Our goal of this proof is to prove that for all trees t:

(1) party t = party_tr_call t

In order to do this we need to generalize our theorem to work with any continuation. Further we will have to break this proof into parts to examine and compare the functions party and party_tr, party_in and party_in_tr, party_out and party_out_tr, and append and append_tr.

First we want to prove that for all lists l1 and l2

(2) append l1 l2 = append_tr l1 l2

Note here that append is just the expanded code for the built-in OCaml function @. Now in order to prove this we have to generalize it such that for all lists l2 and functions cont: 'a list -> 'b,

(3) cont (append l1 l2) = append_helper l1 l2 cont

where the code for both is below:

```
31  let rec append l1 l2 =
32    match l1 with
33    | [] -> l2
34    | h::t -> h :: (append t l2)
35
36  let append_tr l1 l2 =
37    let rec append_helper l1 l2 cont =
38      match l1 with
39      | [] -> cont l2
40      | h::t -> append_helper t l2 (fun r -> cont (h::r))
41    in
42    append_helper l1 l2 (fun r -> r)
```

Base case: l1 = []
LHS: cont (append [] l2)
=> cont l2 by prog append

RHS: append_helper [] l2 cont
=> cont l2 by prog append_helper

So clearly the LHS and RHS evaluate to the same value.

Inductive step: l1 = h::t
IH: cont' (append t l2) = append_helper t l2 cont'
for all l2 and cont': 'a list -> 'b

LHS: cont (append (h::t) l2)
=> cont (h::(append t l2)) by prog append

RHS: append_helper (h::t) l2 cont
=> append_helper t l2 (fun r -> cont (h::r))
by prog append_helper
=> (fun r -> cont (h::r)) (append t l2) by IH where cont' = (fun r -> cont (h::r))
=> cont (h::(append t l2)) by evaluating the function application

So clearly the LHS and RHS evaluate to the same value thus we have proved (3)

Now we can prove (2) for all l1 and l2:

LHS: append l1 l2

RHS: append_tr l1 l2
=> append_helper l1 l2 (fun r->r) by prog append_tr
=> (fun r->r) (append l1 l2) by (3)
=> append l1 l2 by evaluating the function application

So clearly the LHS and RHS evaluate to the same value thus we have also proven (2)

Now we want to prove that for all trees t and functions cont : int * string list

(4) cont (Party t) = Party_tr t cont

where the code for both is below:

```
(* Memoized Algorithm *)
type tree = Empty
          | Node of int * string * tree * tree *
                    ((int*string list) option) ref

let rec party(t): int * string list =
  match t with
    Empty -> (0, [])
  | Node(v,name,left,right,memo) ->
      (match !memo with
         Some result -> result
       | None ->
           let (infun, innames) = party_in(t) in
           let (outfun, outnames) = party_out(t) in
           let result =
             if infun > outfun then (infun, innames)
             else (outfun, outnames)
           in
           (memo := Some result); result)

and party_in(t) =
  match t with
    Empty -> (0, [])
  | Node(v,name,l,r,_) ->
      let (lfun, lnames) = party_out(l)
      and (rfun, rnames) = party_out(r) in
      (v + lfun + rfun, name :: (append lnames rnames))

and party_out(t) =
  match t with
    Empty -> (0, [])
  | Node(v,_,l,r,_) ->
      let (lfun, lnames) = party(l)
      and (rfun, rnames) = party(r) in
      (lfun + rfun, append lnames rnames)
```

```
(* Tail-recursive Algorithm *)
let rec party_tr (t: tree) (cont: int * string list -> 'a) : 'a =
  match t with
  | Empty -> cont (0, [])
  | Node (v, name, _, _, memo) ->
      match !memo with
      | Some result -> cont result
      | None ->
          party_in_tr t (fun (infun, innames) ->
              party_out_tr t (fun (outfun, outnames) ->
                  let result =
                    if infun > outfun then (infun, innames)
                    else (outfun, outnames)
                  in
                  memo := Some result; cont result)
            )

and party_in_tr (t: tree) (cont: int * string list -> 'a) : 'a =
  match t with
  | Empty -> cont (0, [])
  | Node (v, name, l, r, memo) ->
      party_out_tr l (fun (lfun, lnames) ->
          party_out_tr r (fun (rfun, rnames) ->
              let result = (v + lfun + rfun, name::(append_tr lnames rnames))
              in
              cont result
            )
        )

and party_out_tr (t: tree) (cont: int * string list -> 'a) : 'a =
  match t with
  | Empty -> cont (0, [])
  | Node(v, name, l, r, memo) ->
      party_tr l (fun (lfun, lnames) ->
          party_tr r (fun (rfun, rnames) ->
              let result = (lfun + rfun, append_tr lnames rnames) in
              cont result
            )
        )

(* Driver function *)
let party_tr_call (t: tree) =
  party_tr t (fun r -> r)
```

We can do so by doing a proof by structural induction on t:

Base case: t = Empty

LHS: cont (Party Empty)
=> cont (0, []) by prog party

RHS: Party_tr Empty cont

=> cont (0, []) by prog party_tr

So clearly LHS and RHS evaluate to the same value

Inductive Step: t = Node(v, name, l, r, memo)
IH1: cont_l (Party l) = party_tr l cont_l
for all cont_l
IH2: cont_r (party r) = party_tr r cont_r
for all cont_r

Now we can proceed by examining two cases based on the result of pattern matching !memo

Case 1: !memo = Some result

LHS: cont (Party Node(v, name, l, r, memo))
=> Cont result by prog Party

RHS: Party_tr Node(v, name, l, r, memo) cont
=> Cont result by prog party_tr

So clearly in this case the LHS and RHS evaluate to the same value.

Case 2: ! memo = None
Now in order to complete the rest of this proof, we will have to proceed by first proving, again by structural induction on a tree t that

(5) cont_out (Party_out t) = Party_out_tr t cont_out
for all cont_out

Base case: t = Empty

LHS: cont_out (Party_out Empty)

=> cont_out (0, []) by prog party_out

RHS: party_out_tr Empty cont_out
=> cont_out (0, []) by prog party_out_tr

Inductive Step: t = Node (v, name, l, r, memo)
note here we will add an additional two inductive hypotheses that we won't use directly to prove (5) but that we will need later on in the proof.

IH 3: cont_out_l (party_out l) = party_out_tr l cont_out_l
for all cont_out_l
IH 4: cont_out_r (party_out r) = party_out_tr r cont_out_r
for all cont_out_r

* let party l = (lfun, lnames) *
* let party r = (rfun, rnames) *
LHS: cont_out (party_out Node (v, name, l, r, memo))
=> cont_out (lfun + rfun, append lnames rnames)
by prog party_out

RHS: party_out_tr Node (v, name, l, r, memo) cont_out
=> party_tr l (fun (lfun, lnames) ->
party_tr r (fun (rfun, rnames) ->
cont_out (lfun + rfun, append_tr lnames rnames)))
by prog party_out_tr
=> (fun (lfun lnames) -> party_tr r (fun (rfun, rnames) ->
cont_out (lfun + rfun, append_tr lnames rnames))
(party l)
by IH 1

=> party_tr r (fun (rfun, rnames) ->
cont_out (lfun + rfun, append_tr lnames rnames))
by evaluating the function application

=> (fun (rfun, rnames) ->
    cont_out (lfun + rfun, append_tr lnames rnames))
    (party r)
by IH2

=> cont_out (lfun + rfun, append_tr lnames rnames)
by evaluating the function application

=> cont_out (lfun + rfun, append lnames rnames)
by (2)

this clearly the LHS and RHS evaluate to the same value so we have proven (5) for all t

Now to proceed with our proof, we still need to prove by structural induction on t that

(6) cont_in (party_in t) = party_in_tr t cont_in
for all cont_in

Base case: t = Empty

LHS: cont_in (party_in Empty)
=> cont_in (0, [])
by prog party_in

RHS: party_in_tr Empty cont_in
=> cont_in (0, [])
by prog party_in_tr

Inductive step: t = Node(v, name, l, r, memo)
note that we don't need to introduce any additional induction hypotheses here as we can use the formulas we have previously proved

☆ let party_out l = (lfun, lnames) ☆
☆ let party_out r = (rfun, rnames) ☆

LHS: cont_in (party_in Node(v, name, l, r, memo))
=> cont_in (v + lfun + rfun, name :: (append lnames rnames))
by prog party_in

RHS: party_in_tr Node(v, name, l, r, memo) cont_in
=> party_out_tr l (fun (lfun, lnames) ->
party_out_tr r (fun (rfun, rnames) ->
cont_in (v + lfun + rfun, name :: (append_tr lnames rnames))))
by prog party_in_tr

=> (fun (lfun, lnames) -> party_out_tr r (fun (rfun, rnames) ->
cont_in (v + lfun + rfun, name :: (append_tr lnames rnames))))
(party_out l) by (5)

=> party_out_tr r (fun (rfun, rnames) ->
cont_in (v + lfun + rfun, name :: (append_tr lnames rnames))))
by evaluating the function application

=> (fun (rfun, rnames) ->
cont_in (v + lfun + rfun, name :: (append_tr lnames rnames))))
(party_out r) by (5)

=> cont_in (v + lfun + rfun, name :: (append_tr lnames rnames))
by evaluating the function application

=> cont_in (v + lfun + rfun, name :: (append lnames rnames))
by (2)

thus clearly the LHS and RHS evaluate to the same value so we have proven (6) for all t.

Now we want to come back to our original inductive step which recall we were proving

(4) cont (party t) = party_tr t cont

for t = (v, name, l, r, memo)
and we were in case 2 where !memo = None
Now again we have two subcases depending
on the tuples returned by party_in and party_out

* let (infun, innames) = party_in Node (v, name, l, r, memo) *
* let (outfun, outnames) = party_out Node (v, name, l, r, memo) *

case 2.1 infun > outfun

LHS: cont (party Node(v, name, l, r, memo))
=> cont (infun, innames) by prog party

RHS: party_tr Node(v, name, l, r, memo) cont
=> party_in_tr Node (v, name, l, r, memo) (fun (infun, innames) ->
party_out_tr Node(v, name, l, r, memo) (fun (outfun, outnames) ->
cont (if infun > outfun then (infun, innames)
else (outfun, outnames)))) by prog party_tr

* note we include the if statement still as we have not passed values to the arguments infun and outfun yet so we don't know which branch will execute *

=> (fun (infun, outfun) ->
party_out_tr Node(v, name, l, r, memo) (fun (outfun, outnames) ->
cont (if infun > outfun then (infun, innames)
else (outfun, outnames))))
(party_in Node (v, name, l, r, memo)) by (6)

=> party_out_tr Node(v, name, l, r, memo) (fun (outfun, outnames) ->
cont (if infun > outfun then (infun, innames)
else (outfun, outnames)))
by evaluating the function application

=> (fun (outfun, outnames) ->
    cont (if infun > outfun then (infun, innames)
          else (outfun, outnames))
    (party_out Node(v, name, l, r, memo)) by (5)

note that now both infun and outfun have the same value as in the RHS so we can evaluate the if branch

=> cont (infun, innames)
by evaluating the function application

Clearly the LHS and RHS evaluate to the same value in this subcase thus (4) holds here

Case 2.2 infun ≤ outfun

LHS: cont (party Node(v, name, l, r, memo))
=> cont (outfun, outnames) by prog party

RHS: party_tr Node(v, name, l, r, memo) cont
=> party_in_tr Node(v, name, l, r, memo) (fun (infun, innames) ->
    party_out_tr Node(v, name, l, r, memo) (fun outfun, outnames ->
      cont (if infun > outfun then (infun, innames)
            else (outfun, outnames)))) by prog party_tr

=> (fun (infun, outfun) ->
    party_out_tr Node(v, name, l, r, memo) (fun outfun, outnames)->
      cont (if infun > outfun then (infun, innames)
            else (outfun, outnames))))
    (party_in Node(v, name, l, r, memo)) by (6)

=> party_out_tr Node(v, name, l, r, memo) (fun outfun, outnames)->
    cont (if infun > outfun then (infun, innames)
          else (outfun, outnames)))
by evaluating the function application

=> (fun (outfun, outnames) ->
    cont (if infun > outfun then (infun, innames)
          else (outfun, outnames))
    (Party_out Node(v, name, l, r, memo)) by (5)

=> cont (outfun, outnames)
    by evaluating the function application

Clearly the LHS and RHS evaluate to the same value in this subcase thus (4) holds here also

So, since (4) holds in every case we have proven it is true for all trees t.

Now finally we can prove our original theorem

(1) party t = party_tr_call t

LHS: party t

RHS: party_tr_call t
=> party_tr t (fun r -> r) by prog party_tr_call
=> (fun r -> r) (party t) by (4) where cont=(fun r->r)
=> party t by evaluating the function application

Thus we have proved that for all trees t, our tail-recursive version of the given party-optimization problem gives the same result.