Laboratoire d'Informatique Signaux et Systèmes (I3S)
Université de Nice-Sophia-Antipolis
C.N.R.S - U.R.A. 1376
Thème Architectures Logicielles et Matérielles
Équipe Lambda-X
41 Bd. Napoléon III – 06041 NICE Cédex

RAPPORT DE RECHERCHE 95-34

Implementation des λ-matrices à l'aide du λ-calcul

Guilhem de Wailly et Fernand Boéri

Résumé: Dans ce rapport interne, il est montré comment réaliser un interprète de λ-matrices à l'aide de λ-expressions. Les λ-matrices sont un language abstrait (sémantique) orienté *Data Flow Synchrone et Fonctionnel.* Il permet de modéliser les applications de traitements du signal en vue de leur réalisation sur une architecture matérielle parallèle.

Ce rapport a pour seul objectif de montrer que cette réalisation est possible, et donc que les λ-matrices sont fonctionnelles.

Table des matières

Table des matières 1

1 Introduction 2

2 Conventions syntaxiques 2

3 Opérateurs standards 2

4 Opérateurs complémentaires 3

5 Type des acteurs 6

6 Opérateur 7

7 Structure des acteurs 7

8 Atomes 9

9 Objet 10

10 Ensemble d'appartenance des acteurs 12

11 « Applicateurs » 13

12 Comparaison 14

13 Opérateurs arithmétique 15

14 Opérateur de base 15

15 Environnement 16
- 15.1 Chaînage des environnements 16
- 15.2 Valeur Associée 17
- 15.3 Environnement Associé 17
- 15.4 Extension d'environnement 19
- 15.5 Comparaison d'environnement 19

16 Opérateurs dynamiques 19
- 16.1 Réduction 19
- 16.2 Évaluation 19
- 16.3 Régénération 21
- 16.4 Résolution 21
- 16.5 Fermeture 22
- 16.6 Calculabilité 22
- 16.7 Norme 23
- 16.8 Stabilité 24

Bibliographie 25

1 Introduction

Ce rapport de recherche montre une traduction possible du formalisme des λ-matrices[dWB96b, dWB96c, dWB96a, dWB96d] en λ-calcul[Chu51, Joh85, Kri90, Rev88], ce qui montre que le formalisme utilisé est entièrement fonctionnel. Cette traduction peut servir de base à l'élaboration d'un interprète de λ-matrices. Par exemple, convertir les λ-expressions suivantes dans le langage SCHEME[ASS87, CR90, Jaf90, Gal95] ne pose aucune difficulté.

De plus, l'utilisation des λ-expression pour décrire ce formalisme permet de lever toutes ambiguité. Elles forment une sémantique opérationnelle de λ-matrices.

2 Conventions syntaxiques

Le λ-calcul offre une syntaxe simple et très expressive. Cependant cette syntaxe peut être ardue à lire. Un certain nombre de conventions syntaxiques destinées à faciliter la lectures des λ-expressions sont donc utilisées.

Ces conventions portent en premier lieu sur les noms utilisés :

Syntaxe	Description
nom	définition de base
NOM	constante
nom?	prédicat retournant **true** ou **false**
nom:	constructeur de l'objet nom
nom-	sélecteur retournant le champs nom
nom'	primitive de la fonction nom

En second lieu, les pseudo-opérateurs **if**, **then** et **else** avec des indentations adéquate permettent de faciliter la lecture des expressions. Ces opérateurs sont définis dans la sections suivante. Parfois l'expression utilisant ces opérateurs sera réduite, mais le plus souvent cette forme qui n'est pas en forme normale sera conservée.

3 Opérateurs standards

Les constantes et opérateurs suivants, dont la définition peut être trouvée dans [Rev88]. Ils sont les opérateurs primitifs utilisées abondamment par la suite :

```
true  = λ x.λ y.x
false = λ x.λ y.y
not   = λ x.((x) false) true
and   = λ x.λ y.((x) y) false
or    = λ x.λ y.((x) true) y
xor   = λ x.λ y.((x)(not) y) y
0     = λ f.λ x.x
1     = λ f.λ x.(f) x
2     = λ f.λ x.(f) (f) x
3     = λ f.λ x.(f) (f) (f) x
...
a     = 0
```

```
b     = 1
c     = 3
...
succ  = λ n.λ f.λ x.(f)((n) f) x
pred  = λ n.(((n) λ p.λ z.
        ((z) (succ) (p) true)(p)true) λ z.((z) 0) 0) false
zero? = λ n.((n) (true) false) true
Y     = λ y.(λ x.(y) (x) x) λ x.(y) (x) x
cons  = λ a.λ b.λ z.((z) a) b
head  = λ l.(l) true
tail  = λ l.(l) false
nil   = false
nil?  = λ v.(not) v
```

Pour simplifier les écritures, Les formes syntaxiques suivantes sont définies :

```
if    = λ cond.λ alors.λ sinon.
            ((cond) alors) sinon
then  = λ alors.alors
else  = λ sinon.sinon
elsif = if
```

4 Opérateurs complémentaires

Les opérateurs complémentaires repsent sur les opérateur standard définis plus haut. Ils concernent l'arithmétique entière, la logique des prédicats, les opérateurs de comparaison et les listes.

Addition + : Effectue la somme arithmétique de deux nombres de CHURCH.

```
+ = λ a.λ b.(a) (succ) b
```

Soustraction - : Calcule la différence de deux nombres de CHURCH. Si le premier nombre est plus petit que le second, l'opérateur retourne zéro.

```
- = λ a.λ b. ((((<?) a) b) 0) (a) (pred) b
```

Multiplication * : Calcule le produit de deux nombres.

```
* = λ a.λ b.
  (((Y) λ rec. λ a. λ t.
    (((if) (zero?) a)
      (then) t)
      (else)
        ((rec) (pred) a) ((+) t) b) a) 0
```

Division / : Calcule la division de deux nombres. Si le dividende est inférieure au diviseur, alors le résultat est zéro.

```
/ = λ a.λ b.
  (((Y) λ rec. λ a. λ t.
    (((if) ((<?) a) b)
      (then) t)
      (else)
        ((rec) ((-) a) b) (succ) t) a) 0
```

Supérieur ou égal ≥? : Retourne **false** si le premier nombre est inférieure au second, et **true** dans le cas contraire.

```
≥? = λ a.λ b.
            (((if) (zero?) ((-) ((max) a) b) a)
              (then) true)
              (else) false
    = λ a.λ b.(zero?) ((-) ((max) a) b) a
    = λ a.λ b.(zero?) ((-) ((((≥?) a) b) a) b) a
    = λ a.λ b.(Y) λ rec.
              (zero?) ((-) ((((rec) a) b) a) b) a
```

Supérieur strict >? : Retourne **false** si le premier nombre est inférieure ou égale au second, et **true** dans le cas contraire.

```
>? = λ a. λ b.
       (zero?) ((-)(succ) ((max) a) b) a)
```

Égal =? : Retourne **false** si les deux nombres sont différents, et **true** dans le cas contraire.

```
=? = λ a. λ b.
  (((if) ((and) ((≥?) a) b) ((≥?) b) a)
    (then) true)
    (else) false
=? = λ a. λ b.
  ((((and) ((≥?) a) b)
     ((≥?) b) a) true) false
```

Différent ≠? : Retourne la valeur **false** si les deux nombres sont égaux, et **true** dans le cas contraire.

```
≠? = λ a.λ b.(not) ((=?) a) b
```

Inférieur strict <? : Retourne **false** si le premier nombre est supérieur ou égale au second, et **true** dans le cas contraire.

```
<? = λ a.λ b.(not) ((≥?) a) b
```

Inférieur ou égale ≤? : Retourne **false** si le premier nombre est supérieur au second, et **true** dans le cas contraire.

```
≤? = λ a.λ b.(not) ((>?) a) b
```

Maximum max : Retourne le maximum de deux nombres.

```
max = λ a.λ b. (((if) ((≥?) a) b) (then) a) (else) b
    = λ a.λ b.((((≥?) a) b) a) b
```

Minimum min : Retourne le minimum de deux nombres.

```
min = λ a.λ b.
         (if) ((=?) ((max) a) b) a)
           (then) b)
           (else) a
    = λ a.λ b.
         ((((=?) ((max) a) b) a) b) a
```

Parcourt d'une liste map : Construit une liste avec le résultat de l'application d'une fonction à tous les éléments d'une liste.

```
map = λ fct.λ liste.
           (((if) (nil?) liste)
             (then) nil)
             (else) ((cons)
                        (fct)(head) liste)
                        ((map) fct) (tail) liste
    = (Y) λ rec. λ fct. λ liste.
           (((nil?) liste) nil)
             ((cons)
               (fct) (head) liste)
               ((rec) fct) (tail) liste
```

Réduction d'une liste reduce : Réduit une liste en appliquant une fonction à un accumulateur et tous ses éléments. Le résultat final est l'accumulateur.

```
reduce = λ accu. λ fct. λ liste.
     (((if) (nil?) liste)
       (then) accu)
       (else)
```

```
          (((reduce)
            ((fct) accu)(head) liste)
            fct)
            (tail) liste
  = λ accu. λ fct. λ liste.
      (((nil?) liste)
        accu)
        (((reduce)
          ((fct) accu)(head) liste)
          fct)
          (tail) liste
```

Concaténation de liste append : Concatène deux liste pour n'en former qu'une seule.

```
append = λ l1. λ l2.
            (((if) (nil?) l1)
              (then) l2)
              (else)
                ((cons)
                  (head) l1)
                  ((append) (tail) l1) l2
        = (Y) λ rec. λ l1. λ l2.
            (((nil?) l1) l2)
              ((cons) (head) l1)
                ((append) (tail) l1) l2
```

Longueur d'une liste length : Retourne le nombre d'éléments d'une liste.

```
length = λ liste.
  (((if) (nil?) liste)
    (then) 0)
    (else) (succ)(length)(tail) liste
        = λ liste.
  (((reduce) 0) +) ((map) λ z.1) liste
```

5 Type des acteurs

Voici maintenant les types des acteurs comme des constantes numériques :

```
ALTERNATIVE = 1
APPLICATION = 2
DÉFINITION  = 3
DONNÉE      = 4
ENTIER      = 5
EXTRACTION  = 6
```

```
FLOT        = 7
INDÉFINI    = 8
OPÉRATEUR   = 9
SURDÉFINI   = 10
SYMBOLE     = 11
VECTEUR     = 12
```

6 Opérateur

Un opérateur est une liste contenant dans sa tête la λ-expression de la fonction et dans sa queue son arité. Ainsi, il sera possible d'effectuer un contrôle sur le nombre d'arguments au moment de l'appel. On définit donc un constructeur d'opérateur, et deux sélecteurs, l'un pour la fonction, l'autre pour l'arité.

```
opérateur: = λ fonction. λ arité.
  ((cons) fonction) arité
fonction-  = λ opérateur.
  (head) opérateur
arité-     = λ opérateur.
  (tail) opérateur
```

7 Structure des acteurs

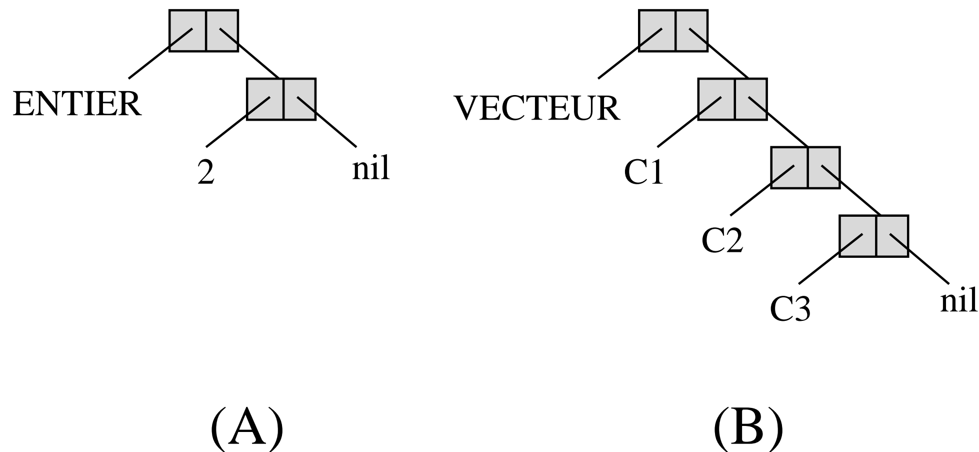

FIG. 1 - Structure des acteurs : (A) est l'atome entier 2, (B) est un vecteur à trois composantes.

Un acteur a une structure de liste 1, avec en tête son type, et en queue, la liste de ses composantes. Pour certains acteurs, cette listes a une longueur constante, comme pour les définitions, et pour d'autre, une longueur variable, comme pour les vecteurs.

Construction d'un acteur typé vide : Construit un acteur d'un type donné sans composantes

```
acteur: = λ type.((cons) type) nil
```

Obtention du type d'un acteur : sélectionne le type d'un acteur

```
type- = λ acteur.(head) acteur
```

Obtention de la valeur d'un acteur : sélectionne la liste des composantes d'un acteur

```
valeur- = λ acteur.(tail) acteur
```

Savoir si un acteur n'a aucune composante : indique si un acteur a des composantes (**true**) ou non (**false**)

```
vide? = λ acteur.(nil?) (valeur-) acteur
```

Ajoute une composante à un acteur : construit un acteur de même type que l'acteur donné dont la liste des composante s'est vue rajouter à sa tête une nouvelle composante

```
ajoute: = λ acteur.λ composante.
  ((cons) (type-) acteur)
    ((cons) composante) (valeur-) acteur
```

Ajouter une liste de composantes à un acteur : construit un acteur de même type que l'acteur donné dont la liste des composantes s'est vue rajouter à sa tête une liste de nouvelles composantes

```
ajoute-liste: = λ acteur.λ composantes.
  ((cons) (type-) acteur)
    ((append) composantes) (valeur-) acteur
```

Savoir si un acteur est d'un certain type : permet de comparer le type d'un acteur à un type donné

```
type? = λ acteur.λ type.
  ((=?) (type-) acteur) type
```

Savoir si un acteur est un atome : permet de savoir si un acteur est un atome (**true**) ou non (**false**)

```
atome? = λ acteur.
  (((if) ((or) ((type?) acteur) ENTIER)
         ((or) ((type?) acteur) DONNÉE)
                ((type?) acteur) SYMBOLE)
    (then) true)
    (else) false
```

8 Atomes

Pour chaque atome, on définit un constructeur, un sélecteur sur ses composantes et un prédicat pour l'identifier. La structure des atomes respecte celle des objet, que l'on peut voir dans la figure 1.

Les atomes sont des acteurs dont les composantes ne sont pas des acteurs. Leur valeurs sont donc des λ-expressions ne répondant pas la la structure de liste des acteurs. Nous avons :

Entier : constructeur, sélecteur et prédicat des entiers

```
entier: = λ valeur.((ajoute:) (acteur:) ENTIER) valeur
entier- = λ entier.(head) (valeur-) entier
entier? = λ entier.((type?) entier) ENTIER
```

Indéfini : constructeur et prédicat du marqueur indéfini

```
indéfini: = ((ajoute:) (acteur:) INDÉFINI) 0
indéfini? = λ acteur.((type?) acteur) INDÉFINI
```

Donnée : constructeur sélecteur et prédicat des données

```
donnée: = λ valeur.((ajoute:) (acteur:) DONNÉE) valeur
donnée- = λ donnée.(head) (valeur-) donnée
donnée? = λ donnée.((type?) donnée) DONNÉE
```

Surdéfini : constructeur et prédicat du marqueur surdéfini

```
surdéfini: = ((ajoute:) (acteur:) SURDÉFINI) 0
surdéfini? = λ acteur.((type?) acteur) SURDÉFINI
```

Symbole : constructeur, sélecteur et prédicat des symboles

```
symbole: = λ valeur.((ajoute:) (acteur:) SYMBOLE) valeur
symbole- = λ symbole.(head) (valeur-) symbole
symbole? = λ symbole.((type?) symbole) SYMBOLE
```

les symboles doivent être conçus comme une liste de lettre formant un acteur de type SYMBOLE. Il faut remarquer qu'il ne sera jamais nécessaire de construire explicitement un symbole. Ils seront donc manipulés comme des listes construites à l'avance.

9 Objet

Pour chaque type d'objet —alternative, application, définition, extraction, flot et vecteur—, on définit un constructeur, un prédicat et des sélecteurs. Nous avons :

Alternative :

```
alternative: = λ condition. λ alors. λ sinon.
  ((ajoute:)
    ((ajoute:)
      ((ajoute:)
        (acteur:) ALTERNATIVE)
        sinon)
      alors)
    condition
alternative? = λ acteur. ((type?) acteur) ALTERNATIVE
condition-   = λ alternative.
  (((if) (alternative?) alternative)
    (then) (head) (valeur-) alternative)
    (else) indéfini:
alors-       = λ alternative.
  (((if) (alternative?) alternative)
    (then) (head) (tail) (valeur-) alternative)
    (else) indéfini:
sinon-       = λ alternative.
  (((if) (alternative?) alternative)
    (then) (head) (tail) (tail) (valeur-) alternative)
    (else) indéfini:
```

Application :

```
application: = λ opérateur.
  ((ajoute:)
    (acteur:) APPLICATION)
    opérateur
application? = λ acteur. ((type?) acteur) APPLICATION
opérateur-   = λ application.
  (((if) (application?) application)
    (then) (head) (valeur-) application)
    (else) indéfini:
arguments-   = λ application.
  (((if) (application?) application)
    (then) (valeur-) application)
```

```
    (else) indéfini:
```

Définition :

```
définition: = λ nom. λ valeur.
  ((ajoute:)
    ((ajoute:)
      (acteur:) DÉFINITION)
      valeur)
    nom
définition? = λ acteur. ((type?) acteur) DÉFINITION
nom-        = λ définition.
  (((if) (définition?) définition)
    (then) (head) (valeur-) définition)
    (else) indéfini:
synonyme-   = λ définition.
  (((if) (définition?) définition)
    (then) (head) (tail) (valeur-) définition)
    (else) indéfini:
```

Extraction :

```
extraction: = λ module. λ index.
  ((ajoute:)
    ((ajoute:)
      (acteur:) EXTRACTION)
      index)
    module
extraction? = λ acteur. ((type?) acteur) EXTRACTION
module-     = λ extraction.
  (((if) (extraction?) extraction)
    (then) (head) (valeur-) extraction)
    (else) indéfini:
index-      = λ extraction.
  (((if) (extraction?) extraction)
    (then) (head) (tail) (valeur-) extraction)
    (else) indéfini:
```

Flot :

```
flot:    = λ état. λ contrat.
  ((ajoute:)
    ((ajoute:)
      (acteur:) FLOT)
      contrat)
    état
```

```
flot?    = λ acteur. ((type?) acteur) FLOT
état-    = λ flot.
  (((if) (flot?) flot)
    (then) (head) (valeur-) flot)
    (else) indéfini:
contrat- = λ flot.
  (((if) (flot?) flot)
    (then) (head) (tail) (valeur-) flot)
    (else) indéfini:
```

Vecteur :

```
vecteur: = λ composante. λ vecteur.
  (((if)(nil?) vecteur)
    (then)((ajoute:)
             (acteur:) VECTEUR)
             composante)
    (else)
      (((if)(vecteur?) vecteur)
        (then) ((ajoute:) vecteur) composante)
        (else) nil
vecteur? = λ acteur. ((type?) acteur) VECTEUR
```

10 Ensemble d'appartenance des acteurs

Les acteurs peuvent appartenir à deux sous-ensembles : les acteurs figés, constitué des vecteurs et des atomes sans les symboles, et les acteurs neutres constitués des acteurs figés et des applications.

Nous définissons donc deux prédicats qui permettent de connaître le sous-ensemble d'appartenance. Ce sont des fonction récursive parcourant la totalité des composantes et des sous-composantes de l'acteur donnée en testant leur l'ensemble d'appartenance. Le parcourt sur les compsantes d'un objet se fait à l'aide de l'utilisation conjoint des puissants opérateur **map** et **reduce**.

```
figé? = λ acteur.
  (((if) (atome?) acteur)
    (then)
      (((if) (symbole?) acteur)
        (then) false)
        (else) true)
  (((elsif) (vecteur?) acteur)
    (then)
      (((reduce) true) and)
        ((map) λ composante.
          (figé?) composante)
          (valeur-) acteur)
```

```
    (else) false
```

11 « Applicateurs »

Deux opérateurs qui permettent d'effectuer des opérations sur les acteurs en vérifiant leur type sont décrits ici. Ces opérateurs reconstruisent avec la valeur de retour de cette application un acteur de même type que le type des acteurs initiaux. On distingue un « applicateur » monadique et un applicateur « dyadique ». Voici leurs expressions :

```
monadique = λ acteur.λ type.λ fonction.
  (((if) ((type?) acteur) type)
    (then) ((acteur:) type) (fonction) (valeur-) acteur)
    (else) indéfini:

dyadique = λ a.λ b.λ type.λ fonction.
  (((if) ((and)
          ((type?) a) type)
          ((type?) b) type)
    (then)
      ((acteur:) type) ((fonction) (valeur-) a)
                                   (valeur-) b)
    (else) indéfini:
```

Ici, on définit une fonction appliquant un opérateur — constitué d'une fonction et d'une arité — à une liste d'arguments.

```
applique'= λ fonction. λ arguments.
  (((if) (nil?) arguments)
    (then) opérateur)
    (else)
      ((applique')
        (fonction)(head) arguments)
        (tail) arguments
applique = λ opérateur. λ arguments.
  (((if) ((=?)
           (length) arguments)
           (arité-) opérateur)
    (then)
      ((applique')
        (fonction-) opérateur)
        arguments)
    (else)
      indéfini:
```

12 Comparaison

Pour tous les atomes, il existe des fonctions permettant de les comparer. Nous définissons ici à titre d'exemple, le comparateur des entiers :

```
entier=?  =  λ a. λ b. ((((dyadique) ENTIER) a) b) =?
```

Les autres opérateurs de comparaison des atomes se déduisent facilement.

Le problème est légèrement plus complexe en ce qui concerne la comparaison des symboles, car il faut effectuer une comparaison sur chacun des caractères les composants. Il est donc composé d'un opérateur frontale appelant un opérateur récursif :

```
symbole=?'  = λ a. λ b.
  (((if) ((or) (nil?) a) (nil?) b)
    (then)
      (((if) ((and) (nil?) a) (nil?) b)
        (then) true)
        (else) false)
    (else)
      (((if) ((=?) (head a) (head) b)
        (then) ((symbole=?)(tail) a)(tail) b)
        (else) false
symbole=?   = λ a. λ b.
  (((if) ((and) (symbole?) a) (symbole?) b)
    (then) ((symbole=?') (tail) a) (tail) b)
    (else) false
```

Maintenant, on définit un comparateur d'atome qui permet de savoir si deux atomes sont identiques. Deux atomes sont identiques s'ils ont untype identique, et si leur valeurs sont elles-même identiques. Pour les données, cette fonction n'est pas définie, car elles sont dépendantes de l'utilisateur final. Il vient :

```
atome=? = λ atome1. λ atome2.
  (((if)
    ((and) (atome?) atome1)
    ((and) (atome?) atome2)
            ((type=?) (type-) atome1) atome2)
    (then)
      (((if) (entier?) atome1)
        (then) ((entier=?) atome1) atome2)
      (((elsif) (donnée?) atome1)
        (then) ((donnée=?) atome1= atome2)
      (((elsif) (symbole?) atome1)
        (then) ((symbole=?) atome1) atome2)
      (else) true)
    (else) false)
```

Ici, on trouve un comparateur d'objets qui permet de savoir si deux objets sont identiques. Deux objets sont identiques s'ils ont même type, et si leur composantes sont

identiques deux à deux. La fonction fait appel à une fonction de comparaison de la liste des composantes, appelée par une fonction frontale.

```
composantes=? = (Y) λ test. λ a. λ b.
            (((if) ((objet=?) (head) a) (head) b)
              (then)
                ((test) (tail) a) (tail) b)
              (else) false


objet=? = λ objet1. λ objet2.
  (((if) ((type=?) (type-) objet1) objet2)
    (then)
      (((if) (atome?) objet1)
        (then) ((atome=?) objet1) objet2)
        (else)
          ((composantes=?) (valeur-) objet1) (valeur-) objet2)
    (else) false
```

13 Opérateurs arithmétique

À titre d'exemple, on donne les opérateurs arithmétiques pour les opérations sur les entiers. Ces fonctions pourraient faire partie de l'algèbre de l'utilisateur final.

```
Add = ((opérateur:)
  λ a. λ b. (((dyadique) ENTIER) a) b) +)
  2
Sub = ((opérateur:)
  λ a. λ b. (((dyadique) ENTIER) a) b) -
  2
Mul = ((opérateur:)
  λ a. λ b. (((dyadique) ENTIER) a) b) *
   2
Div = ((opérateur:)
  λ a. λ b. (((dyadique) ENTIER) a) b) /
  2
```

14 Opérateur de base

Index : permet de sélectionner une des composantes d'un objet. Il est constitué d'une partie frontale et d'une partie récursive.

```
index'  = λ n. λ liste.
  (((if) (nil?) liste)
    (then) indéfini:)
    (else)
      (((if) (zero?) n)
```

```
            (then) (head) liste)
            (else)
              ((index') (pred) n) (tail) liste
    index   = λ n. λ acteur.
      (((if) ((or) (not) (entier?) n) (zero?) n)
        (then) indéfini:)
      (((elsif) (atome?) acteur)
        (then) indéfini:)
      (else)
         ((index') (pred) n) (valeur-) acteur
```

Type : voir definition du type.

Dimension : la définition de la dimension d'un acteur repose sur un opérateur frontal et un opérateur récursif :

```
    dimension' =  λ liste. λ n.
      (((if) (nil?) liste)
        (then) n)
        (else)
          ((dimension')
            (tail) liste) (succ) n
    dimension  =  λ acteur.
      (((if) (atome?) acteur)
        (then) 1)
        (else) ((dimension') (valeur-) acteur) 1
```

15 Environnement

Les environnements ne sont pas des objets de l'utilisateurs car il ne peut les manupuler directement. Pour simplifier, ils sont des listes dont les composantes sont des vecteurs.

15.1 Chaînage des environnements

Cet opérateur permet de chaîner un vecteur à un environnement. Il contrôle qu'il s'agit bien d'un vecteur.

```
    environnement: = λ vecteur. λ environnement.
       (((if) (not) (vecteur?) vecteur)
         (then) nil)
         (else) ((cons) vecteur) environnement
```

15.2 Valeur Associée

L'opérateur de recherche de la valeur associée à un symbole dans un environnement est constituée de trois parties non réduites en forme normale pour simplifier les expressions.

recherche dans la liste des composantes d'un vecteur :

```
cherche'' = λ symbole. λ composantes.
  (((if) (nil?) composantes)
    (then) nil)
    (else)
      (((if) (définition?) (head) composantes)
        (then)
          (((if) ((symbole=?)
                   symbole)
                   (nom-) (head) composantes)
            (then) (valeur-) (head) composantes)
            (else) ((cherche'') symbole) (tail) composantes)
        (else) ((cherche'') symbole) (tail) composantes
```

recherche dans la liste des composantes d'un environnement :

```
cherche' = λ symbole. λ environnement.
  (((if) (nil?) environnement)
    (then) indéfini:)
    (else)
      (λ locale.
        (((if) (not)(nil?) locale)
          (then) locale)
          (else) ((cherche') symbole) (tail) environnement)
      ) ((cherche'') symbole) (valeur-)(head) environnement
```

recherche du symbole :

```
cherche = λ symbole. λ environnement.
  (((if) (nil?) environnement)
    (then) indéfini:)
    (else) ((cherche') symbole) environnement
```

15.3 Environnement Associé

Cet opérateur retourne l'environnement de définition d'un symbole. Il est lui aussi composé de trois fonctions dont la structure est proche de celle de l'opérateur précédent.

recherche dans la liste des composantes d'un vecteur :

```
associé'' = λ symbole. λ composantes. λ environnement.
  (((if) (nil?) composantes)
    (then) nil)
    (else)
      (((if) (définition?) (head) composantes)
        (then)
          (((if) ((symbole=?)
                    symbole)
                    (nom-) (head) composantes)
            (then) environnement)
            (else) (((associé'')
                      symbole)
                      (tail) composantes)
                      environnement)
        (else) (((associé'')
                  symbole)
                  (tail) composantes)
                  environnement
```

recherche dans la liste des composantes d'un environnement :

```
associé' = λ symbole. λ environnement.
  (((if) (nil?) environnement)
    (then) nil)
    (else)
      (λ locale.
        (((if) (not)(nil?) locale)
          (then) locale)
          (else) ((associé') symbole) (tail) environnement)
      ) (((associe'')
          symbole)
          (valeur-)(head) environnement)
          environnement
```

recherche de l'environnement associé :

```
associé = λ symbole. λ environnement.
  (((if) (nil?) environnement)
    (then) nil)
    (else) ((associé') symbole) environnement
```

15.4 Extension d'environnement

Cette fonction permet d'étendre un environnement avec un acteur placé en tête de la liste des composantes du premier vecteur de la liste des vecteurs.

```
étendre: = λ symbole. λ valeur. λ environnement.
  (((if) (nil?) environnement)
    (then) ((environnement:)
                 ((vecteur:) ((definition:) symbole) valeur)
                             nil)
                nil)
    (else) ((environnement:)
              ((vecteur:) ((definition:) symbole) valeur)
                          (head) environnement)
              (tail) environnement
```

15.5 Comparaison d'environnement

Ce prédicat permet de comparer deux environnements donnés. Ils sont identiques si chacun des vecteurs qui les composent sont identiques deux à deux. Cette fonction est en fait identique à la fonction comparant la liste des composantes d'un objet. Nous avons :

```
environnement=? = λ a. λ b.
  ((composantes=?) a) b
```

16 Opérateurs dynamiques

Les opérateurs dynamiques sont des opérateurs à deux paramètre : un acteur et un environnement. L'environnement représente l'environnement courant de l'acteur en question. Ce sont des fonctions itératives, bien qu'elles apparaissent comme récursives.

16.1 Réduction

La réduction est la fonction qui applique un opérateur à la liste des arguments. Elle est synonyme de la fonction d'application que nous avons déjà décrit.

```
réduit: = λ application.
  (applique)
    (opérateur-) application)
    (argument-) application
```

16.2 Évaluation

L'évaluation est la fonction qui donne une valeur exploitable à tous les constituants d'un système. Dans le cas d'un symbole, elle poursuit son processus sur la valeur associée à ce symbole dans son environnement de définition. Dans le cas des vecteurs et des objets,

elle construit un objet de même type dont les composantes sont le résultat de l'évaluation de composantes de l'objet d'origine.

```
évalue: = λ acteur. λ environnement.
  (((if) (alternative?) acteur)
    (then)
      (((if) (zero?)
                ((évalue:)
                  (condition-) alternative)
                  environnement)
        (then)
          ((évalue:)
            (sinon-) alternative)
            environnement)
        (else)
          ((évalue:)
            (alors-) alternative)
            environnement)
  (((elsif) (application?) acteur)
    (then)
      (reduit:)
        (ajoute-liste:)
           (application:) (operateur-) acteur
           ((map) λ acteur.
                    ((evalue:) acteur) environnement)
                  (arguments-) acteur)
  (((elsif) (atome?) acteur)
    (then)
      (((if) (symbole?) acteur)
        (then)
          ((évalue:)
            ((cherche) acteur) environnement)
            ((associe) acteur) environnement)
        (else) acteur)
  (((elsif) (définition?) acteur)
    (then) ((évalue:) (synonyme-) acteur) environnement)
  (((elsif) (extraction?) acteur)
    (then) (λ module.
      (((if) (vecteur.) module)
        (then)
          ((index)
            ((évalue:) (index-) acteur) environnement) module)
        (else) indéfini:
            ) ((évalue:) (module-) acteur) environnement)
  (((elsif) (flot?) acteur)
    (then) ((évalue:) (état-) acteur) environnement)
  (((elsif) (vecteur?) acteur)
    (then)
```

```
    ((ajoute-liste)
      (acteur:) VECTEUR)
      ((map)
        λ composante.
          ((évalue:)
            composante)
            ((environnement:) acteur) environnement)
        (valeur-) acteur)
(else) indéfini:
```

16.3 Régénération

La fonction de régénération fait évoluer le système en remplaçant la valeur de l'état des flot par l'évaluation de leur contrat. C'est cette évaluation qui produit l'effet de retard attendu.

```
régénère: = λ acteur. λ environnement.
  (((if) (atome?) acteur)
    (then) acteur)
  (((elsif) (flot?) acteur)
    (then)
      ((flot:)
        ((évalue:)   (contrat-) acteur) environnement)
        ((régénère:) (contrat-) acteur) environnement)
  (((elsif) (vecteur?) acteur)
    (then)
      ((ajoute-liste)
        (acteur:) VECTEUR)
        ((map)
          λ composante.
            ((évalue:)
              composante)
              ((environnement:) acteur) environnement)
          (valeur-) acteur)
  (else)
      ((ajoute-liste)
        (acteur:) VECTEUR)
        ((map)
          λ composante.
            ((évalue:)
              composante)
              environnement)
          (valeur-) acteur
```

16.4 Résolution

L'opérateur de résolution est itérative. Il peut se poursuivre indéfiniment dans le temps, car se condition d'arrêt est une valeur située à l'intérieur du système à résoudre qui doit être la marqueur indéfini pour stopper le processus.

```
résout: = λ système. λ terminaison. λ environnement.
  (((if) (zero?) ((evalue: ((extraction:) système) terminaison)
                              environnement)
    (then)
      (((résout) ((regénère) système) environnement)
                 terminaison
                 environnement)
    (else)
      système
```

16.5 Fermeture

Le critère de la fermeture vérifie qu'à chaque utilisation d'un symbole correspond dans son environnement une définition.

```
fermé? = λ acteur. λ environnement.
  (((if) (atome?) acteur)
    (then)
      (((if) (symbole?) acteur)
        (then)
          (((if) (indéfini?) ((cherche) acteur) environnement)
            (then) false)
            (else) true)
        (then) true)
  (((elsif) (définition?) acteur)
    (then) ((fermé?) (synonyme-) acteur) environnement)
  (((elsif) (vecteur?) acteur)
    (then)
      (((reduce) 1) *)
        ((map) λ composante.
          ((fermé?)
            composante)
            ((environnement:) acteur) environnement)
          (valeur-) acteur)
  (else)
    (((reduce) true) and)
      ((map) λ composante.
        ((fermé?) composante) environnement)
        (valeur-) acteur
```

16.6 Calculabilité

La calculabilité permet de savoir si un acteur est calculable. S'il l'est, son évaluation se termine. S'il ne l'est pas, c'est qu'il contient une équation de point-fixe.

```
calculable? = λ acteur. λ environnement.
  (((if) (definition?) acteur)
    (then) ((calculable) (synonyme-) acteur)
                              environnement)
  (((if) (symbole?) acteur)
    (λ valeur.
       (((if) (indéfini?) valeur)
         (then) true)
       (((elsif) (surdéfini?) valeur)
         (then) false)
       (else)
         ((calculable?) valeur
                        (((étend) acteur)
                                  indefini:)
                                  environnement
     ) ((cherche) acteur) environnement)
  (((if) (atome?) acteur)
    (then) true)
  (((elsif) (vecteur?) acteur)
    (then)
      (((reduce) true) and)
        ((map) λ composante.
          ((calculable?)
            composante)
            ((environnement:) acteur) environnement)
          (valeur-) acteur)
  (else)
    (((reduce) true) and)
      ((map) λ composante.
        ((calculable?)
          composante)
          environnement)
        (valeur-) acteur
```

16.7 Norme

La norme d'un acteur est la valeur maximale de la dimension de son évaluation.

```
norme- = λ acteur. λ environnement.
  (((if) (alternative?) acteur)
    (then)
      ((max)
        ((norme) (alors-) acteur) environnement)
```

```
      ((norme) (sinon-) acteur) environnement)
(((elsif) (application?) acteur)
  (then) 1)
(((elsif) (atome?) acteur)
  (then)
    (((if) (symbole?) acteur)
      (then)
        ((norme)
          ((cherche) symbole) environnement)
          ((associé) symbole) environnement)
      (else) 1)
(((elsif) (définition?) acteur)
  (then) ((norme) (synonyme-) acteur) environnement)
(((elsif) (extraction?) acteur)
  (then)
    (((if) (vecteur?) (module-) acteur)
      (then)
        (((if) (entier?) (index-) acteur)
          (then)
            ((norme)
              ((index) (index-) (module-) acteur) acteur)
              ((environnement:) (module-) acteur) environnement)
          (else)
            (((reduce) 0) max)
              ((map) λ composante.
                ((norme)
                  composante)
                  ((environnement:) acteur) environnement)
                (valeur-) (module-) acteur)
   (((elsif) (symbole?) acteur)
      (then)
        ((norme)
          ((extraction:)
            ((cherche) acteur) environnement)
            (index-) acteur)
          ((associé) acteur) environnement)
    (else) indéfini:)
(((elsif) (flot?) acteur)
  (then) ((norme) (état-) acteur) environnement)
(else)
  (((reduce) 0) +)
    ((map) λ composante.
      ((norme)
        composante)
        ((environnement:) acteur) environnement)
      (valeur-) acteur
```

16.8 Stabilité

La stabilité pemret de déterminer si la dimension d'un acteur est suppérieure à la dimension de son régénéré, ce qui signifie que sa dimension décroît au cours des régénérations.

```
stable? = λ acteur. λ environnement.
  (((if) (alternative?) acteur)
    (then)
      (((if) ((=?)
                    ((norme) (alors-) acteur) environnement)
                    ((norme) (sinon-) acteur) environnement)
        (then)
          ((and) ((stable?) (condition-) acteur) environnement)
            ((and) ((stable?) (clauseThen-) acteur) environnement)
                       ((stable?) (clauseElse-) acteur) environnement)█
        (else) false)
  (((elsif) (atome?) acteur)
    (then) true)
  (((elsif) (flot?) acteur)
    (then)
      (((if) ((and) (figé?) (état-) acteur)
                     ((≥?)
                       ((norme) (état) acteur) environnement)
                       ((norme) (contrat) acteur) environnement)
        (then) ((stable?) (contrat-) acteur) environnement)
        (else) false)
  (((elsif) (vecteur?) acteur)
    (then)
      (((reduce) true) and)
        ((map) λ composante.
          ((stable?)
            composante)
            ((environnement:) acteur) environnement)
          (valeur-) acteur)
  (else)
    (((reduce) true) and)
      ((map) λ composante.
        ((stable?)
          composante)
          environnement)
        (valeur-) acteur
```

Références

[AB82] M. Auguin and F. Boéri. Efficient multiprocessor architecture for digital processing. *ICASSP*, pages 675–679, 1982.

[AB86] M. Auguin and F. Boéri. The opsila computer. *in Parallel Algorithms and architectures, Eds. M. Cosnard et coll., North Holland*, pages 143–154, 1986.

[ASS87] H. Abelson, G.J. Susman, and J. Susman. *Structure and Interpretation of Computer Programs.* MIT Press, 1987.

[AW76] E.A. Ashcroft and W.W. Wadge. LUCID, a formal system for writing and proving programs. *SIAM j. Comput*, 5(3):336–354, September 1976.

[AW77] E. A. Ashcroft and W. W. Wadge. Lucid, a Nonprocedural Language with Iteration. *j-CACM*, 20(7):519–526, July 1977.

[Bac78] J. Backus. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *j-CACM*, 21(8):613–641, aug 1978.

[BL90] A. Benveniste and P. LeGuernic. Hybrid dynamical systems theory and the SIGNAL language. *IEEE Transactions*, 35:535–546, 1990.

[Cha92] R. Chassaing. *Digital Signal Processing with C and the TMS320C30.* Topics in Digital Signal Processing, 1992.

[Chu51] A. Church. *The Calculi of Lambda-conversion.* Annals of Mathematical Studies, vol. 6, Princeton University Press, Princeton (N.J.), 1951.

[CR90] W. Clinger and J. Rees. Revised^4 report on the algoritmic language scheme. Technical report, MIT Artificial Intelligence Laboratory, CSDTR 174, october 1990.

[CT93] M. Cosnard and D. Trystram. *Algorithmes et architectures parallèles.* InterÉdition, 1993.

[dWB96a] G. de Wailly and F. Boéri. A cad tool chain for signal processing applications, with parallel implementation issues. In *Groningen Information Technology Conference*, february 1996.

[dWB96b] G. de Wailly and F. Boéri. Dataflow language for signal processing modeling with parallel implementations issues. In *VIII European Signal Processing Conference (EUSIPCO'96).* EURASIP, september 1996.

[dWB96c] G. de Wailly and F. Boéri. λ-matrix, a formal approach to functional modelling of synchronous data-flow systems. *EuroMicro Journal - TO APPEAR*, 1996.

[dWB96d] G. de Wailly and F. Boéri. Specification of a functional synchronous dataflow language for parallel implementation with the denotationnal semantics. In *Symposium on Applied Computing.* ACM, february 1996.

[FM91] P. Fradet and D. Le Métayer. Compilation of functional languages by program transformation. In *Transaction on Programming Languages and Systems*, volume 13,1, pages 21–51. ACM, 1991.

[Gal95] E. Gallesio. Stk reference manual, version 2.2. Technical report, Laboratoire I3S-CNRS URA 1376 - ESSI, e-mail:kaolin.unice.fr:/pub/, october 1995.

[HCRP91a] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. Programmation et vérification des systèmes réactifs : le langage LUSTRE. *Techniques et Sciences Informatiques*, 10(2):139–158, 1991.

[HCRP91b] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data flow programming language LUSTRE. In *Proceedings of the IEEE*, pages 1305–1319, 1991. Published as Proceedings of the IEEE, volume 79, number 9.

[Ive62] K. Iverson. *A Programming Language*. Wiley, New York, 1962.

[Jaf90] A. Jaffer. Scm : A scheme implementation 4. Technical report, jaffer@ai.mit.edu, 1990.

[Joh85] T. Johnsson. Lambda lifting : Transforming programs to recursive equations. *Computer Sciences 201, Springer-Verlag, Berlin*, 1985.

[Kah74] G. Kahn. The semantics of a simple language for parallel programming. In *IFIP 74 Congress*. North Holland, Amsterdam, 1974.

[Kri90] J.L. Krivine. *Lambda-Calcul, types et modèles*. Masson, 1990.

[Kun80] M. Kunt. *Traité d'Électricité-Traitements numérique des signaux*. Edition Georgi, 1980.

[Lan66] P.J. Landin. The next 700 programming languages. *Communication of ACM*, 9:157–166, march 1966.

[LB90] E.A. Lee and J.C. Bier. Architectures for statically scheduled dataflow. *Journal of parallel and Distributed Computing*, 10:333–348, 1990.

[Lee91] E.A. Lee. Consistency in dataflow graphs. *IEEE Transactions on Parallel and Distributed systems*, 2(2):223–235, April 1991.

[Llo86] A. Lloyd. *A pratical introduction to denotational semantics*. Cambridge Computer Science Texts 23, 1986.

[LM87] E.A. Lee and D.G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, September 1987.

[Mey92] B. Meyer. *Introduction à la théorie des langages de programmation*. Inter Edition, 1992.

[Miq85] R. Miquel. *Le Filtrage Numérique par microprocesseurs*. édiTESTS, 1985.

[NR85] M. Nivat and J.C. Reynolds. *Algebraic methods in semantics*. Cambridge University Press, 1985.

[Rev88] G.E. Revesz. *Lambda-Calculus, Combinators, and Functional Programming*. Cambrige University Press, 1988.

[Sto77] J.E. Stoy. *Denotational Semantics: The Scott — Strachey Approach to Programming Language Semantics.* MIT Press Series in Computer Science, 1977.

[Str66] C. Strachey. *"Towards a Formal Semantics", Formal Language Description Languages for Computer Programming.* éd. Tom B. Steel Jr., pp. 198-220, North-Holland Puplishing Co., Amsterdam, 1966.

[WA85] W.W. Wadge and E.A. Ashcroft. *Lucid, the DataFlow Programming Language.* Academic Press, 1985.