Lucid Synchrone:
une expérience d'extension de Lustre avec des traits venant de ML

Marc Pouzet
LIP6

Marc.Pouzet@lip6.fr

Les origines

Quelles sont les relations entre:

- réseaux de Kahn
- programmation synchrone data-flow (e.g., Lustre)
- programmation fonctionnelle paresseuse (e.g., Haskell)
- types et horloges
- machines a état et fonctions de suites

Observations

- on peut “simuler” le data-flow en fonctionnel paresseux (9 lignes)
- on programme dans la sémantique (dénotationnelle)
- on profite des traits du langage hôte: typage, modules, ordre supérieur, etc.
- pertinent (du point de vue pragmatique)

Applications

- langages de descriptions de circuits synchrones: Lava [Sheeran]
- Functional Reactive Programming [Paul Hudak, John Petterson]
- Fran (Functional Reactive Animation) [Conal Elliot]

Faiblesses

Programmation Fonctionnelle Réactive (FRP)

- accepte trop de programmes: synchrones et asynchrones
- pas de *garanties* à la compilation
- pas de compilation (ou alors macro-expansion)
- run-time Haskell (mais macro-expansion vers C possible)
- le système de types de Haskell n'est pas suffisant

Circuits Synchrones (Lava)

- très élégant et bien adapté aux circuits synchrones
- vision "two-stage": l'exécution du programme produit une net-list
- fonctions préservant les longueurs (une horloge de base uniquement)
- le système de type de Haskell n'est pas suffisant
- pas d'analyse des boucles de causalité, etc.

Retour au Synchrone traditionnel

Observations

- synthèse automatique des types: ça existe dans Scade (et Simulink)
- synthèse automatique des horloges: idem
- polymorphisme (utile pour écrire des librairies): idem
- mais tout ça est un peu ad-hoc
- compilation modulaire: peut-on l'expliquer simplement?
- ordre supérieur (écrire un noeud paramétré par un autre): le faire à la main

Questions

- utiliser des techniques conventionnelles de typage?
- types et horloges (suréchantillonnage, synthèse)?
- comment expliquer (simplement) la compilation modulaire?
- peut-on avoir un mécanisme propre d'abstraction générale (fonctions curryfiées, ordre supérieur)?
- définir un ensemble d'analyses de programmes modulaires?
- qu'est ce qui est spécifique la dedans?

Lucid Synchrone

Un langage laboratoire

- étudier des extensions de Lustre (synchrone fonctionnel)
- expérimenter jusqu'au bout et écrire des programmes!
- Version 1 (1995), Version 2 (2001), V3 (2005)

Sémantique

- Réseaux de Kahn synchrones [ICFP'96]
- Calcul d'horloge = types dépendants [ICFP'96]
- fonctions de suites synchrones et fonctions de transition (co-induction *vs* co-itération) [CMCS'98]
- Calcul d'horloge à la ML [Emsoft'02]
- analyse de causalité [ESOP'01]
- analyse d'initialisation [SLAP'03, STTT'04]
- Ordre supérieur général et typage [Emsoft'04]

Problèmes (durs)

Typage comment éviter d'avoir les couples de flot, les flots de couples, les fonctions de flot, les flots de fonctions instantanées (scalaires), les flots de fonctions de flots, etc ?

Horloges comment synthétiser automatiquement des horloges en ayant une logique simple (à la Lustre) mais permettant d'avoir un peu de sur-échantillonnage, plusieurs variables d'horloge et des signatures prévisibles (et compréhensibles)?

Analyses comment vérifier la causalité (même syntaxique), l'absence de certaines erreurs (initialisation) de manière rapide et modulaire?

Compilation comment éviter la compilation par expansion naive du code (compilation séparée, composants compilés une fois pour toute)? Spécialisation de code?

Structures de controle, Ordre supérieur

Quelques exemples (V3)

Fonctions combinatoires

```
let average (x,y) = (x + y) / 2

let average (x,y) = o where
  o = (x + y) / 2

let average (x,y) =
  let o = (x + y) / 2 in
  o

val average :  int * int -> int
val average :: 'a * 'a -> 'a
```

Fonctions séquentielles

```
let node min x = m where
  rec m = x -> if x < pre m then x else pre m

val min : int => int
val min :: 'a -> 'a

let node min_max x = (min, max) where
  rec min = x -> if x < pre min then x else pre min
  and max = x -> if x > pre max then x else pre max

val min_max :  int => int * int
val min_max :: 'a -> 'a
```

Dans la V3, on distingue les fonctions combinatoires (`->`) des fonctions séquentielles (`=>`).

Polymorphisme (réutilisation de code)

```
let node delay x = x -> pre x
let node edge x = false -> x <> pre x


val delay :  'a => 'a
val delay :: 'a -> 'a
```

En Lustre, le polymorphisme est limité à un jeu d'opérateurs prédéfini (e.g, `if/then/else`, `when`) et ne passe pas l'abstraction.

Librairie et Curryfication

```
(* module Numerical *)
let node integr dt x0 dx = x where
  rec x = x0 -> pre x +. dx *. dt

val integr :  float -> float -> float => float
val integr :: 'a -> 'a -> 'a -> 'a

(* module Pendulum *)
let static dt = 0.001

let integr = integr dt

val integr :  float -> float => float
val integr :: 'a -> 'a -> 'a
```

Rejeter des programmes

Rejeter des programmes qui ne peuvent être exécutés séquentiellement

```
let pendul d2x0 d2y0 = theta
  where rec theta =
        integr (integr ((sin theta)*.(d2y0 +. g)
                                ^^^^^^
                        -.(cos theta)*.d2x0)/.l)
thetap depends instantaneously on itself
```

- un critère syntaxique: récursion à travers un retard
- un système de types (avec rangées) [ESOP'01]
- donc, des signatures (interfaces)
- modulaire et ordre supérieur

Question: Peut-on faire mieux?

Rejeter des programmes

Rejeter les programmes dont le résultat dépend de retards non initialisés

```
let pendul d2x0 d2y0 = theta
  where rec theta =
        integr (integr ((sin (pre theta)*.(d2y0 +. g)
        ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
                       -.(cos (pre theta))*.d2x0)/.l)
                       ^^^^^^^^^^^^^^^^^^^^^^^^^^^^

this expression may not be initialized
```

- abstraction 1-bit pertinente
- un système de types (avec sous-typage) [SLAP'02, STTT'04]
- ça marche très bien pour Scade (discipline de programmation)

Question: Peut-on faire mieux?

Les horloges

- nommer les variables booléennes servant à construire des horloges
- se rapprocher (autant que possible) du typage standard

```
let hold init  c input  = output  where
  rec output  = merge c input  ((init  fby output) whenot c)

let clock half =
  let rec h = true -> not (pre h) in h

let over x = hold 1 half x
let stuttering = over nat

val over :: 'a on half -> 'a
```

Automates (Maraninchi)

Préemption forte

```
(* await x to be true and then sustain the value *)
let node await x = o where
  automaton
    S1 -> do o = false unless x then S2
  | S2 -> do o = true done
  end

node await :  bool => bool
node await :: 'a -> 'a
```

Préemption faible

```
let node switch x = o where
  automaton
    False -> do o = false until x then True
  | True -> do o = true until x then False
  end
```

Controleur de souris

```
let node counting e = cpt where
  rec cpt = if e then 1 -> pre cpt + 1 else 0 -> pre cpt

let node controler click top = simple, double where
  automaton
    Await -> do  simple = false and double = false
             until click then One
  | One ->
     do  simple = false and double = false
     unless click then Emit(false, true)
     unless counting top = 4 then Emit(true, false)
  | Emit(x1, x2) ->
     do simple = x1 and double = x2
     until true then Await
  end
```

Ordre supérieur statique et dynamique

Ordre supérieur statique

```
let node rectangle (dt, x0, dx) = x where
  rec x = x0 -> pre x +. dx *. dt

let node double (dt, x0, dx0, d2x) = x where
  rec dx = rectangle (dt, dx0, d2x)
  and x = rectangle (dt, x0, dx)

let node double law (dt, x0, dx0, d2x) = x where
  rec dx = law (dt, dx0, d2x)
  and x = rectangle (dt, x0, dx)

let double = double rectangle
```

Ordre supérieur général

L'ordre supérieur précédent est statique: la loi à intégrer est définie *statiquement*, c'est à dire au moment de l'instanciation du noeud et n'évoluera plus.

Question: Peut-on faire mieux?

Une proposition dans [Emsoft 2004] (Colaço, Girault, Hamon, Pouzet):

- flots de fonctions (de flots)
- reconfiguration dynamique (un composant peut recevoir/emettre une fonction)
- est-ce que ça suffit?