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

Steven Varoumas[1], Benoît Vaugon[2] & Emmanuel Chailloux[1]

1: Sorbonne Universités, UPMC Univ Paris 06, CNRS,
LIP6 UMR 7606, 4 place Jussieu 75005 Paris.
steven.varoumas@lip6.fr, emmanuel.chailloux@lip6.fr
2: Armadillo,
46 bis, rue de la République, 92170 Vanves,
benoit.vaugon@gmail.com

Résumé

Les microcontrôleurs sont des circuits intégrés programmables dont le domaine d'application se concentre essentiellement dans le contrôle d'objets interagissant avec leur environnement. En effet, les programmes exécutés sur microcontrôleurs ont souvent pour rôle de réagir rapidement avec les composants qui les entourent, et de modifier leurs signaux de sortie dès lors que les signaux qu'ils reçoivent en entrée changent. Ainsi, la programmation synchrone à flots de données semble être un candidat de choix pour le développement de tels systèmes qui peuvent être parfois critiques. Nous proposons alors une extension du langage multi-paradigmes OCaml offrant la possibilité de manipuler des nœuds synchrones inspirés du langage Lustre tout en conservant les caractéristiques avantageuses du langage hôte. Destinée à être exécutée sur des microcontrôleurs aux capacités mémoires limitées, cette extension produit un code OCaml de faible taille, et peu gourmand en ressources mémoire. Nos travaux tirent particulièrement profit de la machine virtuelle OCaPIC permettant l'exécution de code-octet OCaml sur les microcontrôleurs de la famille PIC18 qui possèdent des ressources matérielles particulièrement faibles.

1. Introduction

Un microcontrôleur est un circuit intégré comportant plusieurs composants semblables à ceux d'un ordinateur simplifié : une unité de calcul, des mémoires, ainsi que plusieurs dispositifs d'entrées/sorties permettant d'interagir avec l'environnement (généralement cet environnement est constitué de composants électroniques ou d'autres contrôleurs). On retrouve des microcontrôleurs dans de nombreux systèmes embarqués, qu'ils soient de simples appareils électroménagers, des objets de loisir ou de domotique (drones, jouets, objets connectés, ...), voire même d'importants systèmes critiques (automobile, avionique,...). Cette omniprésence s'explique par le faible coût de ces dispositifs, leur taille réduite, ainsi qu'à leur basse consommation électrique, permettant une certaine autonomie énergétique aux systèmes qui en sont dotés.

En contrepartie de ces avantages, les microcontrôleurs souffrent souvent d'une puissance de calcul limitée ainsi que de ressources mémoires restreintes (quelques kilo-octets de mémoire vive). Cela contraint généralement les développeurs de programmes embarqués à faire usage de modèles de programmation de bas niveau pour conserver un contrôle permanent sur l'utilisation des ressources matérielles disponibles : les microcontrôleurs sont donc traditionnellement programmés en langage assembleur ou dans des langages comme Basic ou des sous-ensembles du langage C.

Ces langages d'assez bas niveau n'offrent cependant pas la même abstraction matérielle, la même sécurité, et la même expressivité que plusieurs langages de plus haut niveau comme Python, Lisp,

Java, ou OCaml, c'est pourquoi plusieurs projets destinés à développer des machines virtuelles de langages de haut niveau capables d'être exécutées sur microcontrôleurs ont vu le jour : on peut citer la plateforme MicroEJ [13] et sa MicroJvm, qui permet d'exécuter du code-octet Java sur microcontrôleurs ARM Cortex-M, la Darjeeling Virtual Machine [4], un portage de la JVM pour microcontrôleurs Atmel AVR128 et MSP430, PICBIT [10] et PICOBIT [14] qui permettent d'exécuter des programmes Scheme sur des microcontrôleurs PIC, et enfin OCaPIC [15], une machine virtuelle OCaml pour microcontrôleur PIC18, que nous utilisons dans cet article.

Ces langages généralistes ne sont pour autant pas forcément adaptés à la programmation de microcontrôleurs : ceux-ci sont souvent en interaction continue avec leur environnement, prêts à réagir aux signaux électriques envoyés par les composants électroniques auxquels ils sont branchés. Un programme pour microcontrôleur consiste souvent en une boucle gérant l'interaction en produisant certaines valeurs en sortie à partir de signaux reçus, et cette réaction doit être rapide, et indépendante de l'ordre d'apparition de ces divers stimuli.

Pour cette raison, nous proposons d'étendre le langage OCaml avec des traits de programmation synchrone à flots de données. Cette extension, que nous appelons OCaLustre de par sa similarité avec le langage de programmation synchrone Lustre [6], nous paraît particulièrement adaptée à la nature des applications pour microcontrôleurs, et offre la possibilité de vérifier statiquement de nombreuses propriétés sur les programmes. En tant qu'extension, elle profite de tous les avantages du langage hôte, comme le typage statique ou la gestion automatique de la mémoire, et peut également accéder à des fonctions écrites directement en OCaml.

Dans cet article nous détaillons en section 2 les avantages liés à la machine virtuelle OCaPIC et au paradigme de programmation synchrone, afin d'en justifier leur utilisation commune dans nos travaux. La partie 3 présente la syntaxe et le fonctionnement d'OCaLustre. La partie 4 détaille la façon dont est compilé l'extension vers son langage hôte pour être alors traité par OCaPIC. La partie 5 présente un exemple de programme OCaLustre développé pour un microcontrôleur, ainsi que sa consommation mémoire. Enfin, la partie 6 est une discussion sur les futures améliorations pouvant être apportées à OCaLustre, ainsi que sur les divers travaux annexes qui peuvent tirer profit d'un tel langage.

2. Contexte de développement d'OCaLustre

Pour nos travaux, nous avons fait le choix d'utiliser des microcontrôleurs de la famille PIC18 produits par la société *Microchip*. Ces dispositifs ont l'intérêt d'être couramment utilisés à la fois par des électroniciens amateurs pour des petits projets de loisirs mais aussi par des industriels dans des applications parfois critiques [1]. Nos travaux tirent alors profit de la machine virtuelle OCaPIC qui permet de programmer des microcontrôleurs PIC18 en langage OCaml. Les PIC18 disposent de ressources particulièrement limitées (le modèle PIC18F4620 que nous utilisons dans nos expériences est doté d'une mémoire de masse de 64 kibioctets et de seulement 4 kibioctets de mémoire vive) mais cette limitation est justement intéressante pour nos travaux d'extension d'OCaml avec des traits de programmation synchrone : produire du code capable de s'exécuter sur des dispositifs avec si peu de ressources matérielles nous assure que ce même code pourrait être exécuté sur des microcontrôleurs plus riches en mémoire.

2.1. OCaPIC

OCaPIC[15] est un ensemble d'outils permettant d'exécuter du code OCaml sur les microcontrôleurs PIC de la famille PIC18. Il est principalement constitué d'une implémentation optimisée pour PIC de la machine virtuelle OCaml (la « ZAM »[12]) permettant d'évaluer le code-octet généré par

1. Citons par exemple leur utilisation dans le projet de carte industrielle LCHIP : `https://www.pole-scs.org/projet/lchip`

le compilateur `ocamlc`. Cette machine virtuelle est équipée d'une bibliothèque d'exécution contenant tous les utilitaires nécessaires à son exécution et en particulier des ramasse-miettes basés sur les algorithmes Stop&Copy et Mark&Compact.

Comme on peut le voir sur la figure 1, pour programmer un microcontrôleur PIC en OCaml via OCaPIC, le code source OCaml est d'abord compilé en code-octet via le compilateur standard `ocamlc`, il traverse ensuite différentes passes de nettoyage et de compression, puis est lié avec le code de la machine virtuelle avant d'être transféré sur le PIC via un programmateur.

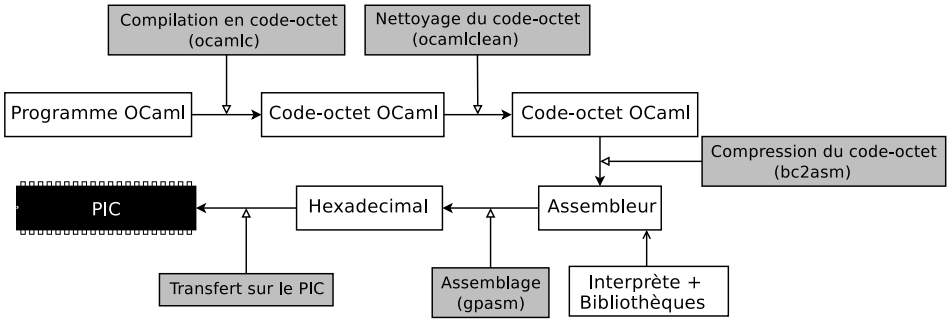

FIGURE 1 – Compilation d'un programme OCaml sur PIC

L'environnement d'un microcontrôleur est très différent de celui d'un ordinateur. Le principal objectif des programmes s'y exécutant est d'interagir avec le circuit électronique dans lequel il est connecté via des protocoles souvent de très bas niveau. OCaPIC est alors muni d'une bibliothèque dédiée aux PIC offrant une interface de haut niveau avec la configuration de ces microcontrôleurs et implémentant des protocoles de communication avec des composants externes classiques comme un afficheurs LCD ou un port série.

Le déboguage d'un programme sur un microcontrôleur est particulièrement délicat. Ceci est d'autant plus vrai lors de l'exécution par une machine virtuelle de code octet généré depuis OCaml. Pour permettre le déboguage des programmes OCaml sur PIC, OCaPIC fournit différents mécanismes de simulation permettant d'observer, sur un ordinateur, le comportement qu'aura le microcontrôleur après programmation.

Sur des architectures où la taille des mémoires est particulièrement limitée, la taille du code peut devenir critique. OCaPIC fournit alors un nettoyeur de code mort OCaml, nommé `ocamlclean`, permettant d'éliminer du code OCaml jamais utilisé (provenant typiquement de bibliothèques partiellement utilisées). Pour gagner plus encore en espace de code et en vitesse de lecture, le code-octet est compressé avant d'être transféré sur le PIC. Il est ainsi possible d'encoder une opération complexe en une seule instruction virtuelle stockée sur quelques octets seulement. Le facteur de compression obtenu est alors de l'ordre de 3,5 par rapport au code-octet d'origine.

Le langage OCaml offrant de nombreuses fonctionnalités (typage statique, fonctions de première classe, exceptions, objets, gestion automatique de la mémoire, etc.), il est un langage cible de choix pour l'implantation de paradigmes de programmation de plus hauts niveau encore, comme par exemple un modèle de programmation synchrone.

2.2. Le modèle de programmation synchrone

Le paradigme de programmation synchrone est un modèle de programmation adapté au développement de systèmes réactifs. Il repose sur un principe d'abstraction semblable à celui appliqué pour la modélisation de circuits : on considère généralement que le temps pris par l'électricité

pour traverser un fil, ou bien pour passer de l'entrée d'une porte logique à sa sortie, est nul. La programmation synchrone suit un principe équivalent, nommé *hypothèse synchrone* : on y considère que le temps pris par un programme pour calculer des valeurs de sortie à partir de valeurs en entrée est nul. Les entrées du programme sont alors vues comme instantanées avec ses sorties, et toutes les instructions du programme sont réalisées au sein d'un même instant logique. Pour ce faire, l'horloge globale du programme segmente le temps en intervalles (les instants) de manière à ce que tous les traitements dans un instant s'achèvent avant le début d'un prochain instant : chaque instant commence lors de l'apparition d'un «tick» logique émis par l'horloge globale, et toutes les valeurs de sortie sont calculées avant l'apparition du prochain «tick». On considère alors que les diverses tâches qui se déroulent pendant un instant s'exécutent en même temps. Ce modèle de programmation permet de simplifier la tâche du programmeur en supprimant les considérations temporelles d'exécution des instructions des programmes, et offre un modèle totalement déterministe permettant une analyse formelle des programmes en vue de la vérification de leur propriétés et de la certification du code généré.

Il est de ce fait adéquat d'étendre les capacités d'OCaPIC pour gérer un tel modèle de programmation. Notre solution prend alors la forme d'une extension du langage OCaml permettant d'y ajouter des traits de programmation synchrone, nommée OCaLustre.

3. OCaLustre

OCaLustre[2] s'inspire fortement des langages à flots de données synchrones Lustre [6] et Lucid Synchrone [5]. On s'applique en particulier à conserver avec OCaLustre la même sémantique que le langage Lustre, représentable par des réseaux de Kahn [7]. OCaLustre est transformé lors de la compilation en code OCaml classique, permettant d'utiliser le code généré dans OCaPIC (ou tout autre compilateur ou machine virtuelle OCaml) sans aucune modifications de celui-ci. La figure 2 décrit la chaîne de compilation d'un code OCaLustre vers un fichier hexadécimal destiné à un microcontrôleur PIC.

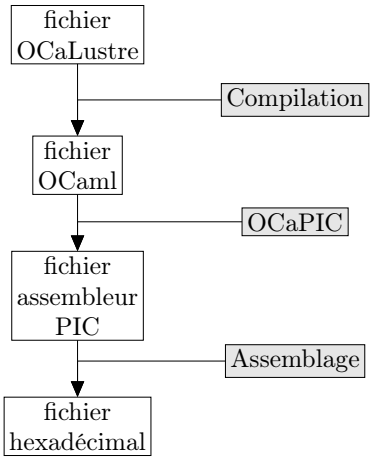

Figure 2 – Chaîne de compilation d'OCaLustre pour microcontrôleurs PIC

Il est à noter qu'en tant qu'extension au langage OCaml, OCaLustre permet un modèle de pro-

2. Distribué sous licence libre sur `https://github.com/stevenvar/OCaLustre`

grammation mixte : il est en effet possible de faire appel à des fonctions OCaml depuis l'extension synchrone, cela permet ainsi de gérer plus aisément les parties algorithmiques des programmes réalisés, et profiter ainsi de la richesse d'expressivité d'OCaml.

Précisons également que notre choix de ne pas réutiliser le compilateur de Lucid Synchrone, alors que le langage a de nombreux points communs avec OCaLustre, repose principalement sur le fait qu'il nous est important de contrôler le compilateur du langage afin de générer un code compact qui soit le moins gourmand en mémoire possible. Cette économie de ressources est un aspect primordial dans le cadre de la programmation de microcontrôleurs, et il est important de la conserver. De plus, OCaLustre est amené à être étendu afin de permettre de vérifier des propriétés sur les programmes créés, et nous avons également pour ambition de certifier notre modèle de compilation. Pour ces raisons, un contrôle complet du langage et de son implémentation paraît essentiel.

3.1. Flots de données

En OCaLustre, toutes les valeurs manipulées sont des flots : des séquences de valeur qui sont susceptibles de varier au cours du temps. Ainsi, une variable nommée x correspond à la séquence de toutes les valeurs que prend x d'instant en instant :

$$x \equiv (x_0, x_1, x_2, x_3, \ldots, x_i, \ldots)$$

Une constante est considérée comme une séquence dont les valeurs ne changent pas :

$$2 \equiv (2, 2, 2, 2, \ldots, 2, \ldots)$$

Les opérateurs d'OCaLustre s'appliquent point à point sur les valeurs des flots :

$$\begin{aligned} \text{si } x &\equiv (x_0, x_1, x_2, \ldots, x_i, \ldots) \\ \text{et si } y &\equiv (y_0, y_1, y_2, \ldots, y_i, \ldots) \\ \text{alors } x \Diamond y &\equiv (x_0 \Diamond y_0, x_1 \Diamond y_1, x_2 \Diamond y_2, \ldots, x_i \Diamond y_i, \ldots) \\ &\text{pour } \Diamond \text{ un opérateur binaire quelconque} \end{aligned}$$

Ces flots sont définis à l'intérieur de *nœuds* : des fonctions synchrones qui sont exécutées à chaque instant logique du programme. La signature d'un nœud commence par le mot-clé `let%node`, suivi du nom du nœud, d'un paramètre labellisé `~i` correspondant aux flots d'entrée du nœud et d'un second paramètre labellisé `~o` qui correspond à ses flots de sortie. Le corps d'un nœud est un système de définitions de flots qui sont calculées à chaque instant. Ces diverses définitions sont des équations de la forme *pat* = *exp*, avec *pat* le nom du flot, et *exp* une expression correspondant à sa valeur.

Par exemple, le nœud `plus_moins` reçoit deux flots et retourne un couple qui contient la somme et la différence de ses entrées :

```
let%node plus_moins ~i:(x,y) ~o:(p,m) =
  p = x + y;
  m = x - y
```

Une expression peut être une simple valeur v, une variable de flot x, une alternative, un tuple d'expressions, l'application d'un opérateur arithmétique ou logique, un appel à un autre nœud, l'exécution de code OCaml (avec l'instruction `call`) ou l'application d'un opérateur temporel. La figure 3 décrit la grammaire complète d'un nœud OCaLustre.

noeud ::= `let%node` *node_id* `~i:` **pat** `~o:` **pat** `=` **seq**
pat ::= *id* | `()` | **pat**`,` **pat**
seq ::= **decl** | **decl**`;` **seq**
decl ::= **pat** `=` **exp**
infix ::= `+` | `-` | `*` | `/` |`+.` | `-.` | `*.` | `/.` | `<` | `>` | `<=` | `>=` | `=` | `<>` | `&&` | `||`
prefix ::= `not` | `-`
exp ::= `()` | *v* | *id* | `if` **exp** `then` **exp** `else` **exp** | **exp infix exp** | **exp** `->` **exp**
| **prefix exp** | **exp**`,` **exp** | *node_id* **exp** | `call` *ml_exp* | **exp** `@whn` *id*
| **exp** `@whnot` *id* | `merge` *id* **exp exp**

FIGURE 3 – Syntaxe d'OCaLustre

3.2. Opérateurs temporels

Aux opérateurs arithmétiques et logiques classiques, on ajoute plusieurs opérateurs qui dépendent du temps :

— L'opérateur de décalage `->` (nommé `fby` pour « followed-by » dans Lucid Synchrone et Scade [2]) : l'équation `z = `e_1 `->` e_2 définit un flot `z` qui à l'instant 0 a pour valeur l'expression e_1, puis à chaque instant i celle de l'expression e_2 à l'instant $i-1$:

$$\begin{aligned} \text{si } e_1 &\equiv (x_0, x_1, x_2, x_3, \dots, x_i, \dots) \\ \text{et } e_2 &\equiv (y_0, y_1, y_2, y_3, \dots y_i, \dots) \\ \text{alors } e_1\texttt{->>}e_2 &\equiv (x_0, y_0, y_1, y_2, y_3, \dots, y_i, \dots) \end{aligned}$$

Dans l'exemple suivant, on définit grâce à cet opérateur le nœud `count` qui retourne au i-ème instant l'entier $x \equiv n + i$:

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

— Les opérateurs d'échantillonnage `@whn` et `@whnot` (pour « *when* » et « *when not* ») permettent de ralentir le calcul d'une expression : dans l'équation `z = `e `@whn` c l'évaluation de e n'a lieu que quand le flot booléen c a pour valeur `true`, et on dit alors que l'équation est sur l'horloge c. Si e est sur une horloge de type α alors le type de l'horloge de `z` est α *on* c (ce système de type est issu de [8]).
À l'inverse, e `@whnot` c est évalué uniquement lorsque c vaut `false`, et l'équation est sur l'horloge *not* c. Son type d'horloge est alors α *on* (*not* c)

On ne peut utiliser les opérateurs d'OCaLustre que sur des flots sur la même horloge, et le flot résultant est alors lui-même sur cette horloge :

```
let%node sampler ~i:(c,x,y) ~o:(s) =
  a = x @whn c;
  b = y @whn c;
  s = a + b
```

a pour type d'horloge : $\forall\alpha.\forall ck_1.(ck_1 : \alpha) * \alpha * \alpha \rightarrow \alpha\ on\ ck_1$ (la notation $(ck_1 : \alpha)$ indiquant que la variable d'horloge nommée ck_1 est elle-même sur une horloge α).

— L'opérateur de fusion `merge` permet générer un flot à partir de deux flots sur des horloges complémentaires : `merge` c e_1 e_2 produit un flot qui vaut la valeur de e_1 quand c est vrai et celle de e_2 lorsque c est faux. Il faut donc que e_1 soit sur horloge c et que e_2 soit sur $not\ c$. Le flot résultant est alors sur l'horloge de c.

Par exemple, le nœud suivant :

```
let%node merger ~i:(ck,tic,tac) ~o:(tictac) =
  tictac = merge ck tic tac
```

a pour type d'horloge : $\forall\alpha.\forall ck_1.(ck_1 : \alpha) * \alpha\ on\ ck_1 * \alpha\ on\ (not\ ck_1) \rightarrow \alpha$

4. Compilation d'OCaLustre

La compilation d'un nœud OCaLustre est modulaire, et s'inspire de [3]. On s'approche du modèle de compilation « en boucle simple » du langage Lustre : tout nœud OCaLustre est compilé vers une fonction séquentielle OCaml et le nœud principal du programme est exécuté dans une boucle infinie.

Ce processus de compilation se déroule en quatre étapes : normalisation, ordonnancement, inférence d'horloges, et enfin génération de code OCaml (et de la machine d'exécution dans la situation où le noeud compilé est le noeud principal du programme).

4.1. Normalisation

La compilation d'un nœud OCaLustre passe d'abord par une étape consistant à normaliser les définitions de flots. Lors de cette étape, on extrait les expressions dépendant de l'état interne d'un nœud (c'est-à-dire celles définies avec l'opérateur `->` et les appels à d'autres nœuds) qui apparaissent dans d'autres expressions.

Ainsi, le nœud

```
let%node call ~i:(b) ~o:(x) =
  x = if b then (count 12) else (count 32)
```

est transformé en

```
let%node call ~i:(b) ~o:(x) =
  aux1 = count 12;
  aux2 = count 32;
  x = if b then aux1 else aux2
```

Cette traduction permet de conserver la sémantique du langage Lustre : l'évaluation du `if ␣ then ␣ else ␣` n'est donc pas paresseuse car ses deux branches doivent s'exécuter sur la même horloge. Les deux instances de `count` sont alors exécutées chaque fois que `call` est appelée.

4.2. Ordonnancement

En OCaLustre, l'ordre d'apparition des définitions de flots dans un nœud n'importe pas : ainsi, on peut définir un flot avant même de donner la définition des flots dont il dépend. Afin de permettre la génération d'un code OCaml correct, il est nécessaire de réordonner ces différentes définitions pour

qu'on ne fasse jamais référence à un flot qui n'a pas été défini. Cette étape d'ordonnancement est réalisée simplement grâce au tri topologique d'un graphe représentant les dépendances entre les flots. Ce tri retourne l'ordre correct dans lequel doit apparaître chaque définition, et permettra la génération d'une séquence d'assignations de variables correspondant à la définition de chaque flot. Ce processus permet également, dès lors qu'un cycle dans le graphe de dépendances est détecté, de rejeter toute boucle de causalité au sein d'un nœud : ainsi, si deux flots dépendent l'un de l'autre dans le même instant, le programme est refusé, et le compilateur retourne une erreur.

4.3. Inférence d'horloge

Lors de la compilation, chaque équation est annotée avec son horloge. Cette annotation, semblable à une annotation de type, permet de vérifier la cohérence de la combinaison de flots : deux flots ne peuvent être utilisés dans la même expression qu'à condition qu'ils soient sur la même horloge (à l'exception de l'opérateur de fusion `merge` qui combine des flots d'horloges opposées). L'annotation des flots permet de générer un type d'horloge pour les nœud, et cette information, essentielle lors de l'étape de génération de code pour conditionner l'exécution de certains appels, peut alors être utilisée lorsque un nœud fait appel à un autre nœud. Le mécanisme d'inférence permettant d'annoter chaque flot avec une horloge est équivalent à un algorithme d'inférence de type dans un langage de programmation à la ML [9] et les types des horloges respectent les contraintes énoncées dans [8]. Toute incohérence dans le typage d'horloges est détectée pendant cette phase d'inférence et provoque une erreur.

4.4. Génération de code

L'étape de génération de code consiste à produire du code OCaml à partir d'une définition de nœud. Puisqu'un nœud conserve un état interne (qui correspond à la fois aux valeurs précédentes des flots définis avec l'opérateur `->>` et aux diverses instances d'autres nœuds) nous avons choisi de générer pour chaque nœud OCaLustre une fonction OCaml qui retourne une fermeture. Celle-ci (dont le nom a le suffixe `_step`), correspond à une fonction d'avancement qui est exécutée à chaque instant. Le contexte de cette fermeture correspond à l'état interne du nœud, qui est ainsi conservé entre chaque appel.

Pour chaque équation de la forme `x` $= e_1$ `->>` e_2, on génère un registre `st_x` dans le contexte de la fermeture ayant pour valeur e_1, et on met à jour `st_x` avec la valeur de e_2 à la fin de la fonction d'avancement.

Pour chaque appel à un autre nœud, on génère une instance de celui-ci en faisant appel à la fonction du même nom. Cette fonction retourne la fonction d'avancement du nœud qui sera exécutée dans celle de l'appelant.

Par exemple, à partir du nœud :

```
let%node caller ~i:(init) ~o:(x,y) =
  x = init ->> (x + 1);
  y = count x
```

On génère la fonction OCaml suivante :

```
let caller init =
  let st_x = ref init in
  let x = !st_x in
  let count1_step = count x in
  let caller_step init =
```

```
  let x = !st_x  in
  let y = count1_step x
  in st_x := (x + 1);
  (x, y)  in
caller_step
```

La compilation des opérateurs `@whn` et `@whnot` est plus particulière : ces opérateurs contraignent l'exécution de l'expression qu'ils sous-échantillonnent : par exemple dans `x = (f 2) @whn c` l'exécution de `f 2` ne doit se faire que lorsque l'horloge `c` est vraie, afin de ne pas modifier l'état de `f` à une allure plus rapide que son horloge. On génère donc une condition sur la valeur de `c` : `let x = if c then f 2 else` *nil*. Pour respecter les contraintes du système de type d'OCaml, il est important de donner une valeur « bouche-trou » à un flot lorsque l'horloge est fausse. Nous la notons *nil*, bien que cette valeur n'existe pas en OCaml. Pour l'implémentation, nous pourrions utiliser la valeur polymorphe `Obj.magic ()`, définir les flots échantillonnés comme des valeurs de type option, ou même demander au programmeur de définir une valeur par défaut (inutilisée pour les calculs) en l'absence du flot. Notre implémentation utilise la valeur `Obj.magic ()`, mais elle peut être amenée à évoluer en fonction des contraintes liées à l'extension et la certification d'OCaLustre.

L'appel à un autre nœud multi-horloges nécessite également d'être contraint par les horloges de ses flots de sortie : en effet, lorsqu'on fait par exemple appel à un nœud qui a une horloge de type $((c : \alpha) * \alpha) \rightarrow \alpha$ *on* c il est important de vérifier que la valeur de c est vraie. Ainsi, on s'assure que les valeurs retournées par l'appel de la fonction sont bien présentes, et elles peuvent alors être utilisées dans le reste du nœud.

Les schémas suivants décrivent la façon dont est généré le code OCaml à partir de la forme de chaque construction du langage OCaLustre. La notation $[\![e]\!]_n^{ck}$ représente la compilation d'une expression e d'horloge ck dans le contexte de la définition d'un flot nommé n. La fonction $Init_{eq}$ permet de générer un registre pour chaque flot défini avec `->>` ainsi qu'une instance pour chaque nœud appelé. $Init_{seq}$ itère $Init_{eq}$ sur une séquence d'équations. La fonction $Update_{eq}$ met à jour chaque registre par la fonction $Init_{eq}$ avec la valeur adéquates. $Update_{seq}$ itère $Update_{eq}$ sur une séquence d'équations. Ces fonctions sont à considérer comme des macros dans les schémas suivants.

— Définitions globales :

$$
\begin{aligned}
&[\![\texttt{let\%node}\ node\_id\ \sim\texttt{i}:ins\ \sim\texttt{o}:outs = \ seq]\!]_{\_}^{ck_{in} \rightarrow ck_{out}} \rightarrow \\
&\quad \texttt{let}\ node\_id\ [\![ins]\!]_{\_}^{ck_{in}} = \\
&\qquad Init_{seq}(seq) \\
&\qquad \texttt{let}\ node\_id\_step\ [\![ins]\!]_{\_}^{ck_{in}} = \\
&\qquad\quad [\![seq]\!]_{\_}^{cks} \\
&\qquad \texttt{in}\ Update(seq) \\
&\qquad [\![outs]\!]_{\_}^{ck_{out}} \\
&\quad \texttt{in}\ node\_id\_step
\end{aligned}
$$

avec

$$
\begin{array}{lll}
Init_{seq}(eq\texttt{;}\ seq) & = & Init_{eq}(eq)\ \ Init_{seq}(seq) \\
 & & \text{(avec } eq \text{ de la forme } \textbf{\textit{pat}} = \textbf{\textit{exp}} \text{ et } seq \text{ une séquence d'équations)} \\
Init_{eq}(p\ \texttt{=}\ e_1\ \texttt{->>}\ e_2) & = & \texttt{let}\ \ \texttt{st\_}p\ \texttt{=}\ \texttt{ref}\ [\![e_1]\!]_p\ \ \texttt{in} \\
Init_{eq}(p\ \texttt{=}\ node\_id\ e) & = & \texttt{let}\ \ node\_id\_\texttt{step} = node\_id\ [\![e]\!]_p\ \ \texttt{in}
\end{array}
$$

(Les autres formes d'équations ne produisent pas de code généré)

et

$$
\begin{array}{lll}
Update_{seq}(eq\texttt{;}\ seq) & = Update_{eq}(eq)\ \ Update_{seq}(seq) \\
 & \text{(avec } eq \text{ de la forme } \textbf{\textit{pat}} = \textbf{\textit{exp}} \text{ et } seq \text{ une séquence d'équations)} \\
Update_{eq}(p\ \texttt{=}\ e_1\ \texttt{->>}\ e_2) & = \texttt{st\_}p\ \texttt{:=}\ [\![e_2]\!]_p\ \texttt{;}
\end{array}
$$

(Les autres formes d'équations ne produisent pas de code généré)

— Définitions locales :

$$
\begin{array}{ll}
[\![eq\texttt{;}\ seq]\!]^{ck;cks}_{\_} & \rightarrow [\![eq]\!]^{ck}_{\_}\ \ \texttt{in}\ \ [\![seq]\!]^{cks}_{\_} \\
 & \text{(avec } eq \text{ de la forme } \textbf{\textit{pat}} = \textbf{\textit{exp}}, \textit{ seq une séquence d'équations} \\
 & \text{et } ck; cks \text{ une séquence d'horloges)} \\
[\![p\ \texttt{=}\ e]\!]^{ck}_{\_} & \rightarrow \texttt{let}\ \ p\ \texttt{=}\ [\![e]\!]^{ck}_{p}
\end{array}
$$

— Expressions :

$$
\begin{array}{ll}
[\![v]\!]^{ck}_{\_} & \rightarrow v \\
[\![id]\!]^{ck}_{\_} & \rightarrow id \\
[\![e_1\ \texttt{,}\ e_2]\!]^{ck_1 * ck_2} & \rightarrow [\![e_1]\!]^{ck_1}\ \texttt{,}\ [\![e_2]\!]^{ck_2} \\
[\![e_1\ \textbf{infixop}\ e_2]\!]^{ck} & \rightarrow [\![e_1]\!]^{ck}\ \textbf{infixop}\ [\![e_2]\!]^{ck} \\
[\![\textbf{prefixop}\ e]\!]^{ck} & \rightarrow \textbf{prefixop}\ [\![e]\!]^{ck} \\
[\![\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3]\!]^{ck} & \rightarrow \texttt{if}\ [\![e_1]\!]^{ck}\ \texttt{then}\ [\![e_2]\!]^{ck}\ \texttt{else}\ [\![e_3]\!]^{ck} \\
[\![e_1\ \texttt{->>}\ e_2]\!]^{ck}_{n} & \rightarrow \texttt{!st\_}n \\
[\![node\_id\ e]\!]^{ck\ on\ id}_{\_} & \rightarrow \texttt{if}\ [\![id]\!]^{ck}_{\_}\ \texttt{then}\ node\_id\_\texttt{step}\ [\![e]\!]^{ck}_{\_}\ \texttt{else}\ nil \\
[\![node\_id\ e]\!]^{ck}_{\_} & \rightarrow node\_id\_\texttt{step}\ [\![e]\!]^{ck}_{\_} \\
[\![\texttt{call}\ ml\_exp]\!]^{ck} & \rightarrow ml\_exp \\
[\![e\ \texttt{@whn}\ id]\!]^{ck} & \rightarrow \texttt{if}\ [\![id]\!]^{ck}_{\_}\ \texttt{then}\ [\![e]\!]^{ck}_{\_}\ \texttt{else}\ nil \\
[\![e\ \texttt{@whnot}\ id]\!]^{ck} & \rightarrow \texttt{if}\ \texttt{(not}\ [\![id]\!]^{ck}_{\_}\texttt{)}\ \texttt{then}\ [\![e]\!]^{ck}_{\_}\ \texttt{else}\ nil \\
[\![\texttt{merge}\ id\ e_1\ e_2]\!]^{ck} & \rightarrow \texttt{if}\ [\![id]\!]^{ck}_{\_}\ \texttt{then}\ [\![e_1]\!]^{ck\ on\ id}_{\_}\ \texttt{else}\ [\![e_2]\!]^{ck\ on\ (not\ id)}_{\_}
\end{array}
$$

4.5. Génération de la machine d'exécution

La dernière phase de compilation d'OCaLustre consiste à générer une machine d'exécution. Celle-ci consiste simplement en une fonction qui exécute une boucle infinie faisant appel au nœud principal du

programme. Il est alors important que ce nœud ait pour type *unit* → *unit* pour pouvoir l'initialiser automatiquement. Les communications avec l'environnement ont lieu en faisant appel depuis le corps du nœud à des fonctions qui accèdent aux valeurs des broches du microcontrôleur, ainsi qu'à des fonctions qui peuvent les modifier. L'horloge de ce nœud représente l'horloge globale du programme, et un tour de boucle correspond alors à un instant d'exécution pour tous les nœuds définis sur celle-ci.

Par exemple, si le nœud principal du programme OCaLustre se nomme `main`, alors la fonction correspondant à la machine d'exécution sera :

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

5. Exemple d'application d'une réalisation complète

À titre d'exemple, nous avons développé en OCaPIC un programme pour une tempéreuse permettant de faire chauffer du chocolat au bain-marie à une température donnée. Le montage électronique correspondant comprend un microcontrôleur PIC18F4620 relié à deux boutons (l'un pour augmenter la température désirée, l'autre pour la baisser) ainsi qu'à plusieurs résistances permettant de chauffer l'eau, un thermomètre pour connaître la température actuelle de l'eau, et un écran à cristaux liquides (LCD) pour afficher les températures désirée et réelle.

Notre programme contrôle la tempéreuse en calculant une valeur `prop` représentant la quantité de temps pendant laquelle les résistances doivent être alimentées. Cette valeur dépend de la différence entre la température désirée (`wtemp`) et la température actuelle de l'eau (`ctemp`).

Le nœud `main` récupère à chaque instant la valeur des boutons + et − ainsi que la valeur de la température transmise par le thermomètre, et calcule à partir de celles-ci la température désirée, l'état de la tempéreuse (allumée ou éteinte) ainsi qu'un booléen indiquant si les résistances doivent chauffer ou non.

Nous donnons dans l'extrait de code suivant les nœuds OCaLustre qui constituent cette application. Il est à noter que le code concernant l'affichage sur l'écran LCD ainsi que la sauvegarde et le chargement de la valeur de la température désirée est directement rédigé en OCaml et n'est par souci de concision pas retranscrit ici. La figure 4 montre l'exécution du programme généré dans le simulateur inclus dans OCaPIC.

```
    (* La temperature 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))

let%node save_t ~i:(w) ~o:(save) =
  pre_w = 0 ->> w;
  changed = false ->> (w <> (pre_w));
  save = if changed then call (save_temp w) else ()

let%node thermo ~i:(plus,minus,ctemp) ~o:(wtemp, on, h) =
  state = call ( buttons_state plus minus );
  on = thermo_on (state);
  wtemp = if on then change_wtemp ( (call (load_temp ()) ),state) else 0;
  h = if on then heat (wtemp, ctemp) else false;
  save = save_t wtemp

let%node main ~i:() ~o:() =
  plus = call (test_bit plus_button);
  minus = call (test_bit minus_button);
  t = call (read_temp ());
  (wtemp,on,heat) = thermo (plus,minus,t);
  p = call (print_temps (wtemp,t));
  r = if heat then (call (set_bit output)) else (call (clear_bit output))
```

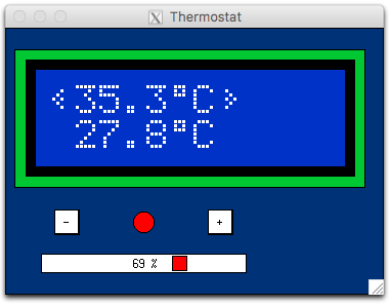

FIGURE 4 – Exécution du programme de la tempéreuse sur le simulateur d'OCaPIC

Par le passé une telle application pour OCaPIC avait déjà été développée directement en OCaml, à des fins de loisir. Son existence nous permet de comparer la taille de notre programme OCaLustre avec celui d'une application au comportement équivalent :

Language	OCaml		OCaLustre	
Type	Code-octet généré	programme chargé sur le PIC	Code-octet généré	programme chargé sur le PIC
Taille du programme	305,75 kio	36,25 kio	305,07 kio	35,91 kio
Allocation mémoire à l'initialisation du programme	-	1,96 kio	-	1,99 kio

6. Conclusion & travaux futurs

Nos expériences ont révélé que le code généré avec OCaLustre s'avère être de taille équivalente à celui d'un code écrit directement en OCaml. Ces résultats nous permettent de confirmer que notre modèle de compilation d'OCaLustre n'ajoute pas un coût supplémentaire au niveau de la taille des programmes ou de leur occupation mémoire au moment de leur initialisation. OCaLustre s'avère économe en mémoire, et offre un niveau de sûreté accru dans le développement de programmes tant par la vérification d'absence de boucle de causalité que par une analyse statique de la cohérence des horloges. Cette sûreté ne peut être acquise qu'à condition de certifier le modèle de compilation d'OCaLustre pour vérifier qu'il respecte bien la sémantique attendue. Nous avons donc pour ambition de certifier le compilateur OCaLustre à l'aide de l'assistant de preuve Coq comme cela avait été entamé pour le langage Lustre dans [1].

Il est à noter que bien qu'un programme puisse sembler sûr d'un point de vue de son typage, de la cohérence de ses horloges, et de son absence de causalité, il est néanmoins toujours susceptible d'avoir un comportement incohérent : en cas d'erreur dans le code de la tempéreuse les résistances pourraient par exemple chauffer lorsque la température actuelle dépasse la température voulue, et vice-versa. Il serait alors intéressant d'extraire, composer et vérifier des propriétés liées aux composants électroniques ou à leur environnement afin de s'assurer formellement que les programmes produisent les sorties attendues dès lors que leurs entrées respectent des conditions à définir. Nos travaux actuels portent en particulier sur la génération de code WhyML à partir d'un code OCaLustre afin de vérifier dans la plateforme Why3 [11] que chaque nœud respecte un contrat qui lui est soumis.

Ces divers travaux sont susceptibles d'entraîner la modification de l'implémentation d'OCaLustre, et il nous est donc essentiel d'avoir le contrôle sur le langage d'entrée et son compilateur. Il nous paraît également important d'étudier de nombreux exemples d'application afin d'estimer la richesse d'expressivité d'OCaLustre : on s'intéresse donc à des applications dans les domaines de la robotique (on pourra s'inspirer des challenges de robotique LEGO en Scade[3]), de la domotique, et à des applications industrielles comme dans le projet LCHIP cité en section 2.

Enfin, plusieurs projets destinés à porter OCaPIC sur d'autres microcontrôleurs (en particulier des microcontrôleurs Atmel pour cartes Arduino et des microcontrôleurs ARM pour cartes Nucléo) sont en cours, et nous encouragent donc à nous abstraire du modèle de microcontrôleur pour l'élaboration de nos futurs travaux.

3. https://sites.google.com/site/synccontest2017/home

Références

[1] C. Auger. *Compilation certifiée de SCADE/LUSTRE*. Thèse de doctorat, Université Paris Sud-Paris XI, 2013.

[2] G. Berry. Scade : Synchronous design and validation of embedded control software. In *Next Generation Design and Verification Methodologies for Distributed Embedded Control Systems*, pages 19–33. Springer, 2007.

[3] D. Biernacki, J.-L. Colaço, G. Hamon, and M. Pouzet. Clock-directed modular code generation for synchronous data-flow languages. *SIGPLAN Not.*, 43(7) :121–130, June 2008.

[4] N. Brouwers, P. Corke, and K. Langendoen. Darjeeling, a Java Compatible Virtual Machine for Wireless Sensor Networks. In *Proceedings of the ACM/IFIP/USENIX Middleware '08 Conference Companion*, 2008.

[5] P. Caspi, G. Hamon, and M. Pouzet. Synchronous functional programming : The lucid synchrone experiment. *Real-Time Systems : Description and Verification Techniques : Theory and Tools. Hermes*, 2008.

[6] P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice. Lustre : A declarative language for real-time programming. In *In 14th Symposium on Principles of Programming Languages (POPL'87). ACM*, POPL '87, pages 178–188, New York, NY, USA, 1987. ACM.

[7] P. Caspi and M. Pouzet. Synchronous kahn networks. In *ACM SIGPLAN Notices*, volume 31, pages 226–238. ACM, 1996.

[8] J.-L. Colaço and M. Pouzet. Clocks as first class abstract types. In *International Workshop on Embedded Software*, pages 134–155. Springer, 2003.

[9] L. Damas and R. Milner. Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 207–212. ACM, 1982.

[10] M. Feeley and D. Dubé. Picbit : a Scheme system for the PIC microcontroller. In *Scheme and Functional Programming Workshop (SFPW'03)*, pages 7–15, Nov. 2003.

[11] J.-C. Filliâtre and A. Paskevich. Why3 — where programs meet provers. In M. Felleisen and P. Gardner, editors, *Proceedings of the 22nd European Symposium on Programming (ESOP)*, volume 7792 of *Lecture Notes in Computer Science*, pages 125–128. Springer, Mar. 2013.

[12] X. Leroy. The ZINC experiment : an economical implementation of the ML language. Technical Report RT-0117, INRIA, Feb. 1990.

[13] MicroEJ. Embedded software solutions for iot devices - `http://www.microej.com`.

[14] V. St-Amour and M. Feeley. Picobit : A Compact Scheme System for Microcontrollers. In *International Symposium on Implementation and Application of Functional Languages (IFL'09)*, pages 1–11, Sept. 2009.

[15] B. Vaugon, P. Wang, and E. Chailloux. Programming Microcontrollers in Ocaml : the OCaPIC Project. In *International Symposium on Practical Aspects of Declarative Languages (PADL 2015)*, number 9131 in Lecture Notes in Computer Science, pages 132–148. Springer Verlag, June 2015.