INSTITUT POLYTECHNIQUE DE PARIS

Reactive and Synchronous Systems
A Compiler for mini-Lustre

Tianchi Yu

April 13, 2021

1. QUESTION 1

Download the compiler then compile and execute the corresponding code for program simple.mls.

The simple.mls contains a node n:

```
node n (i: unit) returns (o: unit);
let
  o = print("coucou\n");
tel
```

After compile, we get the ocaml code, we only show parts of important snippet:

```
type n'_1_mem = unit

let n'_1_init () = ()

let n'_1_step mem' (i) =
  let (aux'1) = (
    (print_string "coucou\n");
    flush_all()) in
  let (o) = aux'1 in
  (o)
```

Here we can see the node memory, node initialization and node update funtion, where we will do more extensions in imp.ml later.

2. Question 2

Carefully read the files typed_ast.mli and imp_ast.mli that represent abstract syntax trees that you have to manipulate.

There are some very important abstract syntax we need to take care.

In **typed_ast.mli** file, it defines the types to represent the abstract syntax tree annotated with type annotations, for example, the type *t_expr* and its element *texpr_desc* which is type of *t_expr_desc* are basic expression type and its description.

In the **imp_ast.mli** file, it defines the types to represent abstract syntax tree from the target imperative language, which is OCaml here. For example, type mem represents the declaration of the intern memory of one node of lustre, which contains *fby_mem* and *node_mem*; type init represents the entity of initialization of the functions, which contains *fby_init* and *node_init*. And the type *m_expr* describes the expression in OCaml. And type *m_equation* is used to represent the equation happens in OCaml.

3. Question 3

Complete places with holes in the files normalization.ml, scheduling.ml and imp.ml. These places are indicated with a (TODO *).*

3.a. normalization.ml

We need to modify normalize method in case of *e.texpr_desc* is **TE_binop** and **TE_fby**. For *TE_binop* there are two expressions, we need to normalize them with order of inputs by the recursive function *normaliwe ctx e*; And for *TE_fby*, firstly, we need to consider the normalisation of the expression e, by setting the *teq_expr=TE_fby* which contains the normalisation of the element after fby; because *TE_fby* is *const list * t_expr*, c is const list which doesn't change values of ctx and e, we also need to implement normalisation for e1 and combine the declarations and equations.

```
let rec normalize ctx e =
  match e.texpr_desc with
  | TE_const _ | TE_ident _ -> ctx, e

  | TE_unop(op,e1) ->
      let ctx, e1' = normalize ctx e1 in
      ctx, { e with texpr_desc = TE_unop(op,e1') }
```

```
  | TE_binop(op,e1,e2) ->
      let ctx, e1' = normalize ctx e1 in
      let ctx, e2' = normalize ctx e2 in
        ctx, {e with texpr_desc = TE_binop(op,e1',e2')}

  | TE_fby(c,e1) ->
      let (new_vars,new_eqs), e1' = normalize ctx e1 in
      let e1_decl, e1_patt, e1_expr = new_pat e1' in
      let e1_eq ={ teq_patt = e1_patt;teq_expr = e1'; } in
      (*transfer e1_expr.texpr_desc to TE_tuple type*)
      let e1_expr = match e1_expr.texpr_desc with
        | TE_tuple _ -> e1_expr
        | _ -> { e1_expr with texpr_desc = TE_tuple [e1_expr]}
      in
      let e_decl, e_patt, e_expr = new_pat e in
      let e_eq = { teq_patt = e_patt;
                   teq_expr = {e with texpr_desc = TE_fby(c,e1_expr) };}in
      (e_decl@e1_decl@new_vars, e_eq::e1_eq::new_eqs), e_expr
```

3.B. SCHEDULING.ML

This transformation orders equations from a node so that they can be executed sequentially. The order between equations should respect the causality relation between variables. Precisely, an equation x = e must be scheduled after all the variables read in e have been scheduled. An equation x = v fby e must be scheduled before all equations that read x.

In this file, we need to complete *TE_ident*, *TE_fby* and *TE_if*. Note that, if we use *add_vars_of_exp s e2* for *TE_fby*, there will be error of causality, since *add_vars_of_exp* is only used for adding the variables whose expression is instant.

```
let rec add_vars_of_exp s {texpr_desc=e} =
  match e with
  | TE_const _ -> s
  | TE_ident x -> if S.mem x s then s else S.add x s
  | TE_fby (e1, e2) -> s (*add_vars_of_exp s e2*)
  | TE_unop (_,e') -> add_vars_of_exp s e'
  | TE_binop (_,e1,e2) ->
      let s = add_vars_of_exp s e1 in
      add_vars_of_exp s e2
  | TE_if(e1,e2,e3) ->
          let s = add_vars_of_exp s e1 in
          let s = add_vars_of_exp s e2 in
                add_vars_of_exp s e3
  | TE_app (_,l) | TE_prim (_,l) | TE_print l ->
```

```
      List.fold_left add_vars_of_exp s l
  | TE_tuple l -> List.fold_left add_vars_of_exp s l
```

3.C. IMP.ML

In this file, we need to complete *compile_base_expr* function and *compile_equation*. Precisely, we modify the returns of *TE_if* in *compile_base_expr* and *TE_fby* and *TE_app* in *compile_equation*. Here is some code snippets.

```
let rec compile_base_expr e =
  let desc =
    match e.texpr_desc with
    | TE_const c -> ME_const c
    | TE_ident x -> ME_ident x
    | TE_unop (op, e) -> ME_unop(op, compile_base_expr e)
    | TE_binop (op, e1, e2) ->
        let ce1 =  compile_base_expr e1 in
        let ce2 =  compile_base_expr e2 in
        ME_binop (op, ce1, ce2)
    | TE_if (e1, e2, e3) ->
        let ce1 = compile_base_expr e1 in
        let ce2 = compile_base_expr e2 in
        let ce3 = compile_base_expr e3 in
        ME_if (ce1,ce2,ce3)
    | TE_tuple el -> ME_tuple (List.map compile_base_expr el)
    | TE_print el -> ME_print (List.map compile_base_expr el)
    | TE_fby _ -> assert false (* impossible car en forme normale *)
    | TE_app _ -> assert false (* impossible car en forme normale *)
    | TE_prim(f, el) ->
        let _, f_out_ty =
          try List.assoc f Typing.Delta.prims
          with Not_found ->
            Printf.fprintf stderr "not a prim : %s" f;
            assert false
        in
        ME_prim(f, List.map compile_base_expr el, List.length f_out_ty)
  in
  { mexpr_desc = desc; mexpr_type = e.texpr_type; }
```

For the returns of *TE_fby* and *TE_app* in *compile_equation*, I want to explain in details. See the comments of in this code snippets.

```
    let compile_equation
      {teq_patt = p; teq_expr = e}
      ((mem_acc: Imp_ast.mem),
```

```
  (init_acc: Imp_ast.init),
  (compute_acc: Imp_ast.m_equation list),
  (update_acc: (string * Imp_ast.atom) list)) =
let tvars = compile_patt p in
match e.texpr_desc with
(*firstly, change (e1,e2) to (c,e1)*)
| TE_fby(c,e1) ->
begin
  match e1.texpr_desc with
  (*after normalization, the type of e1 should be TE\_tuple*)
  | TE_tuple [] ->
    mem_acc, init_acc, compute_acc, update_acc
  | TE_tuple e1_l ->
  begin
    (*let ce1 = compile_base_expr e1 (*return type is m_expr*) in*)
    let para1 = (mem_acc,init_acc,update_acc,[]) in
    let para2 = (List.combine (List.combine e1_l c) tvars) in
    (*note that tvars is type {tpatt_desc,tpatt_type}*)
    let (new_mem_acc, new_init_acc, new_update_acc, cfby) = List.fold_left
        ( fun (mem_acc, init_acc, update_acc, cfby) ((e,c),(d,t)) ->
          let next_name = gen_next_id d in
          (*change d to "fby", the name of next from aux' to fby*)
          let new_fby_name = (next_name, t) in
          let new_mem_acc = {fby_mem = new_fby_name::mem_acc.fby_mem ;
                             node_mem = mem_acc.node_mem;} in
          let new_init_name = (next_name, c) in
          let new_init_acc = {fby_init=new_init_name::init_acc.fby_init;
                              node_init=init_acc.node_init;} in
          (*(compile\_atom for each expr element)*)
          let new_atom = (next_name,compile_atom e) in

          let new_update_acc = new_atom::update_acc in
          (*let test_name = gen_next_id "test" in *)
          (*change d to "fby", the name of next from aux' to fby*)

          let cfby =  { mexpr_desc = ME_mem next_name;
                        mexpr_type = [t]}::cfby in
          (*this value affect the name in the step mem equation*)
          (new_mem_acc, new_init_acc, new_update_acc, cfby)
        ) para1 para2
    in
    let eq = {meq_patt = tvars; meq_expr = {mexpr_desc = ME_tuple cfby ;
              mexpr_type = e.texpr_type}} in
    let new_compute_acc = eq::compute_acc in
```

```
        new_mem_acc, new_init_acc, new_compute_acc,new_update_acc
      end
      | _ -> assert false
    end


  | TE_app(n,el) ->
      (*mem_acc, init_acc, compute_acc, update_acc*)

      begin
    (* generate a new node mem*)
    let mem_name = gen_mem_id n in
        let new_node_mem = (mem_name, n) in
        (* adding it to the record of mem_acc *)
        let new_mem_acc = {fby_mem = mem_acc.fby_mem ;
                           node_mem = new_node_mem::(mem_acc.node_mem) } in
        let new_el = List.map (fun e1 -> compile_base_expr e1) el in
    let desc = ME_app(n,mem_name, new_el) in
    let new_eq = {meq_patt = tvars;
                  meq_expr = { mexpr_desc = desc ;
                               mexpr_type = e.texpr_type;} ;} in
    let new_init_acc = {fby_init = init_acc.fby_init ;
                        node_init = new_node_mem::init_acc.node_init} in
    let new_compute_acc = new_eq::compute_acc in

    (* To update the update_acc variable, but no need for this type*)
(*
    let ael = List.map compile_atom el in
    (*List.map (fun e1 -> compile_atoms e1) el in*)
        let update_ael = List.map (fun atom -> (mem_name, atom)) ael in
    let new_update_acc = List.append update_ael update_acc in
*)
    new_mem_acc, new_init_acc, new_compute_acc, update_acc (* DONE *)
      end


  | _ ->
    let eq = {meq_patt = tvars; meq_expr = compile_base_expr e} in
    mem_acc, init_acc, eq::compute_acc, update_acc
```

4. QUESTION 4

We propose that you complete the compiler with two extensions : The addition of sampling ***when*** *and combination operation* ***merge****. The modular reset operator.*

4.A. WHEN & MERGE

Firstly, define the abstract syntax for when and merge. Because the expression of when and merge are like "z = x when true(h)" and "merge h (ture -> z) (false -> t)". As we know that,

```
type const =
  | Cunit
  | Cbool of bool
  | Cint of int
  | Cfloat of float
  | Cstring of string
```

Then we could add the type of when and merge to all the modules, for example, in *ast.mli*

```
type p_expr =
  { pexpr_desc: p_expr_desc;
    pexpr_loc: location; }
and p_expr_desc =
  | PE_const of const
  | PE_ident of ident
  | PE_unop of unop * p_expr
  | PE_binop of binop * p_expr * p_expr
  | PE_app of ident * p_expr list
  | PE_if of p_expr * p_expr * p_expr
  | PE_fby of p_expr * p_expr
  | PE_tuple of p_expr list
  | PE_when of p_expr * const * p_expr
  | PE_merge of p_expr * (const * p_expr) list
```

Without considering the clocks, I want to implement the normalization to these two methods. The logic for normalize *TE_when* is to realize that

```
e1 when C(e2) =>
{ x = y when C(z);
  y = normalize e1 ;
  z = normalize e2 ;
  x;}
```

so we only need to normalize for all expressions and combine them in order, here is the code:

```
  | TE_when(e1,c,e2) ->
      let ctx, e1' = normalize ctx e1 in
```

```
    let e1_decl, e1_patt, e1_expr = new_pat e1' in
    let e1_eq = { teq_patt = e1_patt;teq_expr = e1'; } in
    let e1_expr = { e1_expr with texpr_desc = TE_tuple [e1_expr] } in

    let (new_vars, new_eqs), e2' = normalize ctx e2 in
    let e2_decl, e2_patt, e2_expr = new_pat e2' in
    let e2_eq ={ teq_patt = e2_patt;teq_expr = e2'; } in
    let e2_expr = { e2_expr with texpr_desc = TE_tuple [e2_expr] } in

    let e_decl, e_patt, e_expr = new_pat e in
    let e_eq = { teq_patt = e_patt;
                 teq_expr = { e with texpr_desc = TE_when(e1_expr,c,e2_expr) }; } i
    (e_decl@e1_decl@e2_decl@new_vars, e_eq::e1_eq::e2_eq::new_eqs), e_expr
```

For the *TE_merge*, the normalization should be

```
merge e' (c,e) list =>
{ x = merge y (c1 -> z1) ... (cn -> zn);
  y = normalize e';
  z1 = normalize e_1;
  ...
  zn = normalize e_n;
  x ;}
```

So the code is following:

```
| TE_merge(e1, cel) ->
    let ctx, e1' = normalize ctx e1 in
    let e1_decl, e1_patt, e1_expr = new_pat e1' in
    let e1_eq = { teq_patt = e1_patt; teq_expr = e1'; } in
    let e1_expr = { e1_expr with texpr_desc = TE_tuple [e1_expr] } in

    let (new_vars,new_eqs), cel' = List.fold_left
      (fun (ctx, l) (c, e) -> let ctx, e' = normalize ctx e in (ctx, (c, e')::l))
      (ctx, []) cel in

    let cel_decl, cel_expr, cel_eq = List.fold_left
      (fun ( cel_decl, cel_expr, cel_eq ) (c, e')->
        let e_decl, e_patt, e_expr = new_pat e' in
        let e_eq ={ teq_patt = e_patt; teq_expr = e'; } in
        let e_expr = { e_expr with texpr_desc = TE_tuple [e_expr] } in
        ( e_decl@cel_decl, (c, e_expr)::cel_expr,e_eq::cel_eq))
      ([], [] ,[]) cel' in

    let e_decl, e_patt, e_expr = new_pat e in
    let e_eq ={ teq_patt = e_patt;
```

```
                teq_expr = { e with texpr_desc = TE_merge(e1_expr, cel_expr) }; }
    in

    (e_decl@e1_decl@cel_decl@new_vars, cel_eq@(e_eq::e1_eq::new_eqs)), e_expr
```

Given the scheduling.ml as following:

```
| TE_when (e1,c,e2) ->
    let s = add_vars_of_exp s e1 in
    add_vars_of_exp s e2
| TE_merge (e1,cel) ->
    let s = add_vars_of_exp s e1 in
    List.fold_left (fun s (_, e) -> add_vars_of_exp s e) s cel
```

Also, to make the program running, add corresponding methods to deal with them in other ast files, typing.ml, checks.ml and typed_ast_utils.ml, etc.

However, I didn't finish the imp.ml.

4.B. Optimisation

To implement this program, we need to talk more about how the clocks work. The clocks are first calculated independently for each node. Ideally, we need to create a Clock module, to implement clock expressions and functionalities. Actually, the printer, abstract syntax tree and tools of *Clock* could be similar with *Typed* files.

Once we build the Clock Module, we need to change the type inside of our normalization, scheduling and implementation.

4.C. RESET

The lustre has no reset modular, so it is painful to make such a component.