FunCPS (k, _, _, _, _) ->
        (*
        print_string "\nFunCPS \n";
        *)
        convert_f_cont k name_of_f
    | FixCPS (k, _, _, _, _, _) ->
        (*
        print_string "\nFixCPS \n";
        *)
        convert_f_cont k name_of_f
    )

and

convert_f_cont k name_of_f =
    (
    match k
    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] Robert I. Soare, *Computability and recursion*, BULL. SYMBOLIC LOGIC, 1996

[2] Programming Languages and Compilers : CS421, Fall 2015, University of Illinois, Urbana Champaign, Course Web Page