Programmation synchrone aux JFLA

Guillaume Baudart[1], Louis Mandel[1], et Marc Pouzet[2]

[1] IBM Research
[2] Sorbonne Universités, UPMC Univ. Paris 06
École normale supérieure, PSL University
Inria Paris

Résumé

Depuis 1999, 39 articles et exposés liés à la programmation synchrone ont été présentés aux JFLA. Ces articles couvrent de nombreux aspects qui illustrent les liens étroits qui existent entre la programmation synchrone et les langages applicatifs : conception de langages, sémantique, typage, compilation, exécution, analyse de programmes, certification de compilateurs. Dans cet article nous revenons sur quelques uns de ces résultats qui illustrent la proximité des deux domaines.

1 Contexte historique

Les langages de programmation synchrones sont dédiés à la conception et l'implémentation de systèmes concurrents. Ils reposent sur un modèle de temps et de parallélisme idéal qui fait l'hypothèse que l'exécution du système est rythmée par une horloge globale commune à tous les processus [1].

L'appel à communication des Journées Francophones des Langages Applicatifs (JFLA) indique qu'≪ elles ont pour ambition de couvrir les domaines des langages applicatifs, de la preuve formelle, de la vérification de programmes, et des objets mathématiques qui sous-tendent ces outils ≫. Le développement des langages synchrones s'est confronté à toutes ces problématiques.

Nous avons identifié 39 articles, cours ou exposés invités présentés aux JFLA de 1999 (début des archives en ligne) à 2019 qui se rapportent à la programmation synchrone [2–40] [1]. Ces articles traitent entre autre des thèmes suivants.

Modèles et langages de programmation La conférence invitée qui ouvrait les JFLA en 1999 [2] présentait le langage Lucid Synchrone [41], un langage synchrone flot de données étendu avec des principes de ML. D'autres articles étendant ce modèle synchrone flot de données ont été présentés aux JFLA [18, 20, 29]. Nous reviendrons sur Lucid Synchrone dans la section 2. Toujours en suivant une approche flot de données, il y a eu des articles présentant des langages pour l'exécution parallèle [4,11,17,23,24], le traitement du signal [7], les circuits [25], les microcontrôleurs [33,38], ou l'analyse de logs [36]. Mais la programmation synchrone ne se limite pas à l'approche flot de données. Il y a aussi une approche plus impérative avec ReactiveML [16,21,31] et Pendulum [32] ou par réécriture avec MGS [10]. Enfin, il y a aussi des travaux plus théoriques comme ceux sur les fonctions causales [35] ou les domaines spatio-temporels [37].

Typage Les langages synchrones sont généralement des langages fortement typées (typage de données). Ils bénéficient également d'analyses de typage dédiés. Par exemple, les analyses de causalité [9] ou de réactivité [27] garantissent que les programmes produisent des données. Des analyses sur les rythmes de production et de consommation de flots peuvent garantir une exécution en mémoire bornée [18,20,22,29]. Enfin, des systèmes de types peuvent aider à générer du code efficace [14, 16].

1. Veuillez nous excusez pour les articles que nous avons oublié d'inclure dans cette liste.

Compilation et exécution La notion d'exécution concurrente est au coeur du modèle synchrone, les bibliothèques comme BSML [23] profitent de cette concurrence pour avoir une exécution parallèle. Mais cette concurrence peut aussi être exécutée séquentiellement en étant ordonnancée dynamiquement par des bibliothèques de runtime [12, 26, 28, 30]. Enfin, la concurrence peut également être compilée. Par exemple, Caspi et Pouzet [2, 18] ou Kieburtz [8] montrent comment utiliser des techniques de co-itération pour compiler des programmes flot de données en des fonctions de transitions efficaces.

Vérification et test Les langages synchrones sont souvent utilisés pour réaliser des applications dans des domaines critiques comme l'avionique où l'absence de bogues est cruciale [13]. Ces langages sont utilisés dans ce cadre car le modèle de concurrence synchrone a l'avantage d'être à la fois simple à modéliser mathématiquement et est expressif. Il peut par exemple être utilisé pour modéliser des systèmes asynchrones [6]. Ainsi, le test et la vérification formelle, qui sont aussi des thèmes important des JFLA, sont largement étudiés à la fois pour la vérification des outils et des applications.

Frédéric Gava a présenté une librairie certifiée de BSML en Coq [15]. Dans [19], nous avons prouvé en Coq la correction de l'abstraction qui est à la base du système de type qui permet de garantir qu'un programme synchrone peut s'exécuter en mémoire bornée. Timothy Bourke *et. al.*, travaillent sur la preuve formelle complète d'un compilateur Lustre [34, 40].

GATeL [5] est un outil qui permet de générer automatiquement des tests à partir de programmes Lustre. Enfin, Grégoire Hamon [3] a montré comment programmer un environnemt de simulation de programmes synchrone en utilisant un langage synchrone.

Dans cet article, nous revenons sur le mélange entre programmation synchrone et programmation fonctionnelle typée pour le développement de systèmes réactifs, et illustrons deux approches présentées dans les éditions passées des JFLA : l'intégration de traits fonctionnels dans un langage synchrone à flot-de-données, d'une part (Section 2) ; l'intégration du parallélisme synchrone dans un langage fonctionnel, d'autre part (Section 3). Chacune répond à un objectif différent. La première vise à augmenter l'expressivité et la modularité d'un langage synchrone en offrant les mêmes garanties, en particulier la possibilité de générer du code séquentiel s'exécutant en temps et mémoire bornés. La seconde, plus permissive, s'adresse à une classe plus vaste d'applications réactives, non nécessairement temps réel, en cherchant à faire cohabiter toute l'expressivité d'un langage fonctionnel, et le parallélisme synchrone. Dans cet article, nous limitons nos citations principalement aux travaux présentés aux JFLA ignorant ainsi une part importante des travaux reliés.

2 Lucid Synchrone : étendre Lustre avec des traits de ML

Le parallélisme synchrone a connu un grand succès dans la programmation des applications de contrôle/commande les plus critiques dès son introduction [42] : dans les avions (commande de vol, contrôle moteur, freinage), les trains (contrôle de bord, localisation), et les centrales (contrôle et arrêt d'urgence), etc [2]. Ces applications sont développées aujourd'hui avec SCADE [3], un environnement de développement qui s'appuie sur Scade 6, un langage synchrone flot-de-données dont le compilateur répond aux normes les plus strictes de l'avionique civile [43].

Dans un programme flot-de-données, un signal à temps discret est représenté par une suite infinie de valeurs et un système (ou schéma-bloc) est une fonction sur des suites. Le système est

2. Cf. cours de Gérard Berry (http://www.college-de-france.fr/site/gerard-berry/course-2012-2013)
3. http://www.esterel-technologies.com/products/scade-suite/

temps	0	1	2	3	4	5	6	...
x	x_0	x_1	x_2	x_3	x_4	x_5	x_6	...
y	y_0	y_1	y_2	y_3	y_4	y_5	y_6	...
$x + y$	$x_0 + y_0$	$x_1 + y_1$	$x_2 + y_2$	$x_3 + y_3$	$x_4 + y_4$	$x_5 + y_5$	$x_6 + y_6$	...
`pre` y	nil	y_0	y_1	y_2	y_3	y_4	y_5	...
$x \rightarrow y$	x_0	y_1	y_2	y_3	y_4	y_5	y_6	...
x `fby` y	x_0	y_0	y_1	y_2	y_3	y_4	y_4	...

FIGURE 1 – Les primitives flot-de-données

synchrone parce que tous les calculs sont datés par une horloge globale commune. À un dessin correspond une spécification mathématique précise sous la forme d'un ensemble d'équations. La programmation synchrone se trouve à la confluence de plusieurs courants d'idées : 1/ des questions anciennes de sémantique et les travaux de Gilles Kahn sur les réseaux de processus communicant par FIFOs ; 2/ la transcription d'automatismes du continu à l'échantillonné ; 3/ la programmation fonctionnelle, en particulier les techniques d'élimination de structures intermédiaires telles que la *déforestation* de Wadler et la modélisation des structures de données infinies. Or le langage Lustre, par exemple, est un petit langage fonctionnel manipulant des suites ; il est de premier ordre et sans récursivité. En étudiant les liens entre programmation synchrone et programmation fonctionnelle [44], nous cherchions à répondre à deux types de questions : peut-on, d'une part, faire bénéficier le temps réel de facilités de programmation des langages fonctionnels comme la synthèse des types ou l'ordre supérieur ; et, d'autre part, la programmation en général peut-elle bénéficier de l'efficacité apportée par le synchronisme ?

Le langage Lucid Synchrone [41], imaginé par Paul Caspi [45] a été conçu comme outil d'exploration de ces questions, c'est-à-dire un "laboratoire" dans lequel implémenter de nouvelles constructions de programmes ou analyses statiques [2, 44]. Des constructions nouvelles ont été explorées : la réinitialisation modulaire [46] a été le point de départ du travail de Grégoire Hamon sur l'ajout de structures de contrôle à Lustre [47] qui a abouti à une proposition pour combiner des équations flot-de-données avec des automates hiérarchiques [48]. Les analyses statiques telles que l'analyse des boucles causales réalisée par Pascal Cuoq [9, 49], le calcul d'horloges [44], exprimés tous deux sous forme d'un système de types dédiés. Puis une analyse par typage pour vérifier l'absence d'erreurs d'initialisation [50]. Toutes ces extensions ont été développées dans le cadre d'un langage synchrone fonctionnel plus expressif que Lustre, dans lequel les fonctions de suites pouvaient être des citoyens de première classe. Le langage s'est développé continûment, de 1996 à 2006, avec trois principales versions.

Exemple : le pendule inversé

Écrivons un programme qui décrit le mouvement d'une tige tenue en équilibre sur un chariot qui peut se déplacer de gauche à droite [4]. On écrit d'abord une fonction auxiliaire dans un fichier `misc.ls` : une fonction qui intègre un signal par la méthode d'Euler implicite.

```
(* module Misc *)
(* backward Euler: x(0) = x0(0); x(n) = x(n-1) + x'(n) * h(n) *)
let node backward_euler (h, x0, x') = x where
  rec x = x0 -> (pre x) +. x' *. h
```

La fonction `backward_euler (h, x0, x')` a trois arguments qui sont des suites infinies : `h` est le

4. C'est une variation du premier vrai exemple écrit avec la toute première version du compilateur, en 1996.

pas d'intégration, `x0` donne la valeur initiale de la sortie ; `x'` est le signal à intégrer. La fonction calcule la suite `x`, résultat de l'intégration de `x'` et défini par une équation de suite. **`pre`** `x` désigne la valeur de `x` à l'instant précédent. Ainsi, `x` peut être calculé séquentiellement car sa valeur à un instant n ne dépend pas d'elle-même. La figure 1 décrit les primitives de base de Lucid Synchrone.

Dans un fichier `pendulum.ls`, on peut instancier le noeud `backward_euler` pour avoir un intégrateur à pas fixe :

```
let h = 0.02 (* pas *)
let node integr(x0, xprime) = Misc.backward_euler(h, x0, xprime)
```

Le pendule est de longueur `l`, de masse `m`. Nommons `g` la gravitation, `h`, le pas d'échantillonnage et `b` le coefficient de friction entre le chariot et le pendule.

```
let l = 0.5  (* longueur du pendule *)
let m = 1.1  (* masse du pendule *)
let g = 9.8  (* gravitation *)
let b = 0.5  (* frottement *)

(* Pendulum with friction *)
let node pendulum (theta0, x'') = theta where
  rec theta = integr(theta0,
                     integr(0.0, sin (pre theta) *. g *. m -. cos (pre theta) *. x'')
                       /. l -. b *. pre theta)
```

La fonction `pendulum (theta0, x'')` calcule l'angle `theta` du pendule avec la verticale lorsque celui-ci démarre avec un angle `theta0`, une vitesse angulaire initiale nulle et est soumis à une force horizontale continue `x''` à la base du pendule.

Définissions maintenant le comportement du chariot qui maintient le pendule et sur lequel on peut transmettre les commandes `Left` ou `Right` pour appliquer une force vers la gauche ou la droite.

```
type cart_pole = { cart_position: float; pole_angle: float; }
type action = Left | Right | No

let node cart_pole (state_init, action) =
  let force = match action with Right -> 10. | Left -> -. 10. | No -> 0. end in
  let x'' = force /. m in
  let x = integr (state_init.cart_position, integr (0.0, x'')) in
  let theta = pendulum (state_init.pole_angle, x'') in
  { cart_position = x; pole_angle = theta; }
```

Comme en OCaml, on peut définir des types structurés. Le noeud `cart_pole(state_init, action)` calcule la position du chariot et l'angle du pendule à partir d'un état initial `state_init` et d'un flot de commande `action`.

Supposons que nous disposions d'une fonction `random_state` qui crée un état initial aléatoire et d'une fonction `draw` qui affiche le chariot avec le pendule. Nous pouvons alors définir une simulation paramétrée par un contrôleur qui envoie les commandes au chariot.

```
let node simulation controller =
  let state_init = random_state () in
  let rec a = run controller (state_init fby obs)
  and obs = cart_pole (state_init, a) in
  draw obs
```

h	t	t	t	t	t	t	t	...
x	x_0	x_1	x_2	x_3	x_4	x_5	x_6	...
c	t	t	f	t	f	t	t	...
x `when` c	x_0	x_1		x_3		x_5	x_6	...
z			z_0		z_1			...
`merge` c (x `when` c) z	x_0	x_1	z_0	x_2	z_1	x_3	x_4	...

FIGURE 2 – Les opération de filtrage et de complétion

Définissons un contrôleur manuel qui utilise les touches `'l'` et `'r'` pour émettre les actions `Left` et `Right`.

```
let node manual (key, obs) =
  match key with
  | Some 'l' -> Left
  | Some 'r' -> Right
  | _ -> No
  end
```

Nous ne pouvons pas utiliser ce contrôleur directement pour instancier le simulateur. Pour satisfaire la signature attendue, nous créons un noeud anonyme qui prend en entrée uniquement le flot d'observation de la position du pendule et utilise le noeud `get_key` le lire les touches du clavier.

```
let node main () =
  simulation (fun obs => manual (get_key(), obs))
```

Si on dispose d'un noeud `automatic` qui contrôle automatiquement le chariot. On peut définir un noeud qui peut basculer entre un mode manuel et automatique en appuyant sur la touche `'s'`.

```
let node two_modes (key, obs) = action where
  rec automaton
      | Manual ->
          do action = manual (key, obs)
          until (key = Some 's') then Auto
      | Auto ->
          do action = automatic obs
          until (key = Some 's') then Manual
      end
```

La question des horloges : de Lucid Synchrone à Lucy-n

Le programmeur Lustre verra trois différences avec son langage favori : les types n'ont pas besoin d'être écrits (cela ne surprendra pas le programmeur OCaml !), il peut combiner arbitrairement équations de suites et automates hiérarchiques, et il peut écrire des fonctions en paramètre ou en résultat d'autres fonctions. En interne, le compilateur implémente quatre systèmes de types dédiés et produit des signatures pour chacune des variables déclarées et traduit les construction de haut niveau, par exemple les automates, dans un langage noyau flot-de-donnees reposant sur le mécanisme des *horloges*. En effet, comme en Lustre, une suite peut être filtrée selon une condition booléenne pour construire une sous-suite, et deux sous-suites peuvent être combinées pour construire une suite plus longue. Les deux opérations de Lucid Synchrone sont **`when`** et **`merge`**, décrites dans la figure 2. La combinaison de suites et de sous-

suites doit cependant être contrainte. Dans un langage synchrone flot-de-données, on impose que tous les calculs soient datés par rapport à un référentiel de temps global qui bat la mesure : à toute suite s est associée une horloge h qui est une suite booléenne, de sorte que s est présent, c'est-à-dire disponible, lorsque h est vrai. Ainsi, si `x` et `c` ont une horloge h (dans la figure 2, $h(n)$ est vraie), l'horloge h' de `x when c` est vraie lorsque c est présent et vrai. Le but du calcul d'horloge est de synthétiser une expression booléenne. Celui de Lucid Synchrone, comme celui de Lustre, rejette la situation où un opérateur reçoit une entrée qui ne doit pas être présente, et réciproquement ; deux situations qui, si elles arrivaient, obligeraient à insérer un buffer avec un risque de débordement à l'exécution. Ainsi, l'expression `(x when c) + x` est rejetée statiquement. Les expression d'horloges associées à chaque expression et qui sont calculées durant le typage sont exploitées pour générer du code séquentiel efficace [51].

La notion d'horloge, et son système de type associé ont eu un rôle clef dans Lucid Synchrone pour concevoir les constructions de haut niveau, tels que les automates par exemple, définir leur sémantique et leur compilation. Toutes les constructions de haut niveau sont traduites dans un noyau flot-de-donnée avec horloges.

L'idée du n-synchrone est d'assouplir les contraintes d'horloges pour permettre de composer des suites dont les horloges ne sont pas égales mais "synchronisables" au moyen d'un buffer borné. En somme, retrouver le style de programmation des réseaux de Kahn mais en conservant la sûreté des restrictions synchrones : calculer automatiquement par typage la taille des buffers et générer du code séquentiel à temps et mémoire bornés. Par exemple, si une fonction f produit une sortie un instant sur deux, ce que l'on peut décrire par la signature suivante (écrite dans le langage Lucy-n [20]) :

```
val f :: 'a -> 'a on (1 0)
```

où (1 0) désigne la suite binaire (booléenne) périodique infinie 1 0 1 0 ..., et une fonction :

```
val g :: 'a -> 'a on (1 1 0 0)
```

qui signifie que `g` produit une sortie aux instants 1 1 0 0 .... Le programme suivant :

```
let node not_synchronous(x) = f(x) + g(x)
```

est rejeté (en Lucid Synchrone) car l'horloge `'a on (1 0)` n'est pas égale à `'a on (1 1 0 0)`. Pourtant, il suffit d'insérer un buffer qui convertit la suite d'horloge `(1 1 0 0)` en `(1 0)` pour que le programme soit accepté. Il s'écrit ainsi en Lucy-n :

```
let node nsynchronous(x) = f(x) + buffer (g(x))

val nsynchronous :: 'a -> 'a on (1 0)
```

Une fois le calcul d'horloge effectué, l'horloge effective de lecture et d'écriture de `buffer` sont déterminés ainsi que sa taille, et on obtient un programme synchrone. Les travaux sur le n-synchrone ont été présentés a plusieurs reprises aux JFLA [19, 20, 22] et ils ont été étendus pour gérer des flots qui peuvent produire plusieurs valeurs par instants [29]. Ces travaux reprennent de la vigueur aujourd'hui sur le sujet de la génération de code parallèle multi-coeur.

Lustre et Lucid Synchrone reposent sur un modèle de temps discret synchrone. Ils ne permettent pas d'exprimer fidèlement ni de simuler le plus efficacement possible un modèle a temps continu, ni un système mixte ou *hybride* combinant du temps discret et du temps continu. On doit écrire une version approximée des équations différentielles, sous forme d'équations de suites, fixant ainsi très tôt le pas d'échantillonnage et le schéma d'intégration. Dans les étapes préliminaires de conception et les phases de test, il est utile de décrire des modèles mixtes, de plus haut niveau, en utilisant les outils de simulation et d'analyse les plus efficaces. Un

solveur numérique d'équations différentielles tel que SUNDIALS CVODE [5], par exemple, est d'une efficacité redoutable par rapport à sa transcription synchrone à pas fixe, ce qui signifie que l'on pourra simuler des dynamiques beaucoup plus compliquées et sur une période beaucoup plus longue. Le langage Zélus [52] [6] a été conçu dans le but de pouvoir écrire des modèles mixtes exécutables. C'est essentiellement une extension d'un langage synchrone, c'est-à-dire que l'on peut y écrire des fonctions de suite comme en Lustre et Lucid Synchrone, celles-ci ayant la même sémantique statique et dynamique, et des fonctions sur des signaux à temps continu définis par des équations différentielles. Un typage ad-hoc permet de cloisonner les deux mondes et le compilateur génère du code séquentiel lié à un solveur numérique sur étagère.

3 Le synchrone dans ML : ReactiveML

Plutôt que d'intégrer des traits fonctionnels dans un langage synchrone, une approche antagoniste propose d'étendre un langage fonctionnel généraliste avec des aspects synchrones : temps logique global, composition parallèle de processus, communication par signaux, et structures de contrôle temporel. ReactiveML [16, 21] est ainsi une extension synchrone d'OCaml.

Le modèle synchrone est en effet utile à la programmation de tout type de systèmes concurrents, pas seulement les systèmes temps-réels. Cette idée a été introduite à l'origine par Frédéric Boussinot dans les langages ReactiveC [53] (extension de C) et SugarCubes [54] (extension de Java).

Comparé à ces prédécesseurs, ReactiveML étend un langage fonctionnel. La simplicité de la sémantique du langage hôte a permis de formaliser les interactions entre les parties algorithmiques et réactives et ainsi d'avoir des processus et de signaux comme valeurs de première classe [16]. ReactiveML hérite de l'inférence de type d'OCaml qu'il étend pour garantir que les programmes ne contiennent pas de boucle instantanées. Cette analyse de réactivité [27] permet de vérifier statiquement que tous les instants logiques terminent. Enfin, le moteur d'exécution de ReactiveML repose sur l'utilisation de continuations [30].

ReactiveML a été utilisé pour la programmation d'applications variées [31,39] : simulateur de réseaux de capteurs, simulateur de système complexes, programmation musicale, art, interface interactive pour le jeux d'échecs, et interface conversationnelle [39].

Exemple : Les Boids

Introduits par Reynolds [55], les boids permettent de simuler le comportement de groupes d'animaux tels que les bancs de poissons ou les nuées d'oiseaux. Le comportement de chaque individu est modélisé par trois règles :

— cohésion : les boids s'attirent pour former des nuées.

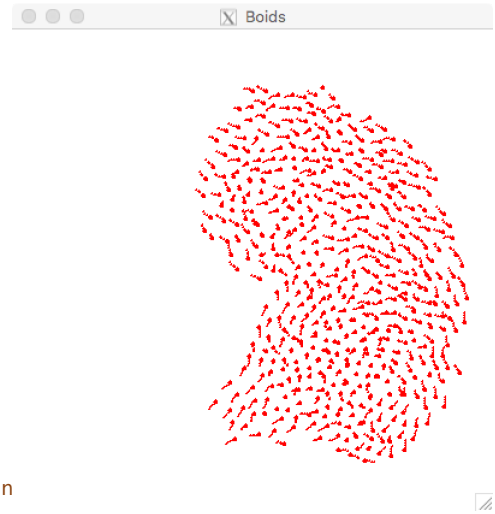
— séparation : les boids se repoussent pour éviter les collisions.

— alignement : les boids alignent leur vitesse pour avancer dans la même direction.

Chacune de ces règles est associée à un champ de vision différent. Un boid repousse ses plus proches voisins, s'aligne sur les membres de son groupe, et tente de rejoindre les boids plus éloignés. Malgré la relative simplicité de ce modèle, l'interaction des boids permet de faire émerger des comportements de groupe convaincants.

La Figure 3 présente le cœur de l'implémentation des boids en ReactiveML. Un boid est caractérisé par trois vecteurs (position, vitesse et accélération) et trois signaux qui correspondent

5. https://computation.llnl.gov/projects/sundials/cvode
6. http://zelus.di.ens.fr




```
type boid =
    { id: int;
      position: vect;
      speed: vect;
      acceleration: vect;
      s_separation: (boid * boid, vect) event;
      s_cohesion: (boid * boid, vect) event;
      s_alignment: (boid * boid, vect) event; }

let rec process boid me flock =
  emit flock me;
  await me.s_separation(f_separation)
     /\ me.s_cohesion(f_cohesion)
     /\ me.s_alignment(f_alignment) in
  let next_me = next_position me
    f_repulsion f_separation f_cohesion f_alignment in
  run boid next_me flock

let process main =
  signal flock, key default [] gather (fun x y -> x :: y) in
  do
    run window flock key ||
    control
      for i=1 to boids_number dopar
        run boid (new_random_boid ()) flock
      done ||
      loop
        await flock (all) in
        dispatcher all
      end
    with key(['p']) done
  until key(['q']) done
```

FIGURE 3 – Les Boids en ReactiveML

à chacune des forces qui s'appliquent sur le boid (séparation, cohésion et alignement). Les boids peuvent donc être représentés par le type enregistrement `boid`.

Le type des signaux de forces est `(boid * boid, vect) event`. Ce type signifie que l'on doit émettre des paires de boids sur le signal et que l'on va lire en réception du signal un vecteur représentant la force à appliquer. Les fonctions de transformation des pairs en boids en vecteurs de force est définie lors de la déclaration des signaux. Par exemple, la déclaration du signal `s_separation` est la suivante.

```
  signal separation
      default vzero
      gather (fun (me, boid) f_sep ->
        let sep = boid.position -: me.position in
        f_sep -: sep)
```

La valeur `vzero` est le vecteur $(0, 0)$. La fonction **`gather`** définie comment lors d'un instant combiner chaque valeur émise avec la valeur courante du signal. Ici, le valeur émise doit être

la pair `(me, boid)` où `me` est le boid sur lequel la force doit être appliquée et `boid` est un de ses voisins. `f_sep` est la valeur courante du signal en construction (c'est-à-dire `vzero` avant la première émission pendant l'instant, puis le résultat de l'appel de la fonction lors de le précédente émission). Le corps de la fonction calcule le vecteur `sep` entre les deux boids (`-:` est la différence entre deux vecteurs) et soustrait cette valeur à `f_sep`.

Le comportement d'un boid est implémenté par le processus `boid` (Figure 3). Un boid émet son état (`me`) sur le signal global `flock` puis récupère la valeur des signaux de force[7]. Il peut alors calculer la position suivante avant de relancer récursivement le processus avec le nouvel état pour le prochain pas de simulation.

La fonction de simulation `main` définie les signaux globaux `flock` et `key` et se décompose en trois parties exécutées en parallèle (opérateur `||`). 1/ Le processus `window` gère l'affichage graphique de la nuée en récupérant les positions des boids sur le signal `flock` et émet le signal `key` lorsqu'une touche du clavier est enfoncée. 2/ Une boucle **`for/dopar/done`** qui exécute en parallèle `boids_numbers` processus `boid`. 3/ Une boucle infinie (**`loop/end`**) qui à chaque émission de valeurs sur le signal `flock`, exécute la fonction `dispatcher` sur l'ensemble de la nuée. Cette fonction calcule les paires de boids qui sont à portée de vue et émet ces paires sur les signaux de force des boids concernés.

Le corps du processus `main` est englobé dans une construction de préemption (**`do/until`**). Ainsi, lorsque le signal `key` est émis avec la valeur `['q']` le programme se termine. La construction **`control/with`**, qui englobe les boids et la fonction de dispatche, suspend et reprend l'exécution de son corps lorsque le signal `key` est émis avec la valeur `['p']`. Cela permet de mettre la simulation en pause.

Un des points singulier de ReactiveML est la création dynamique de processus qui est possible par le mélange de récursion et de composition parallèle. Illustrons cela avec un processus `add_boids` qui crée un nouveau boid à chaque fois qu'un signal `new_position` est émis.

```
let rec process add_boid new_position flock =
  await new_position (position) in
  run add_boid new_position flock ||
  await immediate flock;
  run boid (new_boid position) flock
```

Ce processus est donc paramétré par le signal `new_position` qui déclenche la création d'un nouveau boid et le signal `flock` sur lequel le nouveau boid va communiquer avec les autres boids. C'est un processus récursif qui attend le signal `new_position` pour ensuite en parallèle exécuter un nouveau boid (on attend l'émission du signal `flock` avant d'exécuter `boid` pour s'assurer qu'il n'y a pas de décalage de phase entre les boids) et faire un appel récursif pour se mettre en attente à nouveau sur le signal `new_position`.

4 Aspects dynamiques dans les langages synchrones

On peut se poser la question de savoir si Lucid Synchrone et ReactiveML sont des langages si différents. Premièrement, nous avons vu qu'ils sont de natures différentes : Lucid Synchrone est un langage flot de données alors que ReactiveML est fondée sur Esterel qui est impératif. Mais l'ajout d'automates à Lucid Synchrone a rapproché les deux modèles comme cela a été illustré par Scade 6.

L'exemple du processus `add_boid` a montré que l'on peut créer dynamiquement des processus en ReactiveML. On peut se demander s'il est possible de faire la même chose en Lucid

7. **`await`** `s(x)` **`in`** `e` lie la valeur du signal `s` à la variable `x` dans l'expression `e`

```
Process INTEGERS out QO;
     Vars N ; 1 -> N ;
     repeat INCREMENT N ; PUT(N, QO) forever
Endprocess ;
Process FILTER PRIME in QI out QO ;
     Vars N ;
     repeat GET(QI) -> N ;
            if (N MOD PRIME) <> 0 then PUT(N, QO) close
     forever
Endprocess ;
Process SIFT in QI out QO ;
     Vars PRIME ; GET(QI) -> PRIME ;
     PUT (PRIME, QO) ; Comment emit a discovered prime ;
     doco queues Q ;
       FILTER(PRIME, QI, Q) ; SIFT(Q, QO)
     closeco
Endprocess ;
Process OUTPUT in QI ; Comment this is a library process ;
     repeat PRINT(GET(QI))
Endprocess ;
start doco queues Q1 Q2 ;
     INTEGERS(Q1) ; SIFT(Q1,Q2) ; OUTPUT(Q2)
     closeco ;
```

FIGURE 4 – Crible d'Érathosthène de Kahn et MacQueen [56]

Synchrone ? Lucid Synchrone est fondé sur des idées des réseaux de Kahn, or dès 1976 Kahn et MacQueen donnait un exemple de réseau avec création dynamique [56]. Il s'agit du crible d'Érathosthène qui est reproduit Figure 4 où le processus `SIFT` utilise la composition parallèle et la récursion pour créer dynamiquement des processus. Lucid Synchrone autorise aussi la récursion, il est donc possible de réécrire ce programme :

```
let node integers () = n where rec n = 1 fby n + 1

let rec node sift n =
  let rec prime = n fby prime in
  let clock filter = (n mod prime) <> 0 in
  merge filter (sift (n when filter)) (true fby false)

let node output n = print_int n; print_string " "; flush stdout

let node main () =
  let n = integers () in
  let clock primes = sift n in
  merge primes (output (n when primes)) ()
```

Ainsi, même si Lucid Synchrone s'intéresse principalement à la conception de systèmes temps-réel, il ne se limite pas à ces derniers [8]. Néanmoins, on peut constater que même si le programme grossi dynamiquement, toutes les dépendances sont connues statiquement.

8. L'option `-realtime` du compilateur garantie l'exécution en temps et mémoire bornés.

En ReactiveML, les signaux sont des valeurs de première classe. Il est donc possible de changer dynamiquement les dépendances dans les programmes. On peut par exemple simuler la mobilité du π-calcul. Reprenons l'exemple du chapitre 9.3 de [57].

```
let process p x z = emit x z; run p'
let process q x = await x(y) in run q' y
let process r z = await z(v) in run r' v
let process mobility x z = run p x z || run q x || run r z
```

Trois processus `p`, `q` et `r` sont exécutés en parallèle. D'une part `p` et `q` peuvent communiquer en utilisant un signal `x` et d'autre part `p` et `r` communiquent par le signal `z`. Les processus `p` et `q` peuvent être définis de telle sorte que `q` et `r` puissent communiquer en utilisant `z`.

Ce type de dynamicité n'est pas possible en Lucid Synchrone. Mais il est possible d'étendre le langage avec une notion de canaux partagés qui sont des valeurs de première classe. Ces canaux ont des propriétés similaires aux signaux de ReactiveML. Lorsqu'ils sont déclarer, il faut associer une fonction indiquant comment combiner plusieurs écritures pendant le même instant.

```
share x default 0 gather (+)
```

La lecture de la valeur d'un canal partagé se fait toujours à travers un retard (`last !x`). Enfin, comme les canaux partagés sont des valeurs de première classe, ils peuvent échapper à leur portée. Ainsi, l'horloge du contenu des canaux est toujours l'horloge de base du programme pour pouvoir faire des écritures et lectures à tout instant.

Enfin, le dernier aspect dynamique est la suppression de processus. De part la nature impérative du langage et son runtime interprété, la suppression de processus en ReactiveML est très naturelle. Lorsqu'un processus termine, son contrôle n'est plus accessible et ses ressources peuvent être désallouées. En Lucid Synchrone, tous les flots sont infinis. Il est possible de simuler la terminaison avec un flot dont l'horloge devient fausse pour toujours mais cette condition ne peut pas être testée dynamiquement. Par ailleurs, le compilateur actuel de Lucid Synchrone, le code généré est ordonnancé statiquement. Cela crée du code très efficace, mais il est difficile de libérer toutes les ressources liées aux processus qui terminent.

5 Conclusion

Lucid Synchrone et ReactiveML sont deux exemples de langages synchrones fonctionnels combinant, chacun différemment, le parallélisme synchrone et les traits d'un langage *a la ML*. Ils se sont développés au moment où apparaissaient plusieurs langages embarqués en Haskell, notamment Lava [58] pour les circuits synchrones, Fran [59] puis FRP [60] pour la programmation de systèmes réactifs et des mécanismes pour plonger un langage dédié dans ML et le compiler par macro-génération [61]. Nous avons suivi une voie différente pour mieux maîtriser les programmes synchrones devant être acceptés — détecter par typage les programmes non réactifs ou ne respectant pas les constraintes d'horloges, par exemple — et générer du code efficace en exploitant les propriétés de la sémantique statique. ReactiveML a été utilisé dans plusieurs domaines inattendus, tels que la musique mixte [62]. De nombreuses constructions et principes introduits dans Lucid Synchrone ont été repris dans SCADE. Et plus récemment, dans le langage Stimulus [9] pour la simulation des exigences de haut niveau. Zélus est directement issu des travaux sur Lucid Synchrone et ReactiveML. Les travaux actuels portent sur l'extension probabiliste d'un langage synchrone et sa mise en oeuvre dans Zélus.

9. https://www.argosim.com

Remerciements Nous souhaitons remercier les organisateurs des JFLA de nous inviter à la trentième édition de la conférence. C'est un honneur de pouvoir y participer. Cela a toujours été un grand plaisir soumettre et venir à cette conférence : la qualité des rapports fournit des conseils qui nous permettent d'améliorer grandement nos articles, la qualité scientifique des exposés sont source d'inspiration, l'ambiance conviviale de la conférence en fait une raison supplémentaire de vouloir y revenir. Nous souhaitons également remercier tous nos coauteurs sans qui nous n'aurions pas pu être invité et avec qui cela a été un grand plaisir de travailler.

Références

[1] G. Berry. Real time programming : Special purpose or general purpose languages. *Information Processing*, 89 :11–17, 1989.

[2] Paul Caspi. Lucid synchrone. In Pierre Weis, editor, *Dixièmes Journées Francophones des Langages Applicatifs (JFLA 1999)*, Morzine-Avoriaz, France, February 1999. INRIA. Conférence invitée.

[3] Grégoire Hamon and Marc Pouzet. Un simulateur synchrone pour Lucid Synchrone. In Pierre Weis, editor, *Dixièmes Journées Francophones des Langages Applicatifs (JFLA 1999)*, Morzine-Avoriaz, France, February 1999. INRIA.

[4] Frédéric Loulergue. Extension du BSlambda-calcul. In Pierre Weis, editor, *Dixièmes Journées Francophones des Langages Applicatifs (JFLA 1999)*, Morzine-Avoriaz, France, February 1999. INRIA.

[5] Bruno Marre and Agnès Arnould. Génération automatique de séquences de tests à partir de descriptions Lustre : GATeL. In Catherine Dubois, editor, *Onzièmes Journées Francophones des Langages Applicatifs (JFLA 2000)*, Mont Saint-Michel, France, January 2000. INRIA.

[6] Frédéric Boniol, Gérard Bel, and Jack Foiseau. Modélisation et vérification de systèmes intégrés asynchrones dans le langage synchrone Lustre : application aux systèmes avioniques. In Catherine Dubois, editor, *Onzièmes Journées Francophones des Langages Applicatifs (JFLA 2000)*, Mont Saint-Michel, France, January 2000. INRIA.

[7] Jocelyn Serot. Un compilateur CAML $\rightarrow$ SYNDEX pour les applications de traitement de signal temps-réel distribués. In Catherine Dubois, editor, *Onzièmes Journées Francophones des Langages Applicatifs (JFLA 2000)*, Mont Saint-Michel, France, January 2000. INRIA.

[8] Richard Kieburtz. Coalgebraic techniques for reactive functional programming. In Catherine Dubois, editor, *Onzièmes Journées Francophones des Langages Applicatifs (JFLA 2000)*, Mont Saint-Michel, France, January 2000. INRIA. Conférence invitée.

[9] Pascal Cuoq and Marc Pouzet. Causalité modulaire dans un langage de flots synchrone. In Pierre Castéran, editor, *Douzièmes Journées Francophones des Langages Applicatifs (JFLA 2001)*, Pontarlier, France, January 2001. INRIA.

[10] Olivier Michel, Jean-Louis Giavitto, and Julien Cohen. MGS : transformer des collections complexes pour la simulation en biologie. In Laurence Rideau-Gallot, editor, *Treizièmes Journées Francophones des Langages Applicatifs (JFLA 2002)*, Anglet, France, January 2002. INRIA.

[11] Armelle Merlin and Gaétan Hains. La machine abstraite catégorique BSP. In Laurence Rideau-Gallot, editor, *Treizièmes Journées Francophones des Langages Applicatifs (JFLA 2002)*, Anglet, France, January 2002. INRIA.

[12] Julien Cohen, Olivier Michel, and Jean-Louis Giavitto. Filtrage et règles de réécriture sur des structures indexées par des groupes. In Jean-Christophe Filliatre, editor, *Quatorzièmes Journées Francophones des Langages Applicatifs (JFLA 2003)*, Chamrousse, France, January 2003. INRIA.

[13] Y. Ait Ameur, F. Boniol, S. Pairault, and V. Wiels. Analyse de robustesse de systèmes avioniques. In Jean-Christophe Filliatre, editor, *Quatorzièmes Journées Francophones des Langages Applicatifs (JFLA 2003)*, Chamrousse, France, January 2003. INRIA.

[14] Frédéric Gava and Frédéric Loulergue. Synthèse de types pour Bulk Synchronous Parallel ML. In Jean-Christophe Filliatre, editor, *Quatorzièmes Journées Francophones des Langages Applicatifs (JFLA 2003)*, Chamrousse, France, January 2003. INRIA.

[15] Frédéric Gava. Une bibliothèque certifiée de programmes fonctionnels BSP. In Valérie Ménissier-Morain, editor, *Quinzièmes Journées Francophones des Langages Applicatifs (JFLA 2004)*, Sainte-Marie-de-Ré, France, January 2004. INRIA.

[16] Louis Mandel and Marc Pouzet. ReactiveML, un langage pour la programmation réactive en ML. In Olivier Michel, editor, *Seizièmes Journées Francophones des Langages Applicatifs (JFLA 2005)*, Obernai, France, March 2005. INRIA.

[17] R. Benheddi and F. Loulergue. Sémantiques de MSPML avec composition parallèle. In Thérèse Hardin, editor, *Dix-septièmes Journées Francophones des Langages Applicatifs (JFLA 2006)*, Pauillac, France, January 2006. INRIA.

[18] Marc Pouzet. Programmation synchrone fonctionnelle. In Pierre-Etienne Moreau, editor, *Dix-huitièmes Journées Francophones des Langages Applicatifs (JFLA 2007)*, Aix-les-Bains, France, January 2007. INRIA. Cours.

[19] Louis Mandel and Florence Plateau. Abstraction d'horloges dans les systèmes synchrones flot de données. In Alan Schmitt, editor, *Vingtièmes Journées Francophones des Langages Applicatifs (JFLA 2009)*, Saint-Quentin sur Isère, France, January 2009. INRIA.

[20] Louis Mandel, Florence Plateau, and Marc Pouzet. Lucy-n : une extension n-synchrone de Lustre. In Micaela Mayero, editor, *Vingt et unièmes Journées Francophones des Langages Applicatifs (JFLA 2010)*, Vieux-Port La Ciotat, France, January 2010.

[21] Louis Mandel. Cours de ReactiveML. In Micaela Mayero, editor, *Vingt et unièmes Journées Francophones des Langages Applicatifs (JFLA 2010)*, Vieux-Port La Ciotat, France, January 2010.

[22] Louis Mandel and Florence Plateau. Typage des horloges périodiques en Lucy-n. In Sylvain Conchon, editor, *Vingt deuxièmes Journées Francophones des Langages Applicatifs (JFLA 2011)*, La Bresse, France, January 2011.

[23] Frédéric Gava and Sovanna Tan. Implémentation et prédiction des performances de squelettes data-parallèles en utilisant un langage BSP de haut niveau. In Sylvain Conchon, editor, *Vingt deuxièmes Journées Francophones des Langages Applicatifs (JFLA 2011)*, La Bresse, France, January 2011.

[24] Wadoud Bousdira, Frederic Loulergue, and Louis Gesbert. Syntaxe et sémantique de revised Bulk Synchronous Parallel ML. In Sylvain Conchon, editor, *Vingt deuxièmes Journées Francophones des Langages Applicatifs (JFLA 2011)*, La Bresse, France, January 2011.

[25] Thomas Braibant. De coquets circuits. In Sylvain Conchon, editor, *Vingt deuxièmes Journées Francophones des Langages Applicatifs (JFLA 2011)*, La Bresse, France, January 2011.

[26] Christophe Deleuze. Concurrence et continuations en OCaml. In Assia Mahboubi, editor, *Vingt troisièmes Journées Francophones des Langages Applicatifs (JFLA 2012)*, Carnac, France, January 2012.

[27] Louis Mandel and Cédric Pasteur. Réactivité des systèmes coopératifs : le cas de ReactiveML. In Damien Pous, editor, *Vingt-quatrièmes Journées Francophones des Langages Applicatifs (JFLA 2013)*, Aussois, France, February 2013.

[28] Christophe Deleuze. Concurrence légère en OCaml : muthreads. In Damien Pous, editor, *Vingt-quatrièmes Journées Francophones des Langages Applicatifs (JFLA 2013)*, Aussois, France, February 2013.

[29] Adrien Guatto and Louis Mandel. Réseaux de Kahn à rafales et horloges entières. In Christine Tasson, editor, *Vingt-cinquièmes Journées Francophones des Langages Applicatifs (JFLA 2014)*, Fréjus, France, January 2014.

[30] Louis Mandel and Cédric Pasteur. Exécution efficace de programmes ReactiveML. In Christine Tasson, editor, *Vingt-cinquièmes Journées Francophones des Langages Applicatifs (JFLA 2014)*, Fréjus, France, January 2014.

[31] Louis Mandel and Marc Pouzet. ReactiveML, un langage de programmation réactif. In Christine

Tasson, editor, *Vingt-cinquièmes Journées Francophones des Langages Applicatifs (JFLA 2014)*, Fréjus, France, January 2014. Exposé invité parmi les contributions marquantes des dix dernières années).

[32] Rémy El Sibaïe and Emmanuel Chailloux. Pendulum : une extension réactive pour la programmation Web en OCaml. In David Baelde, editor, *Vingt-sixièmes Journées Francophones des Langages Applicatifs (JFLA 2015)*, Val d'Ajol, France, January 2015.

[33] S. Varoumas, B. Vaugon, and E. Chailloux. OCaLustre : une extension synchrone d'OCaml pour la programmation de microcontrôleurs. In Julien Signoles, editor, *Vingt-huitièmes Journées Francophones des Langages Applicatifs (JFLA 2017)*, Gourette, France, January 2017. INRIA.

[34] T. Bourke, P.-É. Dagand, M. Pouzet, and L. Rieg. Vérification de la génération modulaire du code impératif pour Lustre. In Julien Signoles, editor, *Vingt-huitièmes Journées Francophones des Langages Applicatifs (JFLA 2017)*, Gourette, France, January 2017. INRIA.

[35] B. P. Serpette et D. Janin. Causalité dans les calculs d'événements. In Julien Signoles, editor, *Vingt-huitièmes Journées Francophones des Langages Applicatifs (JFLA 2017)*, Gourette, France, January 2017. INRIA.

[36] Guillaume Baudart, Louis Mandel, Olivier Tardieu, and Mandan Vaziri. Cloudlens, un langage de script pour l'analyse de données semi-structurées. In Julien Signoles, editor, *Vingt-huitièmes Journées Francophones des Langages Applicatifs (JFLA 2017)*, Gourette, France, January 2017. INRIA.

[37] David Janin. Domaines spatio-temporels : un tour d'horizon. In Sylvie Boldo, editor, *Vingt-neuvièmes Journées Francophones des Langages Applicatifs (JFLA 2018)*, Banyuls-sur-Mer, France, January 2018. INRIA.

[38] Steven Varoumas, Benoit Vaugon, and Emmanuel Chailloux. La programmation de microcontrôleurs dans des langages de haut niveau. In Sylvie Boldo, editor, *Vingt-neuvièmes Journées Francophones des Langages Applicatifs (JFLA 2018)*, Banyuls-sur-Mer, France, January 2018. INRIA. Cours.

[39] Guillaume Baudart, Louis Mandel, and Jérôme Siméon. Programmer des chatbots en OCaml avec Watson Conversation Service. In Sylvie Boldo, editor, *Vingt-neuvièmes Journées Francophones des Langages Applicatifs (JFLA 2018)*, Banyuls-sur-Mer, France, January 2018. INRIA.

[40] Timothy Bourke and Marc Pouzet. Arguments cadencés dans un compilateur Lustre vérifié. In Nicolas Magaud, editor, *Trentièmes Journées Francophones des Langages Applicatifs (JFLA 2019)*, Les Rousses, France, January 2019. INRIA.

[41] Marc Pouzet. *Lucid Synchrone, version 3. Tutorial and reference manual.* Université Paris-Sud, LRI, April 2006.

[42] A. Benveniste, P. Caspi, S.A. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone. The synchronous languages 12 years later. *Proceedings of the IEEE*, 91(1), January 2003.

[43] Jean-Louis Colaco, Bruno Pagano, and Marc Pouzet. Scade 6 : A Formal Language for Embedded Critical Software Development. In *Eleventh International Symposium on Theoretical Aspect of Software Engineering (TASE)*, Sophia Antipolis, France, September 13-15 2017.

[44] Paul Caspi et Marc Pouzet. Réseaux de Kahn Synchrones. In *Journées Francophones des Langages Applicatifs (JFLA)*, Val Morin (Québec), Canada, 28-30 janvier 1996.

[45] Paul Caspi. Lucid synchrone. In *Actes du colloque INRIA OPOPAC*, Lacanau, 1993.

[46] Grégoire Hamon and Marc Pouzet. Modular Resetting of Synchronous Data-flow Programs. In *ACM International conference on Principles of Declarative Programming (PPDP'00)*, Montreal, Canada, September 2000.

[47] Grégoire Hamon. *Calcul d'horloge et Structures de Contrôle dans Lucid Synchrone, un langage de flots synchrones à la ML.* PhD thesis, Université Pierre et Marie Curie, Paris, France, 14 novembre 2002.

[48] Jean-Louis Colaço, Bruno Pagano, and Marc Pouzet. A Conservative Extension of Synchronous Data-flow with State Machines. In *ACM International Conference on Embedded Software (EM-*

SOFT'05), Jersey city, New Jersey, USA, September 2005.

[49] Pascal Cuoq. *Ajout de synchronisme dans les langages fonctionnels typés*. PhD thesis, Université Pierre et Marie Curie, Paris, France, october 2002.

[50] Jean-Louis Colaço and Marc Pouzet. Type-based Initialization Analysis of a Synchronous Data-flow Language. *International Journal on Software Tools for Technology Transfer (STTT)*, 6(3) :245–255, August 2004.

[51] Darek Biernacki, Jean-Louis Colaco, Grégoire Hamon, and Marc Pouzet. Clock-directed Modular Code Generation of Synchronous Data-flow Languages. In *ACM International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, Tucson, Arizona, June 2008.

[52] Timothy Bourke and Marc Pouzet. Zélus, a Synchronous Language with ODEs. In *International Conference on Hybrid Systems : Computation and Control (HSCC 2013)*, Philadelphia, USA, April 8–11 2013. ACM.

[53] Frédéric Boussinot. Reactive C : An extension of C to program reactive systems. *Software Practice and Experience*, 21(4) :401–428, April 1991.

[54] Frédéric Boussinot and Jean-Ferdy Susini. The SugarCubes tool box : A reactive java framework. *Software Practice and Experience*, 28(4) :1531–1550, 1998.

[55] Craig Reynolds. Flocks, herds and schools : A distributed behavioral model. In *Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '87. ACM, 1987.

[56] Gilles Kahn and David Macqueen. Coroutines and Networks of Parallel Processes. Research report, 1976.

[57] Robin Milner. *Communicating and Mobile Systems : The π-Calculus*. 1999.

[58] Per Bjesse, Koen Claessen, Mary Sheeran, and Satnam Singh. Lava : Hardware Design in Haskell. In *International Conference on Functional Programming (ICFP)*. ACM, 1998.

[59] C. Elliot and P. Hudak. Functional reactive animation. In *International Conference on Functional Programming (ICFP)*. ACM, 1997.

[60] Zhanyong Wan and Paul Hudak. Functional reactive programming from first principles. In *International Conference on Programming Language, Design and Implementation (PLDI)*, 2000.

[61] Walid Taha. Multi-stage programming : Its theory and applications. Technical Report CSE-99-TH-002, Oregon Graduate Institute of Science and Technology, November 1999.

[62] Louis Mandel, Cédric Pasteur, and Marc Pouzet. ReactiveML, Ten Years Later. In *ACM International Conference on Principles and Practice of Declarative Programming (PPDP)*, Siena, Italy, July 2015. Invited paper for PPDP'05 award.