INSTITUT POLYTECHNIQUE DE PARIS

Reactive and Synchronous Systems
A Compiler for mini-Lustre

Tianchi Yu

March 23, 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*, because *TE_fby* is *const list * t_expr*, c is const list which doesn't change values of ctx and e, we only need to implement normalisation for e1.

```
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 ctx, e1' = normalize ctx e1 in
        ctx, {e with texpr_desc = TE_fby(c,e1')}
```

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_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
  | TE_fby(e1,e2) ->
        (* mem_acc, init_acc, compute_acc, update_acc*)
    begin

      let ce2 = compile_base_expr e2 (*return type is m_expr*) in

      (* the init_names is used to define the variable names
       * corresponding to fby in the memory of node;
```

```
      *)
    let init_names = List.map (fun m_const -> gen_next_id "fby") e1 in

    (* to define the init value of this variable, we combien the names with
     * the pre value, which is e1 here;
     *)
    let add_fby_init = List.combine init_names e1 in
    (* to define the memory and description of this variable,
     * we combien the names with the type of this expression;
     *)
    let add_fby_mem = List.combine init_names e.texpr_type in
    let new_mem_acc =
        {fby_mem = List.append add_fby_mem mem_acc.fby_mem;
        node_mem = mem_acc.node_mem} in
    let new_init_acc =
        {fby_init = List.append add_fby_init init_acc.fby_init;
        node_init = init_acc.node_init} in

    (* then we need to compute the return value of this function,
     * which should change "y = x fby z" to
     * "let y = mem.fby_next1 in mem.fby_next1 <- z; (y)";
     * so the equation added to compute_acc should be the variable
     * of fby with the type m_equation;
     * NOT SUCCESSFUL HERE
     *)
    let eq2 = {meq_patt = tvars; meq_expr = ce2} in
    (* a problem here, because the eq need the value of fby_mem*)

    (* construct the update_acc by combine the init_names and
     * the atoms gotten by the second expression of fby,
     * which is e2 here.
     *)
    let ae2 = compile_atoms e2 in
    let update_ae2 = List.combine init_names ae2 in
    let new_update_acc = List.append update_ae2 update_acc in
    new_mem_acc, new_init_acc, compute_acc, new_update_acc
  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

        (* add the new_node_mem to the node_init *)
        let new_init_acc = {fby_init = init_acc.fby_init ;
        node_init = new_node_mem::init_acc.node_init} in

        (* get all the m_equations in the el (e_texpr list)
         * and add them into compute_acc
         *)
        let new_eq_l = List.map (fun e -> {meq_patt = tvars; meq_expr = e}) new_el in
        let new_compute_acc = List.append new_eq_l compute_acc in

        (* get all the atoms inside of el, and combile the node name
         * update the nodes
         *)
        let ael = List.map compile_atom 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, new_update_acc
          end

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

Eventually, I still have some bugs about fby and app two cases. The key problem for me is that I have no idea how to transfer the value of *fby_mem* to the equation. Here are some alternatives methods which are not that correct, but it offers one idea.

```
        let cfby =  { mexpr_desc =
        ME_tuple(List.map (fun e' -> {mexpr_desc = ME_ident(e');
        mexpr_type = ce2.mexpr_type}) init_names); mexpr_type = ce2.mexpr_type} in
        let eq = {meq_patt = tvars; meq_expr = cfby} in
        eq::compute_acc
```

In this way, I can only get the name of fby value without the reference of node memory.

I write a simple test.mls to test my compiler, and it returns some obvious bugs with the above problems.

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.*

I was blocked by the Question 3, so I didn't have the chance to implement these methods. I explain the basic ideas about them.

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)". Then for example,

```
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 * ident * p_expr
  | PE_merge of p_expr * (ident * p_expr) list
```

Then we need to add corresponding methods to deal with them in other ast files, typing.ml and normalize.ml, schedule.ml, imp.ml, etc.

4.B. RESET

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