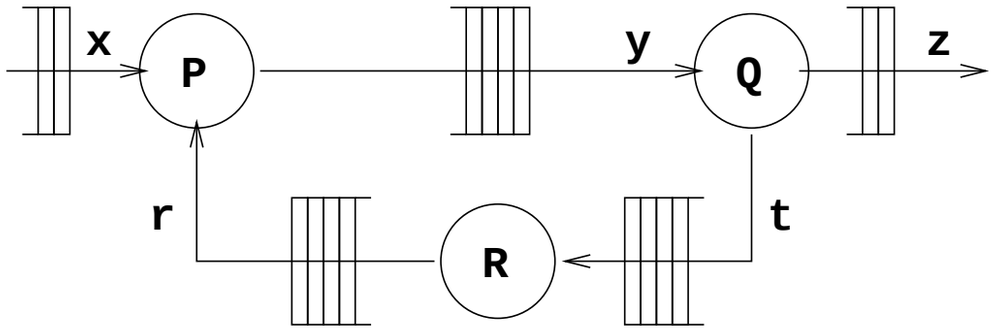
Sémantique de suites

Réseaux de Kahn [Kahn'74, Kahn'77]



- un ens. de processus déterministes (e.g., programmes séquentiels)
- communications par envois de messages (`send/wait`)
  - attente bloquante et envoi non bloquant (i.e., communication par FIFO)
  - canaux fiables, délai d'émission borné
  - attente sur un seul canal à la fois. Le programme:
    `si (A est présent) ou (B est présent) alors ...` est interdit
  - donc pas de "`select`" Unix

Principe de Kahn

- un processus élémentaire est une fonction de suites continue
- la sémantique d'un tel réseau est aussi une fonction continue (i.e., le pppf d'un sysème d'équations de suites)

Sémantique dénotationnelle

Domaine sémantique:

- D^∞, ens. des séquences finies et infinies d'éléments de D ($D^\infty = D^* + D^\omega$)
- D^∞ contient la séquence vide ϵ
- ordre préfixe $\leq$: $X \leq Y$ s'il existe Z tq $X \bullet Z = Y$ (concaténation)
- $(D^\infty, \leq)$ est un cpo (toute chaîne croissante admet un sup)

Interprétation:

- un processus est une fonction de l'*histoire* de ses entrées $X = x_1, ..., x_n, ...$ (suite des valeurs que l'on peut observer)
- un canal de type D est interprété par son histoire ($\in D^\infty$)

- un processus = un ensemble de fonctions de flots. A chaque noeud $P : D_1 \times ... \times D_n \rightarrow D'_1 \times ... \times D'_p$, on associe p fonctions continues:

  $f_1 : D_1^\infty \times D_n^\infty \rightarrow D_1'^\infty$

  ...

  $f_p : D_1^\infty \times D_n^\infty \rightarrow D_p'^\infty$

Monotonie: le calcul des sorties peut commencer sans que le processus dispose de toutes ses entrées

Continuité: un processus ne peut décider de produire une sortie après avoir attendu un infinité d'entrées

Exemples:

Fonction monotone (et continue): `fby` (et tout Lustre):

$$(x.xs)\ \texttt{fby}\ y = x.y$$

Fonction non monotone ($D = \{0, 1\}$):

$$\begin{aligned} f(0) &= 0 \\ f(0.1) &= 1 \end{aligned}$$

ou

$$f(\epsilon, x) = x$$
$$f(x.y, z) = x.f(y, z)$$

Fonction non continue ($D = \{0, 1\}$):

$$\begin{aligned} f(x) &= 0 \text{ où } x \text{ est une suite finie} \\ f(x) &= 01 \text{ si } x \text{ est une suite infinie} \end{aligned}$$

Remarques:

- $|x|$ est la longueur de la séquence x ($|.| : D^\infty \to I\!N^\infty$). $f : A^\infty \to B^\infty$ est causale ssi pour tous x, $|x| \leq |f(x)|$
- $f : A^\infty \to B^\infty$ est contractante ssi pour tous x (en part. pour ϵ), $|x| < |f(x)|$
- dans ce cas $fixf \neq \epsilon$

Propriétés

- **Point-fixe** (Kleene): un système d'équations

  $\vec{X} = F(\vec{X})$

  où F est une fonction continue admet une plus petite solution $(fix(X_1), ..., fix(X_n))$ où $fix(X_i) = lim_{k \to \infty}(X_i^k)$ tq:

  $\vec{X}^0 = (\epsilon, ..., \epsilon)$

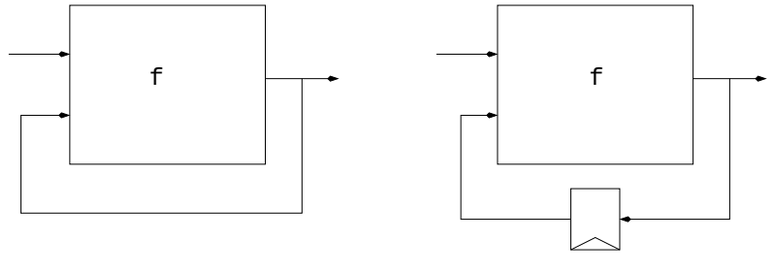
  $\vec{X}^{k+1} = F(\vec{X}^k)$

- **Modularité:** la plus petite solution d'un tel système est encore une fonction continue des paramêtres du système

- **Récursion:** les réseaux peuvent être dynamiques (réseaux définis récursivement): Kahn donne l'exemple du calcul du crible d'Eratosthène

Causalité (boucles)

Que se passe t'il lorsque la sortie d'un bloc est connectée à une des entrées (rebouclage)



- La plus petite solution des équations $X = X$ ou $X = X + 1$ est ϵ
- La plus petite solution de l'équation $X = \texttt{reg}_0(X) + 1$ (où + et 1 sont liftés aux suites) est $Nat = 0.1....n....$

Causalité "syntaxique"

De nombreux outils de type schéma/bloc (e.g., SCADE, SIMULINK) imposent une condition de non rebouclage direct, i.e., tous les cycles doivent traverser un retard (registre).

- Sémantique du retard (ici initilisé à 0): $\texttt{reg}_v(X) = v.X$
- Condition suffisante: si $\epsilon < f(\epsilon)$ alors $fix\ f \neq \epsilon$

- i.e., la causalité est basée sur un argument de longueur. Peut-être vérifié par typage
- Rmq: dualité avec la caractérisation des fonctions “strictes” dans le λ-calcul: f est stricte ssi $f(\perp) = \perp$, i.e., f a besoin de son argument pour produire un résultat.

Intérêts/faiblesses du modèle

Ce modèle connaît actuellement un regain d'intérêt très fort dans l'embarqué: traitement vidéo intensif (médical, TV), Systèmes sur puce (SOC), modélisation, etc.

- **fonctionnel:** sémantique simple, parallélisme déterministe
- **Modularité, compositionnalité**:
- **insensibilité au temps:** (ralentir une entrée ne modifie pas le comportement du système)
- **architecture GALS:** (Globally Asynchronous, Locally Synchronous)

Mais:

- **insensibilité au temps:** (on ne sait predire le temps de calcul)
- **pas de garantie de mémoire:**, taille des buffers eventuellement non bornée (aucun sens en hardware, peu satisfaisant en software)
- **pas de test à l'absence**
- **efficacité:** ordonnancement dynamique (e.g., [Park 92]). Quid de l'ordonnancement statique?

Réseaux de Kahn Synchrones

Et Lustre ?

- programmer directement à partir d'équations sur des flots, sémantique de Kahn
- opérations continues (e.g., délai, initialisation, opération point-à-point, échantillonnage)
- un (petit) sous-ensemble des réseaux de Kahn seulement
  - statiques (pas de récursivité)
  - synchrones: il existe une exécution à buffer nul! (i.e., un ordonnancement des calcul ne nécessitant pas de synchronisation)
  - autre modèle: SDF (Synchronous Data-Flow, Edward Lee, Berkeley)

Remarque:

- Les suites peuvent être représentées par des structures paresseuses
- On se plonge dans un langage fonctionnel plus général

Un noyau data-flow

$$
\begin{array}{rcl}
e & ::= & \texttt{extend}\, e\, e \mid \texttt{const}\ v \mid x \\
 & & \mid \texttt{merge}\ e\ e\ e \\
 & & \mid e\ \texttt{fby}\ e \mid e\ \texttt{when}\ e \\
 & & \mid (e, e) \mid e(e) \mid \lambda x.e \\
 & & \mid \texttt{rec}\ x.e \\
 & & \mid \texttt{fst}\ e \mid \texttt{snd}\ e \\
v & ::= & \texttt{true} \mid \texttt{false} \mid i \mid \ldots \\
 & & \mid \texttt{not} \mid \ldots \\
 & & \mid \texttt{+} \mid \ldots
\end{array}
$$

Remarques

- un noyau fonctionnel classique étendu avec des opérateurs temporels
- le langage importe des valeurs scalaires et des opérateurs n-aires d'un langage hôte (C dans le cas de LUSTRE)
- LUSTRE est un sous-ensemble de 1er ordre de ce langage:
  - en LUSTRE, les fonctions définissent un espace de noms différent
  - elles sont définies au niveau global
  - elles ne peuvent être pris en paramêtre ou rendues en résultat
  - pas d'imbrication de définitions
- ajouter un peu de sucre syntaxique pour écrire des définitions de flots mutuellement récursifs et ne pas manipuler explicitement `const` ou `extend`

Sémantique de Kahn

$$
\begin{array}{lcl}
\texttt{const}\ v & = & v.\texttt{const}\ v \\
\texttt{extend}\,(f.fs)\,(x.xs) & = & (fx).\texttt{extend}\,fs\,xs \\
\\
x.xs\ \texttt{fby}\ ys & = & x.ys \\
\\
x.xs\ \texttt{when}\ true.cs & = & x.(xs\ \texttt{when}\ cs) \\
x.xs\ \texttt{when}\ false.cs & = & xs\ \texttt{when}\ cs \\
\\
\texttt{merge}\ true.cs\ x.xs\ ys & = & x.\texttt{merge}\ cs\ xs\ ys \\
\texttt{merge}\ false.cs\ xs\ y.ys & = & y.\texttt{merge}\ cs\ xs\ ys
\end{array}
$$

- opérations continues
- (mais) pas de contrainte de synchronisme
- (mais) pas de contrainte de causalité

Flots paresseux

```
module Streams where

-- lifting constants
constant x = x : (constant x)

-- pointwise application
extend (f:fs) (x:xs) = (f x):(extend fs xs)

-- delays
(x:xs) `fby` y = x:y
pre x y = x : y

-- sampling
(x : xs) `when` (True : cs)  = (x : (xs `when` cs))
(x : xs) `when` (False : cs) = xs `when` cs

merge (True : c)  (x : xs) y  = x : (merge c xs y)
merge (False : c) x (y : ys)  = y : (merge c x ys)
```

Flots infinis synchrones

- compléter l'espace des valeurs avec une valeur absente
- sémantique partielle
- l'horloge d'un flot x est un flot booléen indiquant les instants où x est présent
- se simule encore dans un langage paresseux

type co-inductif:

$$Stream(A) = A \,.\, Stream(A)$$
$$Value(T) = abs + T$$
$$Clstream(T) = Stream(Value(T))$$
$$Clock = Stream(bool)$$

```
clock (abs.x) = false.(clock x)
clock (v.xs) = true.(clock xs)
clock :  Clstream A -> Stream bool
```

Flot constant:

- flot sur une horloge quelconque

  `clconst v (true.cl) = v.(clconst v cl)`

  `clconst v (false.cl) = `*abs*`.(clconst v cl)`

  `clconst:  A -> Clock -> Clstream A`

- horloge de base

  `base = true.base`

- flot sur l'horloge de base

  `bconst v = v.bconst v`

Application point-à-point

- sémantique partielle
- les arguments sont synchrones

```
extend (v.fs) (w.xs) = (v w).(extend fs xs)
extend (abs.fs) (abs.xs) = abs.(extend fs xs)
extend :  Clstream (A -> B) -> Clstream A -> Clstream B

lift1 f (abs.xs) = abs.(lift1 f xs)
lift1 f (v.xs) = (f v).(lift1 f xs)
lift1:  (A -> B) -> Clstream A -> Clstream B

lift2 f (abs.xs) (abs.ys) = abs.(lift2 f xs ys)
lift2 f (v.xs) (w.ys) = (f v w).(lift2 f xs ys)
lift2:  (A->B->C) -> Clstream A -> Clstream B -> Clstream C
```

Opérateurs temporels

(*abs*.xs) fby (*abs*.ys) = *abs*.(xs fby ys)

v.xs fby w.ys = v.fby1 w xs ys

fby1 v (*abs*.xs) (*abs*.ys) = *abs*.fby1 v xs ys

fby1 v (w.xs) (s.ys) = v.(fby1 s xs ys)

fby: Clstream A -> Clstream A -> Clstream A

Echantillonnage

(*abs*.xs) when (*abs*.cs) = *abs*.(xs when cs)

(v.xs) when (true.cs) = v.(xs when cs)

(v.xs) when (false.cs) = *abs*.(xs when cs)

when : Clstream A -> Clstream bool -> Clstream A

Echantillonnage et horloge

- `x` et `c` doivent avoir la même horloge `cl`
- l'horloge de `x when c` sera notée `cl on c`

```
(true.cl) on (true.cs) = true.(cl on cs)
(true.cl) on (false.cs) = false.(cl on cs)
(false.cl) on (abs.cs) = false.(cl on cs)
on :  Clock -> Clstream bool -> Clock
```

Sur-échantillonnage

```
merge (abs.cs) (abs.xs) (abs.ys) = abs.(merge cs xs ys)
merge (true.cs) (v.xs) (abs.ys) = v.(merge cs xs ys)
merge (false.cs) (abs.xs) (v.ys) = v.(merge cs xs ys)
merge :  Clstream bool -> Clstream A
                       -> Clstream A -> Clstream A
```

Liens entre flots et flots synchrones

`clstream2stream (`*abs*`.cl) = clstream2stream cl`

`clstream2stream (v.cl) = v.clstream2stream cl`

`clstream-to-stream : Clstream A -> Stream A`

Conclusion

- la sémantique des flots synchrones est partielle
- les cas "asynchrones" donnent lieu à des échecs à l'exécution
- le calcul d'horloge, à la manière du typage, permet de rejeter ces programmes: **"well clocks programs cannot go wrong"**
- causalité: éliminer les récursions instantanées (les équations $x = e$ telles que ϵ soit la plus petite solution)
- on peut aller plus loin et utiliser Coq comme langage dans lequel plonger la sémantique [Boulme & Hamon, LPAR 2001]. Les contraintes d'horloge sont encodées dans le type (dépendant) des flots synchrones.

Définition totale: cas synchrones + cas asynchrones

```
const v = v.(const v)

extend (v.fs) (w.xs) = (v w).(extend fs xs)
extend (abs.fs) xs = abs.(extend fs xs)
extend fs (abs.xs) = abs.(extend fs xs)

(abs.xs) fby ys = abs.(xs fby ys)
(v.xs) fby (w.ys) = v.((w.ys) fby ys)
```

```
(abs.xs) when cs = abs.(xs when cs)
xs when (abs.cs) = abs.(xs when cs)
(v.xs) when (true.cs) = v.(xs when cs)
(v.xs) when (false.cs) = abs.(xs when cs)

merge (true.cs) (v.xs) ys = v.(merge cs xs ys)
merge (false.cs) xs (v.ys) = v.(merge cs xs ys)
merge (abs.cs) xs ys = abs.(merge cs xs ys)
merge cs (abs.xs) ys = abs.(merge cs xs ys)
merge cs xs (abs.ys) = abs.(merge cs xs ys)
```

References

[1] E.A. Ashcroft and W.W. Wadge. Lucid, a non procedural language with iteration. *Communications of the ACM*, 20(7):519–526, 1977.

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

[3] Sylvain Boulmé and Grégoire Hamon. Certifying Synchrony for Free. In *International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR)*, volume 2250, La Havana, Cuba, December 2001. Lecture Notes in Artificial Intelligence, Springer-Verlag. Short version of *A clocked denotational semantics for Lucid-Synchrone in Coq*, available as a Technical Report (LIP6), at `www.lri.fr/~pouzet`.

[4] P. Caspi, N. Halbwachs, D. Pilaud, and J. Plaice. Lustre: a declarative language for programming synchronous systems. In *14th ACM Symposium on Principles of Programming Languages*. ACM, 1987.

[5] Paul Caspi and Marc Pouzet. Synchronous Kahn Networks. In *ACM SIGPLAN International Conference on Functional Programming*, Philadelphia, Pensylvania, May 1996.

[6] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.

[7] Gilles Kahn. The semantics of a simple language for parallel programming. In *IFIP 74 Congress*. North Holland, Amsterdam, 1974.

[8] Gilles Kahn and David B. MacQueen. Coroutines and networks of parallel processes. In *IFIP Congress*, pages 993–998, 1977.

[9] Marc Pouzet. *Lucid Synchrone: un langage synchrone d'ordre supérieur*. Paris, France, 14 novembre 2002. Habilitation à diriger les recherches.