Langages Réactifs Synchrones

Master IP Paris – Cyber-Physical Systems

mini-Lustre

Émilie THOMÉ

Mars 2021

1 Introduction

J'ai choisi de faire le projet mini-Lustre. Le but était de créer un générateur de code OCaml à partir de code mini-Lustre.

Le langage mini-Lustre se restreint aux expressions suivantes :

- les contantes booléennes, entières, flotantes et les chaines de caractères

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

- des opérateurs unitaires

```
type unop =
  | Unot | Uminus | Uminus_f
```

- des opérateurs binaires

```
type binop =
  | Beq | Bneq | Blt | Ble | Bgt | Bge
  | Badd | Bsub | Bmul | Bdiv
  | Badd_f | Bsub_f | Bmul_f | Bdiv_f
  | Band | Bor
```

- des appels de variables

```
type ident = string
```

- des application de noeuds ou fonctions
- l'utilisation de if (...) then (...) else (...)
- le "followed by"
- l'utilisation d'appels groupés par tuples
- la forme `x when C(a)`

- la forme `merge (x) (c1 -> a1) ... (cn -> an)`

```
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)
```

Les morceaux de code sont issus des fichiers `ast.mli` et `asttypes.mli`.

2 Contenu

Comme dit précédemment, les expression `when` et `merge` ont été ajoutées.

`(t,y) = (0,x) when false(b);` et `(t,y) = merge (c) (false -> (0,x1)) (true -> (0,x2));` ne sont pas acceptés car `when` et `merge` ne supportent pas les tuples.

Cependant, `if (...) then (...) else (...)` supporte les tuples ainsi que `fby`.

De plus, la variables utilisée dans `when` et `merge` doit être parenthèsée comme suivent `y = (x1) when false(c); y = merge (c) (false -> x1) (true -> x2);`.

L'initialisation des `fby` doit toujours se faire par une constante.

3 Fonctionnement des horloges

Les horloges sont une première fois calculées de manière indépendante pour chaque noeud.

Ces horloge sont des fonctions des horloges d'entrée de noeud. Cela permet d'avoir des entrées de noeuds avec des horloges différentes.

Les horloges des `fby` sont initialisées à `([liste des horloges d'entrée] -> base)` car l'initialisation des `fby` est toujours une constante et donc toujours disponible. Elles seront ensuite mise à jour afin de montrer par récurrence qu'elles sont toujours disponibles au tour suivant ou non. Les entrées sont initialisées à la fonction identité sur les horloges `([liste des horloges d'entrée] -> l'horloge de la dite entrée)`. Les autres variables sont `NotClocked`. Cette première phase permet de vérifier la synchronisation au sein d'un même noeud mais aussi à relever quelles sont les conditions sur les entrées qui permettent d'avoir une synchronisation lors de l'appel du noeud.

On vérifie que le `main` soit bien compatible avec une fonction d'horloge `base -> base`.

Se passe ensuite une deuxième vérification mais cette fois du fichier en général. On regarde si les conditions relevées sont satisfaites lors des appels.

Les horloges sont distinguées en horloges de base (avec l'horloge `Base`, l'horloge `Sample`, la fonction d'horloge `ClockVar` et l'horloge de base non initialisée `NotClockedBase`) et en horloge d'utilisation. Les entiers utilisés dans `ClockVar` et `ClockApp` permettent d'imprimer ces horloges dans la console et de vérifier leur utilisation.

```
and base_clock =
  | Base
  | Sampled of base_clock * const * c_expr
    (* dans les noeuds, les clocks des variables sont
    des fonctions des clocks des inputs du noeuds *)
  | ClockVar of (clock list -> base_clock) * int
  | NotClockedBase
```

```
and clock =
    (* pour les applications *)
  | ClockApp of ((clock list) -> clock) * int
    (* pour les equations : cexpr_clock et cpatt_clock *)
  | ClockTuple of clock list
    (* pour les variables non initialisees *)
  | NotClocked
  | ClockBase of base_clock
```

4 Quelques fonctions mises en lumière

4.1 Compatibilité des horloges

Une fonction `compatible` assure la compatibilité des horloges lors des manipulations :

```
let rec compatible_base add actual_ck expected_ck inputs low_clock =
  begin
  match normalize_base_clock actual_ck, normalize_base_clock expected_ck with

    | ClockVar (f1, i1), ClockVar (f2, i2) when add ->
      Node.add_condition (fun ckl ->
        compatible_base false (f1 ckl) (f2 ckl) ckl low_clock);

      compatible_base false (f1 inputs) (f2 inputs) inputs low_clock

    | ClockVar (f1, i1), ClockVar (f2, i2) ->
      compatible_base false (f1 inputs) (f2 inputs) inputs low_clock

    | Sampled (ck1, c1, ce1), Sampled (ck2, c2, ce2)
    when (compatible_base add ck1 ck2 inputs low_clock) && (c1 = c2) -> true

    | Base, _ -> true
    | ck, Sampled (ck',_ ,_ )-> compatible_base add ck ck' inputs low_clock
    | NotClockedBase, NotClockedBase -> true
    | NotClockedBase, ck -> not low_clock
    | ck, NotClockedBase -> low_clock
    | _, _ -> actual_ck = expected_ck
  end

(* expected_ck <= actual_ck *)
and compatible add actual_ck expected_ck inputs low_clock =
  begin match normalize_clock actual_ck, normalize_clock expected_ck with
    | ClockTuple ckl1, ClockTuple ckl2 ->
      begin try List.fold_left2
            (fun well_ck ac_ck ex_ck ->
              well_ck && (compatible add ac_ck ex_ck inputs low_clock))
            true ckl1 ckl2
      with Invalid_argument _ -> false end
    | ClockBase bck1, ClockBase bck2 ->
      compatible_base add bck1 bck2 inputs low_clock
    | NotClocked, NotClocked -> true
    | NotClocked, ck -> not low_clock
```

```
    | ck, NotClocked -> low_clock
    | ck1, ck2 -> false
  end
```

Les arguments `add` permettent de dire si la compatibilité entre les deux horloges doit être ajoutée aux conditions pour que le fichier soit bien synchrone.

L'argument `inputs` permet de spécifer avec quelles entrées la compatibilité doit être vérifiée. Les entrées peuvent être des `NotClocked`. Ou elles peuvent être des horloges définies dans la deuxième étape de la vérification des conditions.

L'argument `low_clock` permet de savoir si l'on utilise une autre fonction renvoyant l'horloge la plus faible d'une liste d'horloges.

4.2 L'horloge des expressions

La fonction donnant une horloge à une expression est utilisée lors de la première étape de vérification. Elle renvoie des `ClockTuple(...)` d'horloges afin de normaliser les sorties de la fonction. Les horloges sont, je le rappelle, des fonction `ClockVar(...)` prenant en input les horloges des entrée du noeud.

Une constante les toujours accessible donc a une horloge fonction donnant l'horloge de base.

```
and clock_expr_desc loc desc =
    match desc with
    | TE_const c ->
        CE_const c ,
        ClockTuple [ClockBase (ClockVar
            ((fun ckl -> Base), List.length !Node.in_notclocked))]
```

Une variable a l'horloge de son expression associée.

```
    | TE_ident x ->
        let ck = clock_patt_var loc x in
        CE_ident x , ClockTuple [normalize_clock ck]
```

De même pour un opératieur unitaire.

```
    | TE_unop (op, e) ->
        let ce = clock_expr e in
        CE_unop (op, ce) , ce.cexpr_clock
```

Pour un opérateur binaire, la plus petite horloge est choisie.

```
    | TE_binop (op, e1, e2) ->
        let ce1 = clock_expr e1 in
        let ce2 = clock_expr e2 in
        let ckl = [ce1.cexpr_clock; ce2.cexpr_clock] in
        let ck = lowest_clock ckl in
        CE_binop (op, ce1, ce2), ck
```

Pour les applications, je vérifie sont utilisation et ensuite j'applique la fonction horloge du noeud associé.

```
    | TE_prim (f, el) | TE_app (f, el) ->
        begin try
          let ck , is_prim = Delta.find f in
          match ck with
          | ClockApp(ckf, i) when i = (List.length el) ->
          begin
            let cel = List.map clock_expr el in
            let actual_clocks = List.map
```

```
            (fun ce -> normalize_clock ce.cexpr_clock) cel in
          Node.add_call (f, actual_clocks);
          let ck_out =
              (try ckf actual_clocks
              with  ErrorMessage s -> error loc (Other s)) in
          (if is_prim then CE_prim(f, cel) else CE_app(f, cel)), ck_out
        end
        | ClockApp(ckf, i) when i > (List.length el)
          -> error loc TooFewArguments
        | ClockApp(ckf, i) when i < (List.length el)
          -> error loc TooManyArguments
        | _ -> error loc (ExpectedClockApp ck)
      with Not_found -> error loc (UnboundNode f)
      end
```

Pour un print, le tuple des horloges des expressions du tuple d'entrée du print est calculé.

```
  | TE_print el ->
      let cel = List.map clock_expr el in
      let ck = ClockTuple (List.map
          (fun ce -> normalize_clock ce.cexpr_clock) cel) in
      CE_print(cel), ck
```

Je vérifie que les valeurs dans le `if (...) then (...) else (...)` soient accessibles au bon moment.

```
  | TE_if (e1, e2, e3) ->
      let ce1 = clock_expr e1 in
      let ce2 = clock_expr e2 in
      let ce3 = clock_expr e3 in
      let ck = ce1.cexpr_clock in
      begin match ck, ce2.cexpr_clock, ce3.cexpr_clock with

        | ClockTuple [ClockBase (ClockVar (f, i))],
          ClockTuple ckl1,
          ClockTuple ckl2
              when
              (List.fold_left
              (fun b ck1 ->
                  b &&
                  (compatible true ck1
                      (ClockBase (ClockVar ((fun ckl ->
                              Sampled (f ckl, Cbool true, ce1)), i)))
                      !Node.in_notclocked false))
              true ckl1)
              && (* identique avec ckl2 et ce1 vallant false *)
          -> CE_if (ce1, ce2, ce3),
              ClockTuple (List.fold_left
                  (fun l _ -> ClockBase (ClockVar (f, i)) :: l) [] ckl2)

        | _, _, _ -> (* des erreurs *)
      end
```

Le `fby` est mise à jour ici avec l'horloge de ce qu'il y a après le `fby`.

```
  | TE_fby (e1, e2) ->
```

```
    let ce2 = clock_expr e2 in
    CE_fby (e1, ce2), ce2.cexpr_clock
```

Pour un tuple, le tuple des horloges des expressions du tuple est calculé.

```
| TE_tuple el ->
    let cel = List.map clock_expr el in
    CE_tuple cel,
    (ClockTuple (List.map (fun e -> normalize_clock e.cexpr_clock) cel))
```

Pour un `when` je fractionne l'horloge.

```
| TE_when (e1, c, e2) ->
    let ce1 = clock_expr e1 in
    let ce2 = clock_expr e2 in
    let ck1 = ce1.cexpr_clock in
    let ck2 = ce2.cexpr_clock in
    begin match ck1, ck2 with
    |   ClockTuple [ClockBase (ClockVar (f1, i1))],
        ClockTuple [ClockBase (ClockVar (f2, i2))]
        when compatible true ck1 ck2 !Node.in_notclocked false
    ->
      CE_when (ce1, c, ce2),
      ClockTuple [ClockBase (ClockVar ((fun ck1 ->
        Sampled (normalize_base_clock(f1 ck1), c, ce2)), i1))]
    | _, _ -> (* des erreurs *)
    end
```

Pour un `merge` comme pour un `if (...) then (...) else (...)`, je vérfie que les variables sont accessibles au bon moment.

```
| TE_merge (e, c_e_list) ->
    let ce = clock_expr e in
    let ck = ce.cexpr_clock in
    match ck with
    | ClockTuple [ClockBase (ClockVar (f, i))] ->
      begin
        let c_ce_list =
          List.map
          (fun (c, e') ->
            let ce' = clock_expr e' in
            let ck' = ce'.cexpr_clock in
            if (compatible true ck' (ClockTuple [ClockBase
                (ClockVar ((fun ck1 -> Sampled (f ck1, c, ce)), i))])
                !Node.in_notclocked false)
            then (c, ce')
            else (* des erreurs *)
          )
          c_e_list
        in
        CE_merge (ce, c_ce_list), ck
      end
    | _ ->  error loc (ExpectedClockVar ck)
```

5 Tests

Des tests ont été menés, montrant que le compilateur ne sait pas quand `ck on C(x)` est vraie ou non. En effet, je n'ai pas donné accès aux valeurs des variables à l'horloge. Ils sont listés dans le dossier `example`. Les tests avec en titre "notcompile" ne compile pas, soit parce que la synchronisation ne peut pas être assurée par le compilateur soit parce qu'elle n'est juste pas possible.

6 Améliorations

On pourrait faire un modelchecking afin de savoir si telle ou telle varaibles est accessible en fonction de sa valeur. On pourrait rendre les expressions ajoutées résistantes aux tuples. Et bien sûr il faudrait ajouter d'autres expressions.