

# OCaLustre : une extension synchrone d'OCaml pour la programmation de microcontrôleurs

Steven Varoumas<sup>1,2</sup>, Benoît Vaugon<sup>3</sup>, Emmanuel Chailloux<sup>1</sup>

<sup>1</sup>Laboratoire d'Informatique de Paris 6 (LIP6) — UPMC - Sorbonne Universités

<sup>2</sup>Centre d'Étude et De Recherche en Informatique et Communications (CÉDRIC) — Cnam

<sup>3</sup>Armadillo

5 janvier 2017



le cnam

# Contexte

## Les microcontrôleurs ( $\mu$ C)

- **Circuit intégré  $\simeq$  ordinateur simplifié** (unité de calcul, mémoires, broches d'entrées/sorties)
- Utilisé dans les **systèmes embarqués** parfois critiques (automobiles, trains, ...)
- Multiples **interactions avec l'environnement** (composants électroniques)



## Exemple : PIC 18F4620

- Architecture 8-bits
- 40 MHz
- RAM : 4 kio / Flash : 64 kio

⇒ Ressources limitées *mais* ils sont couramment utilisés par les industriels (faible coût et faible consommation énergétique)

# Programmer un $\mu$ C : l'approche machine virtuelle (VM)

## Approche classique

### Développement en C / en langage assembleur

- Programmation de bas niveau (pas d'abstraction matérielle)
- Peu de vérifications à la compilation (ex : typage)
- Parfois compliqué et source d'erreurs

Nous utilisons pour nos travaux une approche machine virtuelle :

## Approche VM

Portage d'une VM d'un langage pour exécution sur microcontrôleur :

- Permet l'exécution du bytecode de langages de plus haut niveau (programmation plus sûre, et plus simple)
- Offre une couche d'abstraction du matériel
- Augmente la portabilité du code
- Diminue souvent la taille du code

## OCaml

- Langage de programmation **multi-paradigmes** (fonctionnel, impératif, orienté objet)
- Langage de haut niveau avec des constructions **riches** et une **expressivité** accrue (foncteurs, types algébriques, lambdas, objets, exceptions ...)
- **Sûreté** augmentée par le typage statique avec inférence
- Sa machine virtuelle (la ZAM) est très **légère**



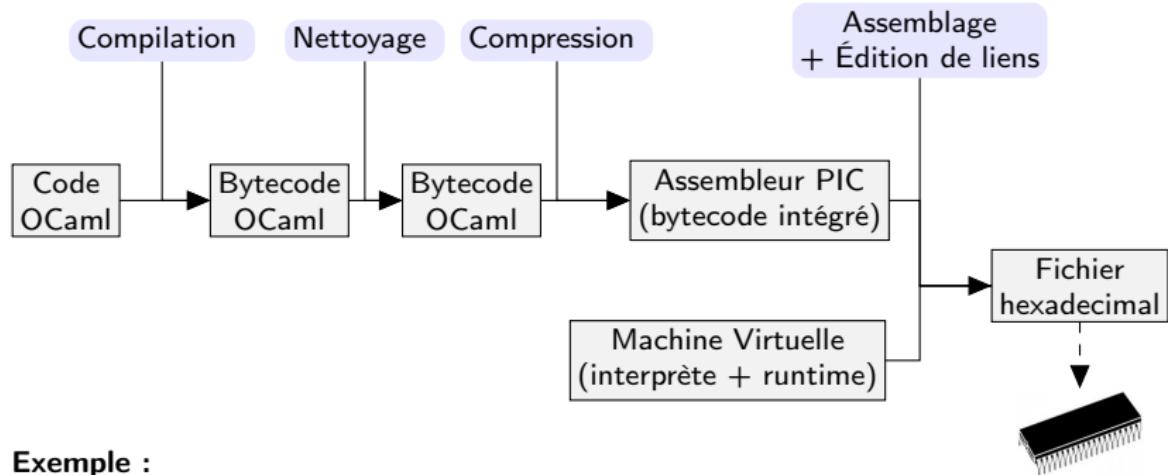
## OCaPIC : *OCaml for PIC*

- **Machine virtuelle OCaml** implantée en assembleur PIC18
- Permet l'exécution de tout le langage OCaml sur un  $\mu$ C PIC
- Fournie avec : simulateurs, bibliothèques pour composants externes ...
- *OCamlClean* : outil pour supprimer les allocations mémoires inutiles  $\rightarrow$  **bytecode plus petit**
- Gestion automatique de la mémoire (garbage collector)



– Benoit Vaugon, Philippe Wang et Emmanuel Chailloux. “Programming Microcontrollers in OCaml : the OCaPIC Project”. In : *International Symposium on Practical Aspects of Declarative Languages (PADL 2015)*

# OCaPIC : Compilation et exemple



## Exemple :

```
open Pic
open Lcd
(* connect the LCD to the correct pins: *)
let disp = connect ~bus_size:Lcd.Four ~e:LATD0 ~rs:LATD2 ~rw:LATD1 ~bus:PORTB;;
disp.init ();
disp.config ();
disp.print_string "Hello_world"
```

Hello world

# Extension à la programmation synchrone

Peut on aller plus loin ?

- Les microcontrôleurs doivent réagir à des stimuli externes
- Rapidement et dans n'importe quel ordre
- On aimerait un modèle de concurrence léger qui offre des garanties

## Le Paradigme Synchrone

- Les composants logiciels s'exécutent dans **le même instant logique**
  - ▷ Une horloge globale segmente le temps en instant discrets
- Les sorties sont considérées **simultanées** avec les entrées au sein d'un même instant (hypothèse synchrone)
- Concurrence déterministe (et modulaire)

→ Extension d'OCaml au paradigme de programmation synchrone

Flot de contrôle (Esterel/ReactiveML) ou flot de données (Lustre/Lucid Synchrone) ?

Le modèle flot de données paraît plus adapté :

- Chaque broche a une valeur à chaque instant → flot
- Représentable par des circuits (cf. Scade)
- Modèle de compilation léger

# OCaLustre : une extension synchrone à flot de données d'OCaml

## OCaLustre

- Un programme OCaLustre est composé de **fonctions OCaml et nœuds “Lustre”**
- Un nœud est un **composant synchrone** qui calcule *instantanément* des flots en sortie à partir de flots en entrée
- Le corps d'un nœud est **un système d'équations**
- Permet l'appel de fonctions OCaml depuis le monde synchrone (`call expr`)

```
let%node exemple ~i:(x,y,b,d) ~o:(u,v) =
  u = 3 * x + w;
  w = if (b && d) then (10 * y) else 42;
  v = 4 - y
```

NB1 : Toutes les valeurs sont des flots :

- La variable `x` correspond au flot :  $(x_0, x_1, \dots, x_i, \dots)$
- Une constante comme `3` est aussi un flot :  $(3, 3, \dots, 3, \dots)$

NB2 : Pas besoin d'annotations de types (inférés à la compilation par OCaml)

# OCaLustre : Opérateurs temporels (décalage)

Les équations sont définies avec des opérateurs arithmétiques et logiques (+,-, `&&` , `||`, ...) , ainsi que des opérateurs qui font référence au temps :

- Retard initialisé :

$a \rightarrow> b \equiv (a_0, b_0, b_1, b_2, \dots, b_i, \dots)$

Exemple : entiers naturels

```
let%node nat ~i:() ~o:(n) =
  n = 0 ->> (n + 1)
```

Exemple : Fibonacci

```
let%node fibonacci ~i:() ~o:(f) =
  k = 1 ->> f;
  f = 0 ->> (f + k)
```

instant	0	1	2	3	4	5	...
k	1	0	1	1	2	3	...
f	0	1	1	2	3	5	...
$f + k$	1	1	2	3	5	8	...

## OCaLustre : Opérateurs temporels (horloges)

Par défaut, tous les flots sont exécutés au même instant. Par exemple dans :

```
x = if b then e1 else e2
```

e1 et e2 sont calculés à chaque instant (pas d'évaluation paresseuse du if)

La notion de multi-horloge permet de récupérer le "contrôle" :

- Les opérateurs `@whn` et `@whnot` permettent de ralentir le calcul (échantillonage) :

▷  $x = a @whn ck \Leftrightarrow$  si  $ck = \text{true}$  alors  $x = a$ , sinon  $x$  n'a pas de valeur  
→  $x$  est sur l'horloge  $ck$

▷  $y = b @whnot ck \Leftrightarrow$  si  $ck = \text{false}$  alors  $y = b$ , sinon  $y$  n'a pas de valeur  
→  $y$  est sur l'horloge (*not ck*)

Les opérateurs OCaLustre ne peuvent être appliqués que sur des flots sur la même horloge, sauf :

- L'opérateur `merge` qui combine deux flots sur des horloges opposées :

▷  $z = \text{merge } ck \ x \ y \Leftrightarrow$  si  $ck = \text{true}$  alors  $z = x$  , sinon  $z = y$   
→  $z$  est sur l'horloge **de**  $ck$

## OCaLustre : horloges

ck	true	false	true	true	false	false	true
a = 4 @whn ck	4		4	4			4
b = 2 @whn ck	2		2	2			2
c = a + b	6		6	6			6
d = -1 @whnot ck		-1			-1	-1	
e = merge ck c d	6	-1	6	6	-1	-1	6

Le type des horloges est inféré à la compilation :

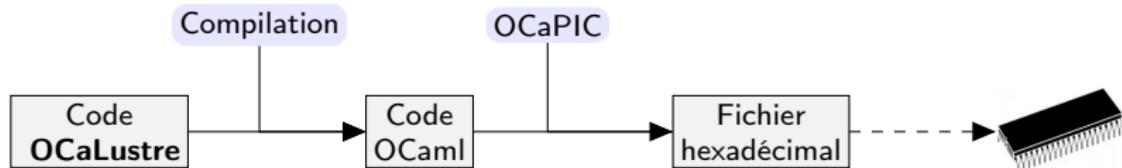
```
let%node merger ~i:(ck,x,y) ~o:(z) =
  z = merge ck x y
```

a pour type d'horloge :  $\forall ck1. \forall \alpha. ((ck1 : \alpha) \times (\alpha \text{ on } ck1) \times (\alpha \text{ on } (\text{not } ck1))) \rightarrow \alpha$

Jean-Louis Colaço et Marc Pouzet. "Clocks as First Class Abstract Types". In : *Embedded Software : Third International Conference, EMSOFT 2003, Philadelphia, PA, USA, October 13-15, 2003. Proceedings*

## Modèle de compilation

- Chaque nœud est compilé vers une fonction OCaml
- Le code OCaml résultant peut être utilisé avec OCaPIC (ou tout autre compilateur OCaml)
- 4 étapes : normalisation, ordonnancement, inférence d'horloges, **génération de code**  
cf. [Dariusz Biernacki et al. "Clock-directed Modular Code Generation for Synchronous Data-flow Languages". In : SIGPLAN Not. \(2008\)](#)



# OCaLustre : compilation (génération de code)

## Génération de code

- Un nœud devient une fonction qui retourne une **fermeture**
- Les équations deviennent des déclarations de variables
- Les opérateurs arithmétiques et logiques ne changent pas
- $\rightarrow\!\!\!>$  entraîne la création d'une référence dans le contexte de la fermeture
- Si  $x = e$  est sur l'horloge  $c$  alors il devient  $\text{if } c \text{ then } x := e$

```
let%node nat ~i:() ~o:(n) =
  n = 0 ->> (n + 1)
```

devient :

```
let nat () =
  let st_n = ref 0 in
  let nat_step () =
    let n = !st_n in
    st_n := n + 1;
    n
  in nat_step
```

## OCaLustre : Exemple : tempéreuse à chocolat

```
(* La température en celsius est (1033-ctemp)/11.67 *)
let%node update_prop ~i:(wtemp,ctemp) ~o:(prop) =
  new_prop = 0 ->> (call (min 100 (max 0 (new_prop + offset))));
  delta = call (min 10 (max (-10) (ctemp-wtemp)));
  offset = call (min 10 (if delta < 0 then (-delta*delta) else (
    delta*delta)));
  prop = new_prop / 10

let%node timer ~i:(number) ~o:(alarm) =
  time = 1 ->> if time = 10 then 1 else time + 1;
  alarm = (time < number)

let%node heat ~i:(w,c) ~o:(h) =
  prop = update_prop (w,c);
  h = timer (prop)

let%node change_wtemp ~i:(default,state) ~o:(w) =
  w = default ->> (
    if state = SlowPlus then w - 1
    else if state = FastPlus then w - 3
    else if state = SlowMinus then w + 1
    else if state = FastMinus then w + 3
    else w )

let%node thermo_on ~i:(state) ~o:(on) =
  on = true ->> (if state = OnOff then not on else on)
```



## OCaLustre : Exemple : tempéreuse à chocolat

```
let%node main ~i:() ~o:() =
  plus = call (test_bit plus_button);
  minus = call (test_bit minus_button);
  state = call ( buttons_state plus minus );
  on = thermo_on (state);
  t = if on then call (read_temp ()) else 0;
  wtemp = if on then change_wtemp (call (load_temp ()),state) else 0;
  h = if on then heat (wtemp, ctemp) else false;
  p = call (print_temps (wtemp,t));
  r = if heat then call (set_bit output) else call (clear_bit output)

let _ =
  let main_step = thermo () in
  while true do
    main_step ();
  done
```

Langage	OCaml	OCaLustre
Taille du programme PIC	35,91 kio	36,25 kio
Allocation mémoire à l'initialisation	1,96 kio	1,99 kio

## OCaLustre : Exemple : tempéreuse à chocolat (avec horloges)

```
let%node thermo ~i:() ~o:() =
  plus = call (test_bit plus_button);
  minus = call (test_bit minus_button);
  state = call (buttons_state plus minus);
  on = thermo_on (state);
  t = call (read_temp ()) @whn on;
  wtemp = change_wtemp ((call (load_temp ())),state) @whn on;
  h = heat (wtemp, t);
  p = call (print_temps (wtemp,t)) @whn on;
  r = merge h
    (call (set_bit output)) @whn h
    (call (clear_bit output)) @whnot h

let _ =
  let main_step = thermo () in
  while true do
    main_step ();
  done
```

## Conclusion :

### Ce qu'on gagne :

- **Légèreté** du modèle
- **Modèle adapté aux applications pour microcontrôleur**
- **Expressivité** de la programmation
- **Sûreté** : typage statique, vérification de boucles de causalité ...
- **Portabilité**

⇒ Semble bien adapté aux systèmes embarqués

### Travaux en cours & futurs

- **Contrats** pour les noeuds (triplets de Hoare vérifiés dans Why3)
- **Propriétés** sur les composants électroniques
- **Preuve** de la compilation
- **Applications concrètes** dans la robotique, la domotique, ...
- **Ciblage vers d'autres microcontrôleurs** (Atmel/Arduino, ARM/Nucléo)