HAL
open science

Pendulum : une extension réactive pour la programmation Web en OCaml

Rémy El Sibaïe, Emmanuel Chailloux

To cite this version:

Rémy El Sibaïe, Emmanuel Chailloux. Pendulum : une extension réactive pour la programmation Web en OCaml. Vingt-septièmes Journées Francophones des Langages Applicatifs (JFLA 2016), Jan 2016, Saint-Malo, France. hal-01333578

HAL Id: hal-01333578

https://inria.hal.science/hal-01333578v1

Submitted on 17 Jun 2016

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Pendulum : une extension réactive pour la programmation Web en OCaml

Rémy El Sibaïe[1] & Emmanuel Chailloux[1]

1 : Sorbonne Universités, UPMC Univ Paris 06, CNRS, LIP6 UMR 7606, 4 place Jussieu 75005 Paris.
remy.el-sibaie@lip6.fr
emmanuel.chailloux@lip6.fr

Résumé

La programmation des interactions des différents composants d'une application Web est devenue particulièrement complexe tant au niveau de la conception que de la mise au point. Peu de garanties peuvent être apportées sur la coordination des interactions internes et les réactions aux évènements externes reposant sur le modèle de concurrence de JavaScript. Il devient alors nécessaire de proposer un modèle de concurrence simple améliorant cette situation. Dans ce cadre, cet article présente une extension réactive pour la programmation Web en OCaml, appelée Pendulum, inspirée des environnements HipHop et Ocsigen.

1. Introduction

L'exécution des applications dans le client Web utilise un modèle d'évènements de haut niveau. Le navigateur donne au développeur accès à l'interface de programmation utilisable via JavaScript. Elle permet d'exprimer comment traiter ces évènements mais pas d'accéder à l'implantation de la boucle interne du navigateur qui les capture. Il existe plusieurs façons d'utiliser cette interface pour adapter le modèle, l'améliorer ou le rendre plus simple à manipuler. Un certain nombre de bibliothèques ont, par ailleurs, vu le jour pour rendre plus cohérente la conception du document HTML et la manipulation du DOM [1], une vue de l'interface graphique de la page sous forme d'arbre. La programmation simultanée du client et du serveur est aussi un paradigme étudié dans de nombreux *frameworks* comme par exemple Hop [14] et Ocsigen [1]. Ces systèmes *multi-niveaux* renforcent la cohérence des différents composants au sein d'une application Web entre client et serveur. Le typage des interactions, des requêtes client-serveur et des évènements apporte aussi davantage de fiabilité à l'application. Hop et Ocsigen proposent des couches spécifiques de concurrence au niveau du client Web comme par exemple LWT [15] pour Ocsigen qui est un modèle de thread coopératifs. Aujourd'hui, le besoin des programmeurs s'oriente de plus en plus vers l'orchestration et le temps réel et notamment sur les interactions et la dépendance entre les données de l'application et les communications. Par exemple, React [6] de Facebook est une bibliothèque qui propose un modèle de programmation en flot de données pour la construction d'interfaces graphiques.

Ces atouts proposés par la recherche et l'industrie du Web, ne sont pourtant pas suffisants pour résoudre les problèmes de causalité introduisant des comportements paradoxaux comme des dépendances entre actions. Il est difficile d'effectuer des vérifications statiques dans un modèle évènementiel tel que celui de JavaScript qui, de par sa flexibilité, est difficile à analyser.

1. Document Object Model

Dans le cadre de la vérification des programmes temporels, la *programmation réactive synchrone* a fait ses preuves depuis son introduction dans les années 80. Elle fut utilisée dans un premier temps dans la programmation des systèmes embarqués temps réel pour la spécification et l'écriture de programme dans un contexte limité en ressources. Elle permet d'exprimer la causalité des évènements d'un système en imposant de fortes contraintes et donc d'obtenir des garanties avant l'exécution. Le modèle est basé sur l'hypothèse synchrone, où les communications entre les tâches concurrentes s'effectuent en temps nul. L'exécution des tâches est basée sur une horloge et elles sont synchronisées à chaque pas d'horloge. Ces propriétés nous apportent à la fois le déterminisme et l'absence d'interblocage au moyen d'une analyse statique. De plus, le modèle réactif impératif[2] à la Esterel [2] propose des constructions expressives pouvant se calquer sur le modèle de programmation des programmes Web client.

Fort de cette constatation, cet article introduit l'extension `pendulum`[3] qui ajoute à la programmation Web évènementielle le concept de programmation *réactive synchrone* au sens d'Esterel, ainsi qu'une interface de programmation adaptée au client Web et au moteur modèle d'exécution sous-jacent, JavaScript. Le langage apporte de l'expressivité au programmeur qui se matérialise par la possibilité de composer les tâches concurrentes au niveau du langage. Ces tâches communiquent par des signaux diffusés à l'ensemble du programme. Cette expressivité facilite grandement l'orchestration d'applications Web complexes, tout en étant accompagnée d'une fiabilité accrue grâce à une analyse de causalité.

Nous utilisons OCaml [8], un langage à typage statique strict, pour l'implantation et l'utilisation de l'extension de syntaxe, pour proposer à la fois une sûreté de typage et une sûreté du comportement. Ainsi, le code `pendulum` s'écrit dans un fichier de code source OCaml. L'extension de syntaxe est embarquée grâce au moteur de macros PPX, qui permet de réécrire des parties de l'arbre de syntaxe abstraite d'OCaml étiquetées par un identifiant choisi. La méthode d'implantation de la compilation, inspirée de l'ouvrage de référence *Compiling Esterel* [13], génère du code séquentiel. `pendulum` propose un modèle de concurrence facile à mettre en œuvre tout en produisant du code efficace. De plus, l'extension de syntaxe est intégrable aux environnements complexes en OCaml sans difficultés de déploiement, lui permettant en particulier de profiter de `js_of_ocaml` afin d'être exécuté dans un navigateur Web.

Cette proposition s'inspire des travaux effectués sur ReactiveML [9] et sur HipHop [3]. Le premier est un langage ML étendu avec des constructions réactives. Le second est une extension de syntaxe pour la programmation réactive synchrone dans le contexte Web du langage Hop. Le modèle proposé ici diffère de ReactiveML par son intégration en tant qu'extension dans un code OCaml standard, ainsi que par son ordonnancement statique. Il diffère de HipHop aussi par l'ordonnancement mais aussi par le typage puisque celui de Hop est dynamique, ce qui a un impact sur les garanties à la compilation.

La section 2 présente l'extension réactive Pendulum et illustre par un exemple simple son utilisation en OCaml. La section 3 s'intéresse à son utilisation pour la programmation du Web, en particulier au niveau des interactions avec JavaScript. La section 4 décrit les différentes étapes de sa compilation, dont la construction du graphe de flot de contrôle du programme, et les vérifications de propriétés de causalité effectuées permettant l'ordonnancement statique des tâches. Pour valider la démarche une application Web plus conséquente est présentée en section 5. La section 6 compare cette approche réactive à des travaux connexes en programmation synchrone et en programmation Web. La conclusion revient sur les motivations de ce travail et sa poursuite.

2. Présentation de Pendulum

La programmation réactive propose un modèle où l'environnement du programme est responsable de la réaction aux interactions. Le programme interagit en continu avec cet environnement. Quand

2. par opposition au modèle flot de données comme Lustre
3. Le prototype peut être téléchargé et installé depuis la page suivante : http://github.com/remyzorg/pendulum.

l'environnement émet une entrée, le programme répond avec la sortie dans un même instant logique. La vision est donc inversée par rapport aux modèles plus standards comme la programmation évènementielle. En programmation évènementielle, on décrit le programme du point de vue du serveur, qui traite les demandes de l'environnement quand il est disponible.

La difficulté de proposer un modèle réactif-synchrone dans un contexte évènementiel asynchrone est d'interfacer élégamment les deux mondes. En particulier, l'interface de programmation du modèle synchrone ne doit pas faire de suppositions sur la façon dont sont gérées les entrées et les sorties du modèle asynchrone. Ces choix doivent être laissés à la liberté de l'utilisateur. C'est pourquoi la programmation en `pendulum` se fait sur deux plans. Premièrement, on décrit le programme réactif avec le langage de l'extension, et ensuite on utilise l'interface de programmation pour y faire appel, le tout dans le même programme OCaml.

2.1. Le langage `pendulum`

Le langage `pendulum` est un sous-ensemble du langage réactif-synchrone Esterel. La syntaxe complète de `pendulum` apparaît dans la figure 1. Cet ensemble est un noyau permettant de définir d'autres instructions de plus haut niveau. Un programme synchrone réactif peut être vu comme un automate à état dont l'exécution est découpée en instants où chaque instruction agit sur l'état de l'automate. A la fin de l'instant, l'automate peut être soit en *pause*, soit *terminé* et une instruction est donc dite *instantanée* si elle termine au premier instant de son exécution.

Le modèle est impératif car les opérations sont des instructions qui communiquent par effet de bord sur un environnement partagé par toute les tâches parallèles. Ces communications se font au moyen de signaux qui sont soit absents soit présents et qui transportent des valeurs. Pour diffuser un signal on utilise l'instruction instantanée `emit`.

Le modèle de concurrence s'exprime ici au travers de l'opérateur de composition parallèle `||`. Dans ce modèle de concurrence, deux tâches en parallèle exécutent leur instant en même temps. L'instruction parallèle termine quand les deux tâches terminent. Pour exécuter deux tâches séquentiellement on utilise l'opérateur `;`.

instr := emit *ident* ocaml-expr
| nothing | pause
| present *ident* *instr* *instr*
| loop *instr* | exit *label*
| *instr* || *instr* | *instr* ; *instr*
| let *ident* = *ocaml-expr* in *instr*
| trap *ident* *instr*
| await *ident*
| atom *ocaml-expr*
| run *ident* (*ident*[, *ident*])
| suspend *signal* *instr*

FIGURE 1 – Grammaire de `pendulum`

La présence d'un signal peut être testée via `present s e1 e2` qui saute intantanément sur une instruction selon le résultat. Le `let-in` déclare un signal local. Les autres constructions sont assez standard : `pause` attend l'instant suivant, `nothing` ne fait rien, `loop e` boucle infiniment, `trap label e` permet de s'échapper de l'instruction `e` si elle exécute `exit label`. L'instruction `await s` attend l'instant suivant tant que `s` est absent. Pour finir, `suspend s e` bloque une instruction pendant l'instant courant si `s` est présent et `run m` exécute le programme réactif `m`. L'avantage de l'extension de syntaxe est qu'elle donne accès à toute les fonctionnalités du langage hôte. C'est un avantage utilisé ici

au niveau du non-littéral *ocaml-expr* qui peut être remplacé par n'importe quelle expression OCaml. On en déduit aussi que les signaux peuvent être de n'importe quel type OCaml. On note en particulier la présence de l'instruction `atom` qui permet d'exécuter une expression envoyant une valeur de type *unit*.

2.2. Interface de programmation

Un programme réactif-synchrone dépend d'une horloge, qui émet une oscillation régulière rythmant l'exécution. Ici, le moteur de cette horloge correspond à la partie asynchrone du programme global, où on doit faire appel à notre programme réactif. On peut dire que l'application est globalement asynchrone mais synchrone localement à un programme réactif. Le but de l'interface de programmation est de laisser à l'utilisateur la gestion de l'horloge avec une certaine liberté pour qu'il puisse l'adapter à différents cas de programmation. On peut associer une description de programme synchrone à une variable de l'environnement de la façon suivante par exemple `let%sync m = ...`. La variable `m` est conservée après le passage du pré-processeur, qui transforme le code `pendulum` à droite du symbole `=` en code OCaml. Vue depuis le code OCaml, la valeur `m` est une fonction dont le corps est le résultat de la compilation. On appelle `m` la fonction d'instanciation ou d'initialisation. Son type dépend des signaux déclarés en entrée (`input`) et en sortie (`output`), ce qui lui donne la forme suivante :

$$\overbrace{t_0 \ * \cdots * \ t_{n-1}}^{entrees} \rightarrow \overbrace{(\tau_0 \rightarrow unit) \ * \cdots * \ (\tau_{k-1} \rightarrow unit)}^{sorties}$$
$$\rightarrow \overbrace{((t_0 \rightarrow unit) \ * \cdots * \ (t_{n-1} \rightarrow unit)) \ * \ (unit \rightarrow status)}^{accesseurs \ * \ reaction} \quad (1)$$

où *entrees* est un produit des types des entrées, *sorties* est un produit dont chaque $\tau_i \rightarrow unit$ est une fonction de rappel pour les signaux de sortie du programme. La valeur de retour est un produit de deux types. Le premier est le produit des types des accesseurs aux entrées du programme dont la forme de chacun est $t_i \rightarrow unit$. Ces accesseurs vont modifier l'environnement de l'instance du programme réactif et en particulier l'état et la valeur des signaux. On notera que la valeur est conservée entre les instants mais que l'état est réinitialisé à absent à chaque fin d'instant. Pour finir, La deuxième composante de la valeur de retour est la fonction de réaction dont le type est $unit \rightarrow status$.

On peut prendre l'exemple du programme réactif suivant qui affiche `Hello world !` si `s` est présent sinon `Hello !` à chaque instant. Les mots clefs `begin` et `end` parenthèsent un groupe d'instructions.

```
let%sync hello =
  input s;
  loop begin (* boucle infinie *)
    atom (print_string "Hello ");
    present s (* si s est present *)
      (atom (print_string "world !\n"))
    (* sinon *)
      (atom (print_string "!\n"));
    pause
  end
```

A partir de la fonction `hello`, on créé les valeurs `set_s` l'accesseur de `s` et `hello_inst`, la fonction de réaction de `hello`. Le premier appel à `hello_inst` affiche simplement `Hello !` car `s` est absent. L'appel à `set_s` permet de passer son état à présent comme on l'observe ci-dessous.

```
(* instanciation *)
let set_s, hello_inst = hello () in
hello_inst (); (* affiche "Hello !" *)
set_s ();
hello_inst (); (* affiche "Hello world !" *)
```

Pour résumer cette section, l'interface de programmation se compose d'une fonction en OCaml qui initialise l'environnement et retourne les fonctions d'accès à cet environnement pour modifier l'état et la valeur des signaux, l'instant, etc. Cette interface est relativement souple pour lui permettre de s'adapter à différents cas d'utilisation. On souhaiterait maintenant adapter notre programmation synchrone à un contexte Web.

3. Programmer pour le Web

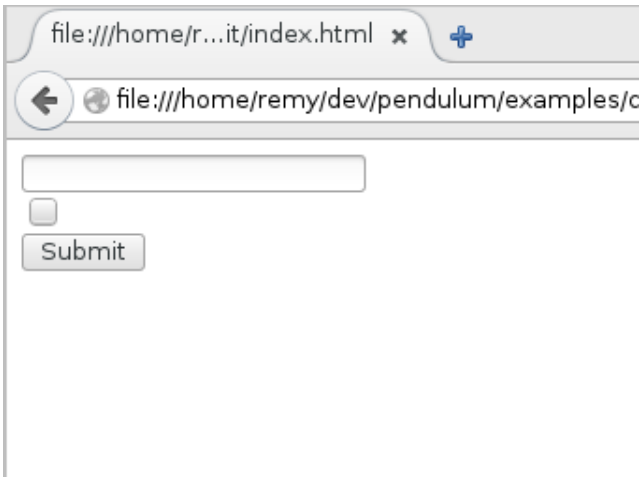
Dans cette section on commence par poser le contexte de programmation en décrivant le fonctionnement des interactions dans le client Web puis nous présentons la méthode de programmation au travers d'un exemple.

La gestion des interactions en JavaScript se présente sous forme d'une programmation évènementielle où les évènements sont lancés à partir d'un composant du DOM. Quand un évènement apparaît, il est ajouté à une file d'évènements. Le navigateur exécute une boucle interne qui n'est pas accessible au programme client. Elle vide la file et exécute les *fonctions de rappel* affiliées à chacun des évènements. Ces *fonctions de rappel* sont ajoutées au DOM par le programmeur. Si la boucle est occupée à traiter un évènement et qu'un autre survient, il est simplement ajouté à la file et sera traité au tour suivant. L'interface de programmation demande donc au programmeur d'éditer les propriétés correspondants aux fonctions de rappel des évènements voulus (*ie. onclick, oninput, etc*).

Nous allons maintenant voir comment appliquer notre modèle de programmation réactive à l'environnement d'exécution JavaScript que nous venons de décrire. On donne pour cela un exemple d'utilisation. Le programme de la figure 2a correspond à une application avec une interface graphique qui contient un champ de texte, une case à cocher et un bouton. Le programme répète l'opération suivante à l'infini : il attend que la case soit cochée (`await checked`), il attend que le bouton soit cliqué (`await request`), il affiche le contenu du champ de texte sur la sortie standard et émet un signal de sortie `reset`. Si la case est décochée (`present unchecked`) avant que le bouton ne soit cliqué, il termine l'itération courante et recommence. L'opérateur `!!` permet de récupérer la valeur courante d'un signal depuis une expression OCaml.

```
let%sync form =
  input checked, unchecked,
      request;
  output reset;
  loop begin
    trap t begin (
        await checked;
        await request;
        atom (print_endline
            !!request);
        emit reset ();
        exit t;
      ) || loop (present unchecked
          (exit t); pause)
    end;
    pause
  end
```

(a) Le programme réactif



(b) Le formulaire

FIGURE 2 – Exemple Web avec un échappement

On doit maintenant instancier notre programme pour récupérer les accesseurs et la fonction de réaction dans le but de l'exécuter comme on l'a montré dans la section précédente. On aimerait exprimer que le programme ne s'exécute que si une action de l'utilisateur survient et qu'un signal n'est présent que si l'action correspondante a été effectuée. On a besoin pour cela d'utiliser l'interface du DOM. Ce dernier est complètement accessible grâce à la bibliothèque `js_of_ocaml` [12]. En particulier, les composants que l'on va manipuler sont l'`input`, pour la case à cocher et le texte, et le `button`. L'interface OCaml, réduite à nos besoins, est la suivante :

```
class type inputElement = object ('self)
  method checked : bool t prop (* vrai si la case est cochee *)
  method onclick : ('self t, mouseEvent t) event_listener writeonly_prop
  method value : js_string t prop (* contenu du champ de texte *)
end

class type buttonElement = object ('self)
  method onclick : ('self t, mouseEvent t) event_listener writeonly_prop
end
```

Cette interface est une version fortement typée adaptée du standard EcmaScript. On comprend que `checked` et `onclick` sont des champs mutables de l'objet `input`. On peut récupérer la valeur de `checked` mais pas de `onclick` qui est en écriture seule. La propriété `onclick` en particulier dissimule une fonction qui est exécutée quand l'utilisateur clique sur l'élément. Si on veut modifier la réaction à cette interaction, il faut modifier la valeur de la propriété. On peut utiliser les opérateurs accessibles depuis `js_of_ocaml`, `##.` pour accéder à une propriété d'un élément ainsi que l'affectation `:=` pour la modifier. On commence par créer la fonction `reset` qui réinitialise l'état de notre interface graphique. On considère que `mycheckbox` et `myinput` sont des composants initialisés dans le document HTML et qu'ils ont déjà été associés à ces variables. On appelle notre fonction d'initialisation `form` généré par le programme réactif pour obtenir les accesseurs et `react`, notre fonction de réaction.

```
let reset () =
  mycheckbox##.checked:= false;
  mytext##.value:= ""
in
(* Instanciation *)
let (set_checked, set_unchecked, set_request, get_reset), react =
  form ((), (), "", reset)
in
```

On modifie ensuite l'attribut `onclick` de notre élément `mycheckbox` en lui attribuant une fonction qui met à jour le signal du programme réactif correspondant à l'état de la case. Si la case est cliquée est que `checked` est vrai, on fait appel à `set_checked`, sinon à `set_unchecked`. Ensuite, quand `mybutton` est cliqué on envoit la valeur du champ de texte au programme réactif via l'accesseur `set_request`. On exécute également la fonction `react` dans nos deux fonctions.

```
mycheckbox##.onclick:= handler (fun _ ->
  if Js.to_bool mycheckbox##checked then set_checked ()
  else set_unchecked ();
  react (); Js._true
);
mybutton##.onclick := handler (fun ev ->
  set_request (Js._string mytext##.value);
  react (); Js._true);
```

Nous avons vu comment déclencher l'exécution d'un instant d'un programme réactif depuis le modèle de programmation évènementielle de JavaScript. Il est possible qu'un programme ait besoin

de s'exécuter sans action de la part de l'utilisateur, pour effectuer une suite d'animations par exemple. L'interface du DOM nous donne accès aux méthodes nécessaires.

L'interface graphique du navigateur se rafraîchit au mieux à une vitesse de 60 images par secondes. Il est possible d'intervenir dans cette étape de rafraîchissement pour y effectuer les modifications de la page. Cette pratique a pour but d'éviter les rendus intermédiaires et donc de faciliter le travail du navigateur. Pour ce faire, le navigateur associe au DOM une liste de requêtes de rafraîchissement initialisée à la liste vide. Pour ajouter une fonction à cette liste, on appelle `requestAnimationFrame()`[16]. Cette fonction est une méthode de `window`, le contexte globale de l'application client. Cet appel a pour effet de repousser l'exécution de la fermeture passée en paramètre jusqu'à la prochaine étape de mise à jour de la page. A ce moment là, le navigateur exécute les fonctions qui sont dans la liste jusqu'à ce que celle-ci soit vide. Il est donc avantageux d'effectuer les écritures sur le DOM dans une fermeture que l'on passe en paramètre à `requestAnimationFrame` pour que le navigateur puisse les appliquer en une seule fois. On peut alors simuler une horloge dont la fréquence est celle du navigateur, en rappelant `requestAnimationFrame` à la fin de la fermeture, comme dans l'exemple suivant avec notre fonction `redraw`.

```
let rec redraw _ =
  let _ = window##requestAnimationFrame (Js.wrap_callback redraw) in
(* Ici on appelle le programme reactif qui produit une animation *)
in
window##requestAnimationFrame (Js.wrap_callback redraw)
```

En résumé, l'interface du DOM nous fournit, au travers d'une solution efficace pour la mise à jour de la page, ce qui se rapproche le plus d'une horloge globale pour l'application. On observe que l'interface de `pendulum` est assez souple pour être utilisée dans le cadre d'un application Web client et surtout qu'utiliser l'interface du DOM et les "les bonnes pratiques" de JavaScript coïncident avec l'interface de `pendulum`.

4. Compilation

Après avoir montré au travers d'exemples la façon de programmer en `pendulum`, on présente les différentes passes de compilation et plus particulièrement l'étape intermédiaire entre le code synchrone et le code OCaml.

4.1. Représentation intermédiaire axé sur le flot de contrôle

La compilation de `pendulum` utilise une structure de flot de contrôle intermédiaire $\mathcal{C}$ qui représente l'exécution de l'instant d'une machine synchrone. La compilation d'un programme $\mathcal{P}$ construit un triplet $\mathcal{C} := (\mathcal{S}, \mathcal{F}, \mathcal{G})$ dont les trois composantes sont : l'arbre de sélection $\mathcal{S}$, $\mathcal{F}$ l'environnement des signaux et $\mathcal{G}$, le graphe de flot de contrôle. Ces primitives sont listées dans la figure 3.

L'arbre de sélection $\mathcal{S}$ indique quelles instructions sont activées pour le prochain instant. C'est une copie de l'arbre de syntaxe de $\mathcal{P}$ où chaque nœud est identifié de façon unique et a une information booléenne de sélection. Il existe deux actions permettant de mettre à jour $\mathcal{S}$: *enter* n qui active un nœud et *exit* n qui désactive un nœud et tous ses fils récursivement. Pour finir *sel* est un prédicat qui vaut vrai si l'instruction s est sélectionnée dans l'arbre de sélection.

L'environnement des signaux $\mathcal{F}$ est un environnement qui contient tous les signaux du programme et leur information de présence. Le prédicat *present* s vaut *vrai* si le signal est présent à l'instant courant dans $\mathcal{F}$. La primitive *emit* s met le signal dans l'état *présent* dans dans $\mathcal{F}$.

Le graphe de flot de contrôle $\mathcal{G}$ est un graphe acyclique orienté qui représente la fonction d'*instant* de $\mathcal{P}$.La figure 3 donne en détails la grammaire de $\mathcal{G}$. Les nœuds de ce graphe sont des instructions correspondant soit à des actions, soit à un branchement, soit à un prédicat.

$$
\begin{array}{l}
\mathit{action} \ ::= \ \text{emit } \mathit{signal} \\
\quad | \ \ \text{exit } \mathit{int} \\
\quad | \ \ \text{enter } \mathit{int} \\
\quad | \ \ \text{local } \mathit{signal} \\
\quad | \ \ \text{inst } (\mathit{id}, \ \mathit{signal}[,\mathit{signal}]) \\
\\
\mathit{test} \ ::= \ \text{sel } \mathit{int} \mid \text{present } \mathit{signal} \\
\quad | \ \ \text{sync } (\mathit{int}, \ \mathit{int}) \mid \text{finished} \\
\quad | \ \ \text{ispaused } \mathit{id} \\
\\
\mathit{instr} \ ::= \ \text{call } (\mathit{action}, \ \mathit{instr}) \\
\quad | \ \ \text{branch } (\mathit{test}, \ \mathit{instr}, \ \mathit{instr}) \\
\quad | \ \ \text{fork } (\mathit{instr}, \ \mathit{instr}) \\
\quad | \ \ \text{finish} \mid \text{pause}
\end{array}
$$

FIGURE 3 – Langage des nœuds du graphe de flots de contrôle

— $finish$ et *pause* sont les instructions de degré sortant nul. La première instruction signifie que le programme est définitivement terminé et la seconde qu'il s'arrête sur une pause.
— L'instruction *call* permet d'exécuter une *action* et de continuer l'exécution en séquence sur le nœud suivant. On découvre aussi l'action *local* qui conserve l'information de déclaration des signaux locaux nécessaire pour la compilation vers OCaml. L'action *inst* fait un travail similaire en instanciant le programme réactif externe en paramètre, dans le but de ce souvenir de l'instance du programme réactif appelé.
— L'instruction $fork$ représente deux tâches parallèles et le prédicat $sync(i_1, i_2)$ correspond à la synchronisation des deux tâches, en testant les cas : si une tâche au moins est en pause, alors on choisit la branche pause, sinon la branche de fin.
— Le prédicat de branchement *ispaused* sert pour la compilation de l'instruction `run`. Il renvoie vrai si l'exécution d'un programme réactif externe appelé se termine sur une pause.

4.2. Construction du graphe de flot de contrôle

Une instruction de programme réactif possède un état interne dans l'arbre de sélection nécessaire pour maintenir l'information du déroulement de l'exécution entre les instants. L'exécution n'est pas la même au premier instant et au n^{ieme} pour une même instruction. On veut introduire ce comportement dans le graphe de flot de contrôle pour chaque instruction. En particulier, si le programme se met en pause, on veut reprendre l'exécution là où elle avait été laissée à l'instant précédent. C'est pourquoi on utilise deux fonctions de construction ayant chacune une mission différente.

La fonction de *surface* construit le flot de contrôle du premier instant d'exécution et la fonction de *profondeur* construit le flot de contrôle des instants suivants. Pour calculer le flot de contrôle global du programme, on agrège les fonctions de surface et de profondeur de chacune des instructions du programme par un procédé de compilation que l'on verra dans la suite de la section.

Ces deux fonctions sont mutuellement récursives et représentées symboliquement par $\mathcal{S}$ et $\mathcal{D}$. Elles sont paramétrées par un environnement qui est transformé au fur et à mesure des étapes. La transformation de l'environnement est donnée sur l'opérateur $\rightsquigarrow$ qui représente l'étape de transformation. A droite de cet opérateur, les fonctions $\mathcal{S}$ et $\mathcal{D}$ capturent implicitement

l'environnement modifié sauf si les paramètres sont explicites. L'environnement est un triplet (ω, κ, Ω). Le symbole ω représente le sous-graphe qui suit l'instruction courante quand elle s'arrête, κ, quand elle se met en pause et Ω un dictionnaire qui associe les labels d'échappement accessibles depuis l'instruction courante aux graphes correspondants à l'échappement. La fonction de transformation $\mathcal{T}$ agrège les résultats de $\mathcal{S}$ et $\mathcal{D}$ et construit le flot de contrôle suivant où p est l'instruction courante.

$$\mathcal{T}(p) \quad \overset{\texttt{finish},\texttt{pause},\{\}}{\rightsquigarrow} \quad \texttt{branch}(finished, \omega, \texttt{branch}(sel\ p, \mathcal{D}(p), \mathcal{S}(p))))$$

Pour construire ce graphe d'initialisation, on fixe les paramètres $\kappa := pause$ et $\omega := finish$. Effectivement la continuation de fin du programme complet est le nœud $finish$, et la continuation de pause du programme complet est le nœud $pause$. Ce graphe représente l'exécution suivante : si le programme est déjà fini ($finished$ est vrai), alors on termine sur ω (qui vaut $finish$), sinon on teste la chose suivante : si la première instruction du programme est sélectionnée, cela signifie que l'on reprend le programme, et donc on choisit la branche qui appelle $\mathcal{D}$, sinon on est à la première exécution et on choisit la branche qui appelle $\mathcal{S}$.

Les figures 4 et 5 montrent respectivement l'implantation des fonctions $\mathcal{S}$ et $\mathcal{D}$ pour l'instruction p. Les lettres minuscules sont des variables pour les identifiants des instructions. L'opérateur $\blacktriangleright$ (associatif à droite) est l'abréviation de *call* de sorte que $enter\ 1 \blacktriangleright pause$ se comprend comme $call(enter\ 1, pause)$. On rappelle pour finir que les instructions que l'on génère sont celles du langage du graphe de flot de contrôle de la figure 3.

$$\begin{array}{rcl}
\mathcal{S}(\texttt{nothing}) & \overset{\omega,\kappa,\Omega}{\rightsquigarrow} & \omega \\
\mathcal{S}(\texttt{pause}) & \overset{\omega,\kappa,\Omega}{\rightsquigarrow} & enter\ p \blacktriangleright \kappa \\
\mathcal{S}(\texttt{emit}\ s) & \overset{\omega,\kappa,\Omega}{\rightsquigarrow} & emit\ s \blacktriangleright \omega \\
\mathcal{S}(\texttt{atom}\ f) & \overset{\omega,\kappa,\Omega}{\rightsquigarrow} & atom\ f \blacktriangleright \omega \\
\mathcal{S}(q\ ;\ r) & \overset{\mathcal{S}(r)^{\omega,\kappa,\Omega},\kappa,\Omega}{\rightsquigarrow} & enter\ p \blacktriangleright \mathcal{S}(q) \\
\mathcal{S}(q \parallel r) & \overset{\texttt{branch}(\texttt{sync}(q,r),\omega,\kappa),\Omega}{\rightsquigarrow} & enter\ p \blacktriangleright \texttt{fork}\ (\mathcal{S}(q), \mathcal{S}(r)) \\
\mathcal{S}(\texttt{loop}\ q) & \overset{\kappa,\kappa,\Omega}{\rightsquigarrow} & enter\ p \blacktriangleright \mathcal{S}(q) \\
\mathcal{S}(\texttt{signal}\ s\ q) & \overset{exit\ p \blacktriangleright \omega,\kappa,\Omega}{\rightsquigarrow} & enter\ p \blacktriangleright local\ s \blacktriangleright \mathcal{S}(q) \\
\mathcal{S}(\texttt{present}\ s\ q\ r) & \overset{exit\ p \blacktriangleright \omega,\kappa,\Omega}{\rightsquigarrow} & enter\ p \blacktriangleright \texttt{branch}\ (signal\ s, \mathcal{S}(q), \mathcal{S}(r)) \\
\mathcal{S}(\texttt{suspend}\ s\ q) & \overset{exit\ p \blacktriangleright \omega,\kappa,\Omega}{\rightsquigarrow} & enter\ p \blacktriangleright \mathcal{S}(q) \\
\mathcal{S}(\texttt{run}\ id\ signals) & \overset{exit\ p \blacktriangleright \omega,\kappa,\Omega}{\rightsquigarrow} & enter\ p \blacktriangleright inst(id, signals) \blacktriangleright \texttt{branch}\ (ispaused\ id, \kappa, \omega) \\
\mathcal{S}(\texttt{trap}\ l\ q) & \overset{exit\ p \blacktriangleright \omega,\kappa,\{(l,\omega)\}\cup\Omega}{\rightsquigarrow} & enter\ p \blacktriangleright \mathcal{S}(q) \\
\mathcal{S}(\texttt{exit}\ l) & \overset{\omega,\kappa,\Omega}{\rightsquigarrow} & \Omega(l)
\end{array}$$

FIGURE 4 – Fonction de surface de p

Il y a plusieurs éléments notables dans ces fonctions :

— Les expressions simples `nothing`, `atom` et `emit` ne sont qu'un branchement d'une action vers la continuation ω. Par contre `pause` branche vers κ.

— Dans le cas de la séquence `;`, la continuation de fin de q est remplacée par le graphe de surface de r pour exprimer que l'on exécute q puis r et $\mathcal{S}(q)$ est paramétrée par ω. Dans la fonction de profondeur, figure 5 la transformation diverge sur deux cas. Si q est instantanée, on a simplement la reprise r, signifiant que cette dernière instruction avait été débutée dans l'instant précédent. Dans le cas contraire, on teste dynamiquement l'état de sélection de q, si il est vrai on reprend

q et on met le graphe de surface de r comme continuation, sinon on exécute le graphe de surface de r au même instant.

— Le parallèle $\parallel$ remplace à la fois ω et κ par un nœud de synchronisation entre q et r dans l'environnement dans les deux fonctions.

— La boucle remplace la continuation de fin par la surface de q, pour représenter l'idée de répétition infinie. De plus, ω disparait et se fait remplacer par κ. Le seul moyen de quitter une boucle est d'utiliser `exit` et `trap`.

— La compilation de `trap` ajoute un libellé dans Ω qui pointe sur ω. Le libellé est utilisé dans la compilation de `exit` pour brancher sur cette continuation.

— La compilation du `run` dans $\mathcal{S}$ génère un nœud *inst* pour conserver l'information dans la génération du code OCaml qu'une nouvelle instance de ce programme est nécessaire en ce point du graphe de flot de contrôle. On enchaine par un nœud *branch* qui teste si l'exécution de cette instance termine sur pause et sélectionne la branche κ ou ω en conséquence.

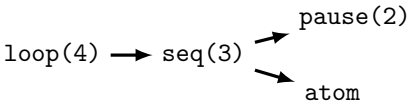
$$
\begin{array}{rcl}
\mathcal{D}(\texttt{nothing}) & \overset{\omega,\kappa,\Omega}{\rightsquigarrow} & \omega \\
\mathcal{D}(\texttt{pause}) & \overset{\omega,\kappa,\Omega}{\rightsquigarrow} & \mathit{exit}\ p \blacktriangleright \omega \\
\mathcal{D}(\texttt{emit}\ s) & \overset{\omega,\kappa,\Omega}{\rightsquigarrow} & \omega \\
\mathcal{D}(\texttt{atom}\ f) & \overset{\omega,\kappa,\Omega}{\rightsquigarrow} & \omega \\
\mathcal{D}(q\ ;\ r) & \overset{\mathit{exit}\ p\blacktriangleright\omega,\kappa,\Omega}{\rightsquigarrow} & \begin{array}{ll} \mathit{si}\ q\ \mathit{est\ instantané} & \mathcal{D}(r), \\ \mathit{sinon} & \texttt{branch}(\mathit{sel}\ q, \mathcal{D}(q)^{\mathcal{S}(r),\kappa,\Omega}, \mathcal{D}(r)) \end{array} \\
\mathcal{D}(q \parallel r) & \overset{\texttt{branch}(\texttt{sync}(q,r),\omega,\kappa),\Omega}{\rightsquigarrow} & \texttt{fork}\ (\texttt{branch}(\mathit{sel}\ q, \mathcal{D}(q), \kappa), \texttt{branch}(\mathit{sel}\ r, \mathcal{D}(r), \kappa) \\
\mathcal{D}(\texttt{loop}\ q) & \overset{\mathcal{S}(q),\kappa,\Omega}{\rightsquigarrow} & \mathcal{D}(q) \\
\mathcal{D}(\texttt{signal}\ s\ q) & \overset{\mathit{exit}\ p\blacktriangleright\omega,\kappa,\Omega}{\rightsquigarrow} & \mathcal{D}(q) \\
\mathcal{D}(\texttt{present}\ s\ q\ r) & \overset{\mathit{exit}\ p\blacktriangleright\omega,\kappa,\Omega}{\rightsquigarrow} & \texttt{branch}\ (\mathit{sel}\ q, \mathcal{D}(q), \mathcal{D}(r)) \\
\mathcal{D}(\texttt{suspend}\ s\ q) & \overset{\mathit{exit}\ p\blacktriangleright\omega,\kappa,\Omega}{\rightsquigarrow} & \texttt{branch}\ (\mathit{signal}\ s, \kappa, \mathcal{D}(q)) \\
\mathcal{D}(\texttt{run}\ \mathit{id}\ \mathit{signals}) & \overset{\mathit{exit}\ p\blacktriangleright\omega,\kappa,\Omega}{\rightsquigarrow} & \texttt{branch}\ (\mathit{ispaused}\ \mathit{id}, \kappa, \omega) \\
\mathcal{D}(\texttt{trap}\ l\ q) & \overset{\mathit{exit}\ p\blacktriangleright\omega,\kappa,\Omega}{\rightsquigarrow} & \mathcal{D}(q) \\
\mathcal{D}(\texttt{exit}\ l) & \overset{\omega,\kappa,\Omega}{\rightsquigarrow} & \omega
\end{array}
$$

FIGURE 5 – Fonction de profondeur de p

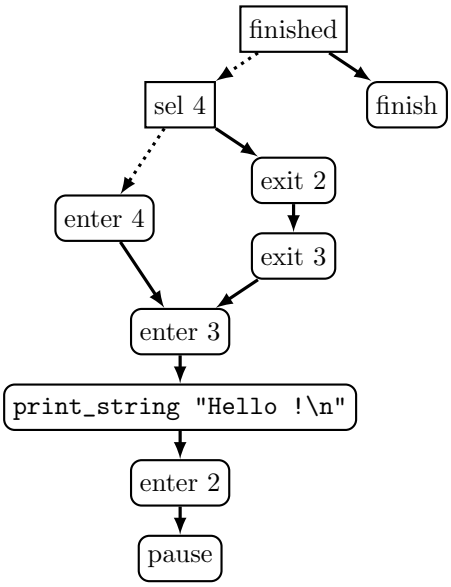
On applique la transformation vers un graphe de flots de contrôle pour le programme exemple `p` de la figure 6a. On commence par le représenter sous-forme d'arbre de sélection (figure 6b) en ajoutant un identifiant numérique aux instructions non-instantanés. On construit ensuite le flot d'initialisation décrit au début de la section. On applique ensuite les fonctions de surface et de profondeur sur le programme que l'on connecte respectivement aux branches *vrai* et *faux* de `sel 4`. On obtient alors la figure 6c qui représente le graphe de flot de contrôle de `p`. On peut exécuter `p` en interprétant dynamiquement ce graphe avec comme environnement l'arbre de sélection, pour se souvenir de l'instant précédent.

```
let%sync p =
loop begin
    atom (print_string
        "Hello!\n");
    pause
end
```

(a) Une boucle avec un `atom`



(b) Arbre de sélection de p



(c) Flot de contrôle de p

FIGURE 6 – Example de transformation en flot de contrôle

4.3. L'ordonnancement statique des tâches

La transformation en flot de contrôle utilise une forme de partage permettant de calculer en une seule fois le flot de contrôle de chaque instruction. On obtient donc un graphe acyclique orienté. Les branches d'un $fork$ se rejoignent au niveau des nœuds *sync* et les nœuds *branch* se rejoignent à la fin de l'exécution de l'instruction correspondante. Dans l'état où est le graphe après l'étape précédente, il n'est pas possible de le transposer vers du code où l'ordonnancement des tâches est statique puisqu'il contient des tâches parallèles. On supprimera donc la structure parallèle dans le but de ne pas calculer l'ordonnancement à l'exécution. L'algorithme d'ordonnancement consiste à réunir deux branches partant d'un nœud $fork$ du graphe en une seule branche. Il faut préalablement créer un ordre topologique des sommets en fonction des émissions et des tests de présence des signaux. Deux nœuds α et β sont en relation si α émet un signal sur lequel β effectue un test. On parcourt le graphe en appliquant la transformation suivante à partir des premiers fils du fork α_0 et β_0 récursivement (selon la terminologie de la figure 3)

- Si α et β sont deux actions (*call*), il n'y pas de relation de dépendance, on les séquence donc dans un ordre arbitraire.
- Si α est un test et β une action, on place β avant α (et inversement).
- Si les deux instructions sont des tests et qu'il existe une relation de dépendance, on les met en séquence dans l'ordre de la relation
- S'il y a un cycle de dépendances entre α et β, le programme est erroné.
- Si α est un nœud $fork$, on applique récursivement la transformation sur α et on l'entrelace ensuite avec β (et inversement).

Si le nœud positionné en premier dans l'ordonnancement est un *branch*, on doit chercher le point où ses deux branches se rejoignent pour appliquer l'entrelacement. On n'applique donc pas l'entrelacement à l'intérieur des branches. Si deux *branch* ont une relation de dépendance cyclique, l'analyse échoue et refuse le programme. L'analyse de causalité est, pour les besoins du prototype plus naïve que celle d'Esterel, mais aucun programme n'est accepté qu'Esterel refuserait.

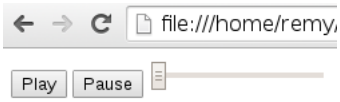
4.4. Du flot de contrôle vers OCaml

Une fois le graphe de flot de contrôle sans parallèles obtenu, on le compile vers OCaml, avec une passe intermédiaire sous forme de séquences de primitives du graphe. L'environnement des signaux à l'exécution est simplement un ensemble de références vers les valeurs des signaux. L'arbre de sélection a la forme d'un tableau de booléens. Le code séquentiel nous donne la fonction de réaction et on obtient l'interface de programmation présentée section 2.2.

5. Exemple d'application Web

On propose de montrer une application où l'utilisation des aspects synchrones aura une conséquence plus importante. On souhaite écrire un lecteur de média dans une page Web où les contrôles sont personnalisés et différents de ceux proposés par défaut pour la navigation dans les vidéos et fichiers son. On ajoute dans la page une balise `audio` contenant une balise `source` pointant vers un fichier musical. On ajoute aussi deux boutons pour les actions lecture (`play`) et pause puis une barre d'avancement et de recherche dans le fichier qui sera mise à jour et qui mettra à jour la audio (`progress`).

```
<audio id="media">
  <source src="aerosmith.mp3" type="audio/mp3">
</audio>
<button id="play">Play</button>
<button id="pause">Pause</button>
<input type="range" id="progress" value="0"/>
```



On veut ensuite créer le programme réactif qui va orchestrer ce mini lecteur audio. On commence par ajouter les signaux d'entrée nécessaires. Les clics sur les boutons vont correspondre aux signaux `play_clic` et `pause_clic`. On peut tout de suite écrire les premières lignes du programme réactif testant à chaque instant si un clic sur les boutons a lieu et lisant ou mettant en pause le son si c'est le cas.

```
let%sync reactive_player =
  input play_clic, pause_clic;
  input media_time, progress;
  input media, seek_on, seek_off;

  loop begin
    present play_clic (atom ((!!media)##play));
    present pause_clic (atom ((!!media)##pause));
    pause
  end
```

Par la suite, on veut gérer la barre de progression en fonction de l'avancée du morceau. Pour cela, on a besoin de l'objet `progress` comme signal d'entrée ainsi qu'un signal qui sera activé très régulièrement en indiquant le temps courant dans le media en flottant. On les nommera ici `progress` et `media_time`. On rappelle que l'opérateur `!!` permet de récupérer la valeur OCaml contenue dans le signal.

```
  ||
  loop begin
    present media_time
      (atom(update_slider !!progress !!media_time));
    pause
  end
```

On ajoute le code ci-dessus en parallèle du précédent. Il teste si le signal d'avancement du media est présent et si oui, il exécute une fonction OCaml externe qui va mettre à jour la barre de progression en fonction de la valeur de `media_time`. Si on compile et que l'on teste ce code on s'aperçoit qu'il y a d'ores et déjà un problème. Si on déplace le curseur de la barre de progression avec la souris, la lecture du media continue et déplace la position du curseur en même temps que l'utilisateur, ce qui donne l'impression d'avoir deux curseurs, puisqu'il se déplace rapidement entre les deux positions.

Pour corriger ce problème on va modifier le code de la tâche précédente en ajoutant une garde sur la présence d'un signal local que l'on crée pour l'occasion : `cant_update`. La barre de progression ne pourra pas être mise à jour tant que ce signal sera présent.

```
||
let cant_update = () in
loop begin present cant_update nothing
    (present media_time (atom(
         update_slider !!progress !!media_time
       )))
end
```

Il nous faut aussi prendre en compte les réactions de la barre de progression avec deux signaux : `seek_on` et `seek_off` qui nous donnent l'information que l'utilisateur commence à déplacer le curseur ou arrête de le déplacer. On ajoute aussi le composant `media` comme signal d'entrée. Le code ci-dessous est ajouté en parallèle des deux tâches précédentes.

```
||
loop begin
  await seek_on;
  trap t (
    loop begin
      emit cant_update ();
      present seek_off (
        atom (update_media !!media !!seek_off);
        exit t
      ); pause
    end
  ); pause
end
```

Dans cette dernière tâche, on attend le signal `seek_on`. Quand il apparaît, on entre dans une tâche en continu qui émet `cant_update` pour bloquer la mise à jour de la barre de progression. Si le signal `seek_off` est présent pendant cette tâche, on en déduit que l'utilisateur a relaché le curseur. On met alors à jour le temps courant de l'audio avec la dernière valeur choisie transportée par le signal `seek_off` et on quitte cette tâche en s'échappant jusqu'au `trap t` avec `exit t`. L'assemblage de ces trois tâches en parallèle nous donne le comportement attendu pour nos quatre composants Web. Il ne reste plus qu'à connecter chaque signal depuis JavaScript avec les fonctions de rappel des évènements cités.

Dans cette partie, on a observé que la programmation synchrone apporte un moyen élégant et expressif pour résoudre les problèmes d'interactions. L'avantage apporté par l'opérateur `||` est non négligeable puisqu'il permet d'écrire un programme incrémentalement, d'autant que la composition de comportements concurrents programmation Web n'est pas du tout facile à programmer. Le fait que l'ordre d'exécution des tâches soit calculé automatiquement simplifie le schéma de programmation et évite les erreurs, en ayant la garantie supplémentaire que le programme est correct du point de vue de la causalité.

6. Travaux connexes

Le travail le plus proche dans le domaine ML est ReactiveML qui est un langage de la famille ML qui se compose d'une partie algorithmique et d'une partie réactive selon le modèle de Boussinot [4]. Cette dernière caractéristique permet d'éviter l'analyse de causalité exigée en Esterel grâce à la contrainte de ne pas pouvoir réagir à l'absence d'un signal à l'instant courant. Du point du vue de l'implémentation, le code OCaml est généré après un langage intermédiaire de continuations et l'ordonnancement est dynamique. Cela apporte à ReactiveML une certaine richesse dans l'expression de la concurrence, et même la possibilité d'être exécuté en parallèle [10]. En terme de vérification statique, ce langage nous propose une analyse de réactivité des processus basée sur un système de type et d'effets [11]. Il est tout de même à noter que la compilation avec un ordonnancement statique implémentée dans `pendulum` a été expérimentée pour ReactiveML dans le but d'améliorer ses performances [7]. En comparaison, l'extension `pendulum` propose une méthode d'intégration d'un langage synchrone dans un monde asynchrone généraliste préexistant uniquement par la syntaxe là où ReactiveML propose un langage à part entière compilant vers OCaml. L'ordonnancement statique quand à lui apporte un gain de performances à l'exécution et il est totalement adapté au contexte du Web où il n'y a pas de parallélisation en général.

Dans le domaine, du Web, le projet Ocsigen a une philosophie assez proche, en proposant une compatibilité avec la bibliothèque React [5] en OCaml et la génération et la modification du DOM. Cette bibliothèque apporte un module de programmation en flot de données permettant de relier des composants de la page à un *signal* qui changeront automatiquement quand le signal changera. Même si `pendulum` a le même objectif, l'utilisation des deux solutions n'est pas mutuellement exclusive. Jusque là, la concurrence et la gestion des interactions dans la page était plutôt prise en main par la bibliothèque de threads légers LWT.

Pour finir, Hop est un langage Web multi-niveau basé sur Scheme muni d'une extension synchrone HipHop [3]. Ces travaux nous ont inspiré l'idée de l'extension de syntaxe et du modèle de programmation hybride. L'extension HipHop est implantée via le système de macros de Scheme. Il permet de décrire des machines synchrones et génère un arbre de syntaxe abstraite qui est interprété en suivant la sémantique constructive d'Esterel. L'ordonnancement est dynamique, basé sur des continuations, par opposition à `pendulum`. Grâce à la réflexivité des langages dynamiques, les machines synchrones sont vues comme des valeurs et peuvent être évaluées, construites à la volée et même transmises entre client et serveur. Cet aspect de construction dynamique des machines n'est pas possible dans `pendulum` dont l'analyse de causalité et la génération de code son complètement statiques.

7. Conclusion

Nous avons présenté dans cet article le langage `pendulum`[4], une extension réactive-synchrone d'OCaml. La programmation en `pendulum` ajoute de l'expressivité et des garanties pour résoudre les problèmes de concurrence du client Web. En plus de profiter du typage fort pour écrire des programmes réactifs, il est possible d'intégrer cette extension facilement dans tout projet OCaml compatible avec la fonctionnalité PPX. La taille du code généré est raisonnable et sa nature séquentielle avec ordonnancement statique font qu'un programme `pendulum` consomme peu de ressources supplémentaires. Le code est généré avant l'étape de typage, mais en reportant les positions des identifiants correctement il est possible de donner des messages d'erreurs assez clairs. En particulier le type des entrées est inféré par OCaml depuis le code généré. Ensuite, la compilation des nœuds se fait séparément même s'il est possible d'utiliser la définition d'un programme réactif dans un autre grâce à `run` ; l'analyse de causalité est pour l'instant effectuée au niveau de chaque programme.

4. Ces travaux font partie du projet UCF et sont partiellement financés par le pôle SYSTEMATIC PARIS-REGION et soutenus par l'IRILL.

Les axes d'amélioration concernent d'une part le cœur du langage, comme par exemple la gestion de plusieurs émissions d'un même signal et d'autre part l'interface avec JavaScript. Il est nécessaire d'engendrer automatiquement le code interfaçant les évènements du navigateur avec les signaux pour réduire les erreurs et la verbosité.

Confronter `pendulum` à une application conséquente vis-à-vis de la gestion du temps réel et de l'orchestration est maintenant nécessaire pour expérimenter à l'échelle. Pour cela, on s'oriente vers un *MOOC* autour de la programmation, ce qui demande la mise en œuvre d'une architecture complexe, avec de fortes interactions et des calculs qui tournent client en même temps que des communications avec le serveur. Pour accomplir cela, il devient essentiel de rendre `pendulum` compatible avec une programmation client-serveur dans un *framework* à la Ocsigen et ainsi expérimenter les qualités du modèle synchrone dans un environnement multitier, où les communications sont asynchrones entre le client et le serveur.

Références

[1] V. Balat. Rethinking Traditional Web Interaction. In *Proceedings of the Eighth International Conference on Internet and Web Applications and Services*, Rome, Italy, 2013.

[2] G. Berry. The foundations of esterel. In *Proof, Language, and Interaction, Essays in Honour of Robin Milner*, pages 425–454, 2000.

[3] G. Berry and M. Serrano. Hop and hiphop : Multitier web orchestration. *CoRR*, abs/1312.0078, 2013.

[4] F. Boussinot. Reactive C : an extension of C to program reactive systems. *Softw., Pract. Exper.*, 21(4) :401–428, 1991.

[5] D. Bünzli. React. http://erratique.ch/logiciel/react, 2010.

[6] Facebook. React. http://facebook.github.io/react/, 2015.

[7] L. Jachiet. Compilation de ReactiveML. Master's thesis, École normale supérieure, 2013.

[8] X. Leroy, D. Doligez, A. Frisch, J. Garrigue, D. Rémy, and J. Vouillon. The OCaml system release 4.02 : Documentation and user's manual. Interne, Inria, Sept. 2014.

[9] L. Mandel. *Conception, Sémantique et Implantation de ReactiveML : un langage à la ML pour la programmation réactive*. PhD thesis, Université Paris 6, 2006.

[10] L. Mandel and C. Pasteur. Exécution efficace de programmes ReactiveML. In *JFLA 2014 - Vingt-cinquièmes Journées Francophones des Langages Applicatifs*, Fréjus, France, Jan. 2014.

[11] L. Mandel and C. Pasteur. Reactivity of cooperative systems. In *Proceedings of 21st International Static Analysis Symposium (SAS'14)*, Munich, Germany, Sept. 2014.

[12] Ocsigen. js_of_ocaml manuel. http ://ocsigen.org/js_of_ocaml/manual/, 2015.

[13] D. Potop-Butucaru. *Compiling Esterel*. Springer, Berlin, 2007.

[14] M. Serrano. HOP : an environment for developing web 2.0 applications. In *International Lisp Conference, ILC 2007, Cambridge, April 1-4, 2007*, page 6, 2007.

[15] J. Vouillon. Lwt : a cooperative thread library. In *Proceedings of the ACM Workshop on ML, 2008, Victoria, BC, Canada, September 21, 2008*, pages 3–12, 2008.

[16] W3C. Timing control for script-based animations. http://www.w3.org/TR/animation-timing/, 2015.