Extension du langage LUSTRE et application à la conception de circuits : le langage LUSTRE-V4 et le système POLLUX

Frédéric Rocheteau

THESE

présentée par
. ROCHETEAU Frédéric

pour obtenir le grade de DOCTEUR
de l'INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE
(arrêté ministériel du 23 novembre 1988)

(Spécialité : Informatique)

Extension du langage LUSTRE et application à la conception de circuits: Le langage LUSTRE-V4 et le système POLLUX

Date de soutenance : 29 Juin 1992

Composition du jury :	Président	G. Mazaré
	Rapporteurs	G. Berry
		P. Quinton
	Examinateurs	N. Halbwachs
		J. Vuillemin

Thèse préparée au sein du Laboratoire de Génie Informatique

Remerciements

Je tiens à remercier

Monsieur Guy Mazaré, Professeur à l'ENSIMAG pour m'avoir fait l'honneur de présider le jury de cette thèse.

Messieurs Gérard Berry, Maître de recherche à l'école des mines, et Patrice Quinton, directeur de recherche au CNRS, pour l'interêt qu'ils ont porté à ce travail et avoir accepté de le juger.

Messieurs Nicolas Halbwachs, directeur de recherche au CNRS, et Jean Vuillemin, chercheur à Digital et professeur à l'école Polytechnique, qui se sont partagés la tâche d'encadrer ce travail et ont su le faire progresser par leurs conseils judicieux.

Cette thèse a été réalisée, dans le cadre d'une convention CIFRE, au sein du collaboration entre l'équipe SPECTRE *du Laboratoire de Génie Informatique de l'IMAG à Grenoble et l'équipe* PAM *du Centre de Recherche de Paris de Digital.*

Je remercie également tous les membres de ces deux équipes avec lesquels j'ai eu grand plaisir à travailler, tout particulièrement Patrice Bertin et Hervé Touati à Paris et Christophe Ratel et Pascal Raymond à Grenoble.

Sommaire

Introduction

Les langages synchrones ESTEREL [BS91], LUSTRE [HCRP91], SIGNAL [LGLL91] et ARGOS [Mar90] ont été développés en réponse à un besoin, en informatique industrielle, de méthodes de conception sûre de systèmes temps réels. Le but recherché était double :

- faciliter la tâche du programmeur, notamment en lui fournissant des formalismes adaptés, permettant une description plus simple et naturelle de ses applications.
- avoir la possibilité de vérifier des propriétés sur un système.

Ces langages ont adoptés des styles de programmation différents, ESTEREL et ARGOS sont impératifs alors que LUSTRE et SIGNAL sont déclaratifs, mais tous sont basés sur le même modèle d'exécution synchrone. Sous cette hypothèse, le comportement des systèmes temps réels décrits est idéalisé, ils produisent sans délai un vecteur de résultats sur leurs sorties chaque fois qu'un vecteur de valeurs est appliqué à leurs entrées. Par conséquent, tout calcul ou communication est supposé être instantané.

Dans cet environnement simplifié par rapport à la réalité, il a été possible de parfaitement définir ces langages et doter chacun d'eux d'une sémantique formelle complète. Chaque primitive a ainsi un comportement non ambigu et totalement déterministe simplifiant grandement les raisonnements sur le temps

Plus récemment, l'implantation matérielle des langages synchrones a été étudiée, afin d'obtenir une implantation efficace de systèmes temps réels. Ces recherche de performances a deux raisons principales :

- L'hypothèse de synchronisme qui est faite dans les langages synchrones implique en théorie qu'un programme ne peut être exécuté que sur une machine infiniment rapide. Cependant, en pratique, il suffit que cette machine soit suffisamment performante pour que les calculs induits par un vecteur d'entrée puissent être faits avant l'arrivée du vecteur suivant.
- Une part importante des spécifications d'un système temps réel concerne des contraintes temporelles qu'il doit impérativement vérifier, le rythme d'acquisition des entrées ou le temps de réponse maximum à un stimulus par exemple. Les langages synchrones permettent de décrire de telle contraintes dans un programme, mais il faut encore que l'implantation du programme soit suffisamment efficace pour les respecter.

A l'inverse, les langages synchrones également un intérêt certain dans le domaines de la conception de circuits matériels qui, si l'on prend ce terme dans son acception la plus large, peuvent être vus comme des systèmes temps réels particuliers.

Un circuit est généralement décomposé en deux parties :

- Le chemin de données qui effectue les calculs
- Le contrôleur chargé de séquencer le déroulement des calculs

Les différents langages synchrones, en fonction du formalisme employé sont plus particulièrement adaptés pour décrire l'une ou l'autre de ces parties :

- ESTEREL et ARGOS pour les contrôleurs [Ber91]
- LUSTRE ou SIGNAL pour les chemins de données.

Le travail qui va être présenté dans ce document concerne les applications de LUSTRE à la description de circuits. Il s'est déroulé au sein d'une collaboration entre le groupe SPECTRE du Laboratoire de Génie Informatique de Grenoble (LGI) et l'équipe de Jean Vuillemin du Paris Research Laboratory de Digital (PRL), à l'utilisation de LUSTRE comme langage de description de haut niveau pour programmer la PAM ("Programmable Active Memory") [BRV90], un co-processeur matériel programmable basé sur une une matrice de circuits prédiffusés programmables.

Un programme décrivant un circuit présente, généralement une bien plus grande régularité qu'un autre décrivant un système temps réel, mais également une taille pouvant être bien plus importante. Le langage nécessite donc certaines extensions permettant une description plus aisée de circuits de grande taille, en tirant parti de leur régularité. Le point délicat est la définition d'extensions simples et adaptées aux besoins, conservant les caractéristiques fondamentales de LUSTRE, en particulier sa sémantique rigoureuse. Ce choix s'avère également crucial au niveau du processus de compilation. Celui-ci devant faire de nombreux calculs et vérifications statiques, une extension trop générale rendrait la compilation très coûteuse voire impossible à effectuer.

La première partie de ce mémoire donne une présentation informelle du langage.

- Le chapitre 1 décrit la version actuelle de LUSTRE.
- Le chapitre 2 regarde la façon dont le problème a été traité dans les langages existant ou traitant de problèmes similaires.
- Des premières expérimentations ont alors permis de montrer l'intérêt de LUSTRE pour la programmation de la PAM. Le langage fournit un niveau d'abstraction suffisant pour simplifier le développement d'applications, tout en laissant au concepteur la possibilité de contrôler de manière complète et très précise leur implantation. Ceci est primordial sur ce genre de composant pour obtenir de bonnes performances et un taux de remplissage important.

  En revanche, il s'est révélé que la description de circuits de grande taille était fastidieuse, LUSTRE ne disposant d'aucune primitive permettant de simplifier la description de structures régulières. Le chapitre 3 présente en détail les différentes extensions qui ont été apportées à LUSTREpour en simplifier l'usage, tout en respectant ses principales caractéristiques : une sémantique simple et rigoureuse, ainsi que la possibilité de faire de nombreuses vérifications statiques pour détecter au moment de la compilation le plus grand nombre d'erreurs de conception.

La seconde partie donne une sémantique du langage et la définition d'un programme correct.

- Les notations utilisées par la suite sont introduites dans le chapitre 4.
- Le chapitre 5 donne la sémantique d'un langage réduit.
- Les chapitre 6 et 7 complètent cette sémantique en introduisant respectivement les structures nommées et les extensions homomorphes des opérateurs.

La dernière partie. enfin, présente le système POLLUX construit autour de LUSTRE V4. Le langage qu'il traite actuellement est un peu plus restreint que celui décrit dans la première partie de ce document. Nous avons en effet pris en compte dans la définition de LUSTRE V4 les constatations faites et l'expérience acquise lors de l'utilisation en grandeur réelle cette première version de POLLUX. Les différences entre le langage implanté et celui défini sont décrite en introduction du chapitre 8.

- Le chapitre 8 présente le pré-compilateur de ce langage chargé d'analyser un programme LUSTRE V4, de s'assurer qu'il est correct, puis de la transformer en un programme LUSTRE V3.
- Le chapitre 9 présente l'utilisation de LUSTRE pour la description et la synthèse de circuits. Il présente plus particulièrement l'utilisation de LUSTRE pour programmer la PAM.
- Le chapitre 10 reprend la compilation classique du langage sur machine séquentielle.
- Le chapitre 11 décrit l'interface graphique de LUSTRE, la traduction d'un programme textuel en un ensemble de schémas ainsi que l'opération inverse.

Partie I

D'un Lustre à l'autre

Chapitre 1

Lustre V2/V3

1.1 Introduction

Le noyau de LUSTRE se réduit à quelques primitives, très simples, mais suffisamment puissantes et adaptées au domaine dans lequel ce langage est utilisé, pour que leur choix n'ait jamais été remis en cause ou qu'il ait fallu en ajouter de nouvelles. Lorsque LUSTRE a commencé à être employé pour décrire des circuits, ces primitives se sont révélées convenir également à ce genre d'application.

Si LUSTRE a été étendu ce n'est donc en aucun cas pour augmenter la puissance d'expression du langage, mais uniquement pour simplifier l'écriture de programmes de grande taille. Dans ce but plusieurs extensions du langage ont été étudiées :

- Des types structurés, comme des tableaux, tout d'abord pour simplifier la description des parties régulières d'un programme
- Une notion de sous programme plus générale facilitant l'écriture de bibliothèques de constituants de base
- La récursivité bornée
- La possibilité de pouvoir compléter la description du comportement d'un programme par d'autres informations.

Cette partie, consacrée à la présentation informelle de LUSTRE, se décomposera en trois chapitres :

- Le premier donnera une description du langage, tel qu'il était à l'origine de ce travail, et de ses principaux concepts.
- Le second donnera un aperçu des langages dont nous nous sommes inspirés pour étendre LUSTRE.
- Le troisième donnera une description des extensions qui y ont été ajoutées pour former LUSTRE V4.

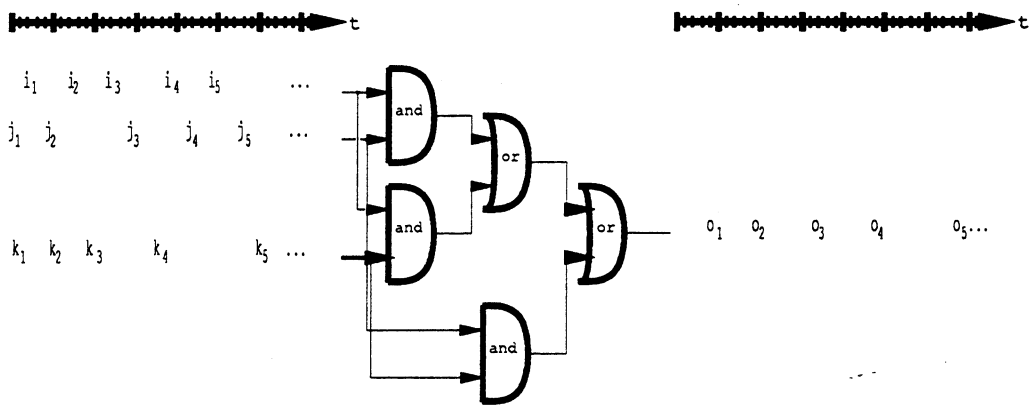


Figure 1.1 *exécution flots de données de la fonction majorité*

1.2 Modèle d'exécution

Un système à flots de données se présente sous la forme d'un réseau d'opérateurs recevant des suites de valeurs sur ses entrées et en produisant d'autres en résultat.

Il n'y a aucun contrôle global de l'exécution, chaque opérateur a un comportement cyclique infini, cadencé uniquement par les données qu'il reçoit. Il s'active chaque fois que toutes ses entrées ont reçu une valeur, effectue un traitement sur ces données, émet éventuellement des résultats, puis attend que ses entrées aient reçu une nouvelle valeur avant de débuter le cycle suivant.

La Figure 1.1 présente un exemple d'exécution d'un système à flots de données calculant la fonction booléenne majorité à trois entrées. La i^{ieme} valeur de sa sortie, o_i, est évaluée une fois connue la i^{ieme} valeur de ses entrées, i_i, j_i et k_i. Elle vaut *true* uniquement si la valeur d'au moins deux entrées est *true* et *false* dans le cas contraire.

Le modèle d'exécution des programmes LUSTRE est basé sur le modèle flots de données. Un programme représente donc un réseau d'opérateurs évoluant au rythme de ses entrées. Dans le but de lui donner un comportement temporel simple et clairement défini, une hypothèse de synchronisme parfait a été faite. Par conséquent,

- Tous les calculs et les transmissions sont supposés être effectués en un temps nul.
- Les flots d'entrées d'un programme sont supposée être cadencés par un même horloge.

A chaque top d'horloge, définissant un instant du programme, toutes les entrées du programme reçoivent simultanément une valeur, à partir desquelles sont instantanément calculées les valeurs correspondantes des sorties.

Ainsi à chaque top d'horloge, toutes les entrées reçoivent simultanément une donnée et les valeurs correspondantes des sorties sont calculées et émises instantanément.

La Figure 1.2 montre ce que devient l'exécution de la fonction majorité sous cette hypothèse de synchronisme.

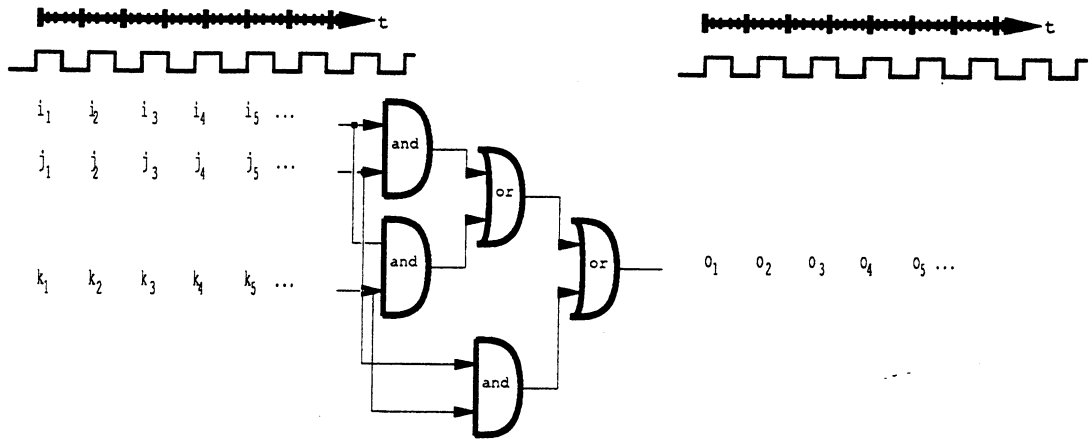


Figure 1.2 *Exécution flots de données synchrone de la fonction majorité*

1.3 Structure des programmes

LUSTRE est un langage déclaratif. Le réseau d'opérateurs d'un programme est décrit au moyen d'un système d'équations de la forme `x = e`, où `x` est une variable et `e`, une expression formée à partir de variables, de constantes et d'opérateurs du langage. Le système d'équations d'un programme LUSTRE étant uniquement un ensemble de définitions, leur ordre est sans importance, il n'a aucune influence sur l'ordonnancement des calculs.

L'équation définissant, par exemple, la fonction majorité est :

```
o = (i and j) or (i and k) or (j and k);
```

La correspondance entre le réseau d'opérateurs et le programme textuel associé est la suivante :

- A toute variable du programme correspond un fil du réseau. Un identificateur de variable peut être vu comme une étiquette attachée à un fil.
- A chaque opérateur du programme correspond un opérateur du réseau.
- Une constante correspond à un opérateur sans entrée, produisant à tout instant la même valeur.

Le résultat de toute expression `e` est une suite infinie de valeurs, celle produite par la sortie du sous-réseau qu'elle décrit. Son i^{ieme} élément, interprété comme étant la valeur de `e` au i^{ieme} instant, est noté e_i.

Une suite de valeurs est également associée à chaque constante ou variable :

- Dans le premier cas il s'agit d'une suite de valeurs constantes. Par exemple, la suite associée à la constante booléenne `true` est *(true, true, true, ...)*.

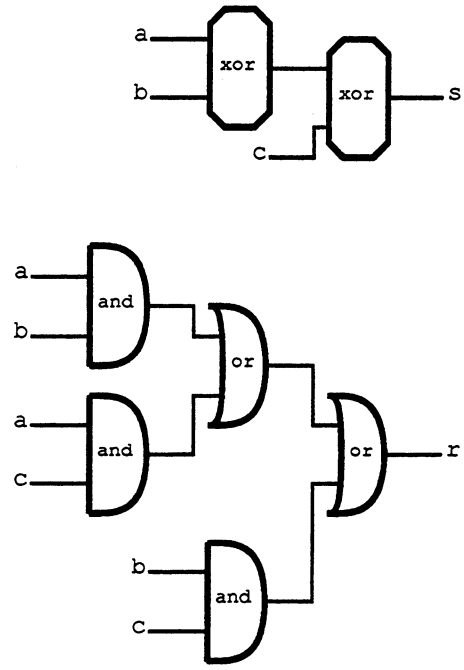


Figure 1.3 *Réseau d'opérateurs d'une cellule d'additionneur*

- Dans le second cas, la suite est donnée par l'équation définissant la variable. Une équation `X = e` indique qu'il y a identité complète entre les deux membres, c'est à dire que $\forall i$ $X_i = e_i$. Aussi, toute occurrence de `X` dans une expression peut être remplacée par une occurrence de `e` et vice versa, sans modifier le comportement du programme.

Un programme LUSTRE est rejeté comme incorrect à la compilation si son système d'équations n'admet pas une unique solution, c'est à dire s'il ne permet pas de déterminer complètement la suite de valeur de chacune de ses variables, en fonction de celle des entrées. Les conditions suffisantes assurant l'unicité de la solution sont :

- qu'il y ait exactement une équation par variable, exception faite des variables d'entrée qui ne doivent en avoir aucune, leurs suites de valeurs étant fournies par l'environnement.
- qu'il n'y ait aucune boucle combinatoire. Cela signifie que dans une équation `x = e`, e_i ne peut être calculée à partir de X_i.

Voici un premier exemple, un programme LUSTRE décrivant une cellule d'additionneur binaire complet et le réseau d'opérateurs associé Figure 1.3

```
node Add1 (a,b,c:bool) returns (s,r:bool);
let
   s = a xor b xor c;
```

```
    r = (a and b) or (a and c) or (b and c);
end;
```

Etant donné le domaine d'application de LUSTRE le parti-pris a été de doter le langage d'une syntaxe stricte et redondante dans le but de permettre un maximum de vérifications statiques et ainsi détecter dès la phase de compilation le maximum d'erreurs. Ainsi, le système d'équations est précédé par un entête où sont déclarés

- le nom du programme
- ses variables d'entrée et de sortie ainsi que leurs types
- le cas échéant, ses variables locales et leurs types

Dans l'exemple précédent, le programme se nomme `Add1`, il comporte trois entrées de type `bool`, `a` et `b`, qui sont les deux bits à additionner, ainsi que `c`, la retenue entrante. Ses sorties sont `s` la somme et `r` la retenue sortante. Le système d'équations quant à lui spécifie que la somme est égale au "ou"' exclusif des entrées du programme et la retenue à leur majorité. Voici, pour illustrer, un exemple d'exécution de ce programme.

`a` : (	*false*	, *true*	, *false*	, *false*	, *true*	, *true*	, *true*	, *false*	, *true*	, ...)
`b` : (	*false*	, *true*	, *true*	, *false*	, *true*	, *false*	, *false*	, *true*	, *false*	, ...)
`c` : (	*false*	, *true*	, *false*	, *true*	, *false*	, *false*	, *true*	, *true*	, *false*	, ...)
`s` : (	*false*	, *true*	, *true*	, *true*	, *false*	, *true*	, *false*	, *false*	, *true*	, ...)
`r` : (	*false*	, *true*	, *false*	, *false*	, *true*	, *false*	, *true*	, *true*	, *false*	, ...)

1.4 Les types

A chaque expression LUSTRE est associé un type, il définit le domaine des valeurs que peut prendre cette expression à chaque instant, ainsi que les opérateurs qui peuvent lui être appliqués. Il y a en LUSTRE trois types de données prédéfinis :

- Les booléens (`bool`).
- Les entiers relatifs (`int`)
- Les réels (`real`)

Les constantes associées à ces types sont notées :

- pour les booléens, *true* pour la valeur vrai et *false* pour la valeur faux.
- pour les deux autres types, avec une syntaxe similaire à celle employée dans les langages C ou PASCAL.

1.5 Les opérateurs

1.5.1 Les opérateurs sur les valeurs

Nous appelons ainsi les opérateurs classiques étendus pour opérer point par point sur les suites : la valeur d'un opérateur à l'instant i est le résultat de son application aux i^{iemes} valeurs de ses entrées. Par exemple, étant donné un opérateur `op` à deux entrées, le résultat de l'expression `e1 op e2` est :

`e1` : (	$e1_0$	, $e1_1$	, $e1_2$	, $e1_3$	, $e1_4$	, ...)
`e2` : (	$e2_0$	, $e2_1$	, $e2_2$	, $e2_3$	, $e2_4$	, ...)
`e1 op e2` : (	$e1_0$ *op* $e2_0$	, $e1_1$ *op* $e2_1$	, $e1_2$ *op* $e2_2$,	$e1_3$ *op* $e2_3$	, $e1_4$ *op* $e2_4$	, ...)

LUSTRE dispose des opérateurs conditionnels, logiques, arithmétiques ou de comparaison traditionnels que l'on trouve dans tous les langages usuels. Les priorités d'opérateurs sont également classiques.

Il y a un seul opérateur sur les valeurs, en LUSTRE, qui ne soit pas classique :

> `#(e1, e2, ... , en)` est un opérateur booléen n-aire dont la valeur est *true* chaque fois qu'au plus une de ses entrées `ei` vaut *true*. Dualement, sa valeur est `false` chaque fois que deux de ses entrées au moins valent `true`. Par exemple,
>
> - `#(true,false,false)` ou `#(false,false,false)` valent *true*
> - `#(true,false,true)` ou `#(true,true,true)` valent *false*

1.5.2 Opérateurs sur les suites

Aussi appelés opérateurs temporels, ce sont des opérateurs propres à LUSTRE qui permettent de manipuler les suites.

1.5.2.1 Opérateur de retard

Si `e` est une expression de type quelconque, alors le résultat de l'expression `pre e` est la suite de valeurs associée à `e`, décalée d'un élément. Au premier instant, la valeur de `pre e` est *nil* (la valeur indéfinie), au i^{ieme} instant sa valeur est celle de `e` à l'instant précédent.

`e` : (	e_0	, e_1	, e_2	, e_3	, e_4	, e_5	, e_6	, e_7 , ...)
`pre e` : (	*nil*	, e_0	, e_1	, e_2	, e_3	, e_4	, e_5	, e_6 , ...)

1.5.2.2 Opérateur d'initialisation

Si `e1` et `e2` sont deux expressions de même type, alors le résultat de l'expression `e1 -> e2` est la suite de valeurs ayant pour premier élément le premier élément de la suite associée à `e1` et pour i^{ieme} élément le i^{ieme} élément de la suite associée à `e2`.

$$
\begin{array}{rl}
\texttt{e1} : (& e1_0 \ , e1_1 \ , e1_2 \ , e1_3 \ , e1_4 \ , e1_5 \ , e1_6 \ , e1_7 \ , \ldots) \\
\texttt{e2} : (& e2_0 \ , e2_1 \ , e2_2 \ , e2_3 \ , e2_4 \ , e2_5 \ , e2_6 \ , e2_7 \ , \ldots) \\
\texttt{e1 -> e2} : (& e1_0 \ , e2_1 \ , e2_2 \ , e2_3 \ , e2_4 \ , e2_5 \ , e2_6 \ , e2_7 \ , \ldots)
\end{array}
$$

1.6 Hiérarchie

Ecrire un programme LUSTRE revient à définir un nouvel opérateur du langage à partir d'autres plus simples. Une fois déclaré, un tel opérateur, appelé noeud (**node**), peut être instancié de manière fonctionnelle dans n'importe quelle expression LUSTRE. Il reçoit une liste de paramètres d'entrée et produit une liste de paramètres de sortie. Il est ainsi possible d'écrire un programme à l'aide d'une hiérarchie d'opérateurs de plus en plus complexes.

Voici par exemple la définition de l'opérateur `Maj` qui calcule la fonction majorité à trois entrées

```
node Maj (x,y,z : bool) returns (maj : bool);
let
    maj = (x and y) or (y and z) or (z and x);
tel;
```

Il est utilisé ci-dessous dans la définition d'une cellule d'additionneur :

```
node Add1s (a,b,c : bool) returns (s,r : bool);
let
    s =  a xor b xor c ;
    r =  Maj(a,b,c);
tel;
```

Il est possible d'indiquer qu'un paramètre de noeud est une constante en précédant sa déclaration du mot clé `const`

1.7 Assertions

Un programme LUSTRE peut contenir, en plus du système d'équations définissant son comportement, un ensemble d'assertions, de la forme **assert e** ou e est une expression booléenne quelconque, définissant les propriétés qu'il vérifie à tout instant de son exécution.

Les assertions, suivant l'utilisation qui est faite de LUSTRE peuvent être interprétées de deux façons :

- comme les conditions sous lesquelles le programme a un comportement correct
- comme les propriétés vérifiées par le programme s'il a un comportement correct.

Dans le premier cas, les assertions permettent d'indiquer que le comportement du programme est moins général que celui décrit par son système d'équations, parce que des contraintes sur

l'environnement limitent les états dans lesquels il peut se trouver. Par exemple, il est possible de spécifier qu'à chaque instant une des deux entrées de l'additionneur vaut *true* en rajoutant l'assertion suivante :

```
assert a or b;
```

Ce genre d'assertions est utilisé lors de la compilation de LUSTRE pour essayer de produire un code plus efficace ou pour prouver, sous certaines hypothèses, qu'un programme vérifie bien certaines propriétés.

Dans le second cas, les assertions permettent de vérifier à l'exécution qu'un programme est bien conforme à sa spécification. Lors de la phase de compilation, un morceau de code supplémentaire est produit pour chaque assertion, il provoque l'arrêt du programme dès qu'une assertion n'est pas vérifiée. Par exemple, si la cellule d'additionneur est correctement décrite, la valeur de la retenue `r` est toujours *true* lorsque `a` et `b` valent *true*. Cette propriété est décrite par l'assertion :

```
assert Implies(a and b, r);
```

Implies étant défini comme :

```
node Implies (a,b:bool) returns (c:bool);
let
   c = not a or b;
end;
```

1.8 Objets importés

LUSTRE étant un langage spécialisé, il n'est pas adapté pour décrire des parties de programmes non purement flots de données ou provoquant des effets de bords, l'affichage de résultats sur un écran par exemple. Ces descriptions peuvent être en revanche très aisées à faire avec des langages plus généraux (en C ou Pascal).

Pour cette raison, LUSTRE possède un mécanisme permettant de s'interfacer avec d'autres langages, de déclarer et d'utiliser des objets externes :

- Des types qui sont considérés comme étant de nouveaux types atomiques. Absolument aucune information sur leur structure n'est fournie. La déclaration d'un type externe `t` est uniquement :

  ```
  type t;
  ```

  Il est possible de déclarer des variables dont le type est importé mais, pour pouvoir donner leur définition par une équation, il est nécessaire d'importer également les fonctions permettant de manipuler des expressions de ce type, une fonction de comparaison par exemple.

- Des fonctions (`function`). Il s'agit, tout comme les noeuds, de définitions de nouveaux opérateurs mais ayant la particularité d'être purement combinatoires, chaque sortie est supposée dépendre instantanément de chaque entrée. Une fonction appelée deux fois avec les mêmes valeurs d'entrée produit toujours les même valeurs de sorties (absence d'effet de bord). Une déclaration de fonction ne consiste donc qu'en un entête où sont déclarées ses entrées et ses sorties. Par exemple, la déclaration

```
function blop (i1:t1; i2:t2; i3:t3) returns (o1:u1; o2:u2);
```

  définit un opérateur externe nommé `blop` ayant trois entrées `i1`, `i2` et `13` de types respectifs `t1`, `t2` et `t3` et deux sorties `o1`, de type `u1` et `o2`, de type `u2`.

- Des constantes qui, tout comme les types importés, sont des objets complètement abstraits dont seul le nom et le type sont connus. Par exemple, la déclaration d'une constante `c` de type `t` est faite par :

```
const c:t;
```

Considérons par exemple un type externe COLOR. déclaré par :

```
type COLOR;
```

Les couleurs, sont des constantes externes de ce type, déclarées de façon similaire :

```
const white, black, red, green, blue, yellow : COLOR;
```

Considérons enfin que les seuls opérateurs disponibles sont :

- l'opérateur de comparaison qui retourne une valeur booléenne *true* si `c1` est plus foncé que `c2`.

```
function darker (c1,c2:COLOR) returns (b:bool);
```

- l'opérateur qui permet de mélanger deux couleurs pour en produire une troisième :

```
function blend (c1,c2:COLOR) returns (c3:COLOR);
```

Un programme ayant importé ceci, peut ensuite déclarer et manipuler des variables de types `COLOR` :

```
...
x : COLOR
...
x = black -> if darker (pre x, gray) then blend(pre x, white) else blue;
```

Le comportement de la variable x décrit par cette équation est : La couleur est initialement noire et tant qu'elle est plus foncée que la couleur grise, elle est mélangée avec du blanc. Une fois que le mélange devient plus clair, la couleur devient bleue.

1.9 Horloge logique

1.9.1 Définition

En LUSTRE l'hypothèse de synchronisme a été un peu reliâchée en introduisant la notion d'horloge logique. Dans un programme LUSTRE purement synchrone, toutes les expressions calculent une nouvelle valeur à chaque top d'horloge, toutes les suites qui leurs sont associées ont le même nombre d'éléments.

Mais LUSTRE permet aussi de décrire des programmes composés de blocs évoluant à des rythmes différents, dans lesquels les expressions ne calculent des valeurs que sur un sous-ensemble des tops d'horloge.

Pour cela, à toute expression `e` est associé, en plus d'une simple suite de valeurs, une expression booléenne `h`, son horloge logique. Une expression n'a de valeur que si au même instant, son horloge logique vaut *true*. Ainsi, `e` prendra sa n^{ieme} valeur à l'instant où `h` vaudra *true* pour la n^{ieme} fois.

Nous appellerons dorénavant flot, un couple formé d'une suite de valeur et d'une horloge logique. Il est à noter qu'une horloge logique étant définie par une expression booléenne, un flot booléen lui est également associé.

`h` : (	*false*	, *true*	, *false*	, *false*	, *true*	, *true*	, *true*	, *false*	, *true*	, ...)
`x` : (		x_0 ,			, x_1	, x_2	, x_3		, x_4	, ...)

Un flot qui a une valeur à chaque top d'horloge, a donc pour horloge logique un flot dont la valeur est *true* à chaque instant, c'est à dire le flot associé à la constante `true` (par définition, les constantes ont une valeur à chaque instant). C'est pour cela que l'horloge qui séquence un programme LUSTRE, aussi appelée horloge de base, est notée `true`. Cette définition impose par conséquent que l'horloge logique de `true` est `true` elle même.

Comme exemple, nous allons définir de nouveau une cellule d'additionneur fonctionnant cette fois, non pas sur l'horloge de base, mais sur une horloge h.

La déclaration des variables `a,b,c` est omplétée par l'information `when h` qui indique que ces variables ont `h` pour horloge.

```
node Add1h (h:bool;(a,b,c:bool) when h) returns (s,r:bool);
let
   s = a xor b xor c;
   r = (a and b) or (a and c) or (b and c);
tel;
```

les entrées `a,b` et `c` ne reçoivent de valeurs qu'aux instants ou `h` vaut `true`. Les expressions, qui dépendent de ces variables, ne peuvent faire de calcul qu'à ces instants là. Par conséquent le noeud ne produit de résultats que si `h` vaut `true`.

L'exemple d'exécution précédent devient alors en fonction de h :

h:(	*true*	, *false*	, *true*	, *true*	, *false*	, *true*	, *true*	, *true*	, *false*	, *true*	, *false*	,...)
a:(	*false*		, *true*	, *false*		, *false*	, *true*	, *true*		, *true*		,...)
b:(	*false*		, *true*	, *true*		, *false*	, *true*	, *false*		, *false*		,...)
c:(	*false*		, *true*	, *false*		, *true*	, *false*	, *false*		, *true*		,...)
s:(	*false*		, *true*	, *true*		, *true*	, *false*	, *true*		, *false*		,...)
r:(	*false*		, *true*	, *false*		, *false*	, *true*	, *false*		, *true*		,...)

1.9.2 Opérateurs de changement d'horloge

Tous les opérateurs présentés jusqu'à présent reçoivent en entrée des flots de données ayant la même horloge logique qui est aussi celle du flot de données produit. Cette contrainte sur les horloges logiques permet de s'assurer que tout programme LUSTRE ne nécessitera pas une mémoire potentiellement infinie pour être exécuté.

Ce serait le cas, par exemple, si l'on permettait à un opérateur d'avoir une de ses entrées sur l'horloge toujours vraie `true` et une autre sur une horloge logique vraie un cycle sur deux. L'opérateur ne pourrait calculer un résultat qu'un cycle sur deux, une mémoire infinie serait alors nécessaire pour stocker les valeurs de la première entrée non utilisées.

Deux opérateurs temporels supplémentaires permettent de manipuler les horloges.

1.9.2.1 Opérateur d'échantillonnage

Si `e` est une expression de type quelconque et `h` une expression booléenne, le résultat de l'expression `e when h` est un flot, d'horloge logique `h`, dont la suite est construite à partir des valeurs de `e` aux instants où `h` vaut *true*. La i^{ieme} valeur de `e when h` est la valeur de `e` à l'instant où `h` vaut *true* pour la i^{ieme} fois.

e : (	e_0	, e_1	, e_2	, e_3	, e_4	, e_5	, e_6	, e_7	, ...)
h : (	*false*	, *true*	, *true*	, *false*	, *false*	, *true*	, *false*	, *false*	, ...)
f = e when h : (		f_0=e_1	, f_1=e_2			, f_2=e_5			, ...)

1.9.2.2 Opérateur de projection

Si `f` est une expression de type quelconque, d'horloge logique `h`, le résultat de l'expression `current f` est un flot dont l'horloge logique est la même que celle de `h` et dont la valeur courante est identique à celle de `f` si `h` vaut *true*, identique à sa valeur précédente sinon.

e : (	e_0	, e_1	, e_2	, e_3	, e_4	, e_5	, e_6	, ...)
h : (	*false*	, *true*	, *true*	, *false*	, *false*	, *true*	, *false*	, ...)
f = e when h : (		$f_0 = e_1$	, $f_1 = e_2$			, $f_2 = e_5$		, ...)
g = current f : (	g_0 =*nil*	, $g_1 = e_1$	, $g_2 = e_2$	, $g_3 = e_2$	, $g_4 = e_2$	, $g_5 = e_5$	, $g_6 = e_5$	, ...)

1.9.3 Horloge d'un opérateur

Le comportement d'un noeud est décrit relativement à sa propre horloge de base, un paramètre implicite de chaque opérateur, qui détermine le rythme auquel il évolue. L'horloge de base d'un

opérateur est définie comme l'horloge la plus rapide de ses paramètres. Si elle n'existe pas, cela signifie que programme est incorrect, les horloges des opérandes de l'opérateur sont incompatibles avec sa définition.

Un opérateur n'est activé qu'aux instants où son horloge de base vaut *true*, tout se passe pour lui comme si aucun autre instant n'existait.

La noeud `Maj` (§ 1.2), par exemple, décrit un opérateur dont tous les éléments ont son horloge de base pour horloge logique. L'horloge de base de l'instance de `Maj` dans le noeud **`Add1s`** (§ 1.2) est l'horloge de base de ce dernier. Par contre, dans le noeud suivant son horloge de base est `h`.

```
node Add1h (h:bool;(a,b,c:bool) when h) returns ((s,r:bool) when h);
let
   s =  a xor b xor c ;
   r = Maj(a,b,c);
end;
```

Une remarque importante, échantillonner un noeud en entrée est différent d'échantillonner un noeud en sortie. Pour illustrer cela, considérons le noeud suivant :

```
   node Fibo (reset:bool) returns (f_2:int);
   var
start:bool;
      f_1,f:int;
   let
start = true -> reset;
      f_2 = if start then 1 else pre f_1;
      f_1 = if start then 1 else pre f;
      f = f_1 + f_2;
   tel;
```

Le noeud *Fibo* produit la suite des nombres de Fibonacci. Un nouvel élément est produit à chaque cycle et chaque fois que reset vaut `true` l'énumération des éléments de la suite recommence au début. `f` est l'élément courant, le n^{ieme} de la suite (n>=2), `f_1` et `f_2` respectivement ceux de rang n-1 et rang n-2.

Dans l'équation `r = Fibo (n when h)` le noeud `Fibo` est échantillonné en entrée, son horloge de base est `h`, il n'est actif qu'aux cycles où `h` vaut *true*. `n` est une variable booléenne sur l'horloge de base, `r` une variable d'horloge logique `h`.

`h` : (	*true*	, *false*	, *true*	, *true*	, *false*	, *true*	, *false*	, *true*	, ...)
`n` : (	*true*	*true*	, *false*	, *true*	*true*	, *false*	*true*	, *true*	, ...)
`r` : (	*2*		, *2*	, *3*		, *3*		, *5*	, ...)

Dans l'équation suivante par contre, `r = (Fibo (n))when h` , `Fibo` est échantillonné en sortie, son horloge de base est celle du programme. Il est actif à chaque cycle, mais sa valeur n'est considérée que lorsque `h` vaut *true*. Pour les même valeurs d'entrées le résultat précédent devient :

h : (	*true*	, *false*	, *true*	, *true*	, *false*	, *true*	, *false*	, *true*	, ...)
n : (	*true*	*true*	, *false*	, *true*	*true*	, *false*	*true*	, *true*	, ...)
r : (	*2*		, *3*	, *5*		, *8*		, *13*	, ...)

1.9.4 Calcul d'horloge

La détermination de l'horloge de toutes les expressions est le résultat d'un calcul de plus petit point fixe. Initialement, seule l'horloge des constantes, des variables d'entrées et éventuellement de quelques autres variables est connue, celles de toutes les autres expressions sont indéfinies.

- L'horloge de toutes les constantes est `true`
- La déclaration de la variable `x`
  - `x:t` définit que son horloge est `true` si `x` est une variable d'entrée, indéfinie sinon.
  - `(x:t) when h` définit que son horloge est `h`

  `t` étant un type et `h` une autre variable booléenne.

Ensuite, les horloges sont propagées à chaque itération des entrées vers les sorties.

- Les horloges des sorties d'un opérateur peuvent être calculées dès que celle d'une de ses entrées n'est plus indéfinie.
- L'horloge d'une variable `x`, définie par l'équation `x = e`, est connue dès que celle de l'expression `e` a été calculée. `x` hérite de l'horloge de `e`.

Ce calcul s'arrête dès que plus aucune nouvelle horloge ne peut être calculée ou dès qu'une incohérence est détectée, deux entrées d'un opérateur ayant, par exemple, des horloges définies différentes. Dans ce cas le programme est déclaré incorrect. Si à la fin du calcul de point fixe, les horloges de certaines expressions sont encore indéfinies le programme est aussi rejeté.

1.9.5 Exemple

Le programme suivant est un compteur de fronts, il calcule le nombre de transitions de 0 à 1 dans une suite de booléens. Sa sortie `c` est un entier dont la valeur à un instant donné est égale au nombre de fronts montants détectés sur l'entrée booléenne `i` du programme, entre l'instant initial et l'instant courant.

```
node EdgeCounter (i:bool) returns (c:int);
var
   e:bool;
let
   e = false -> (i and not pre i);
   c = current ((0 -> (pre c+1)) when (true->e));
end;
```

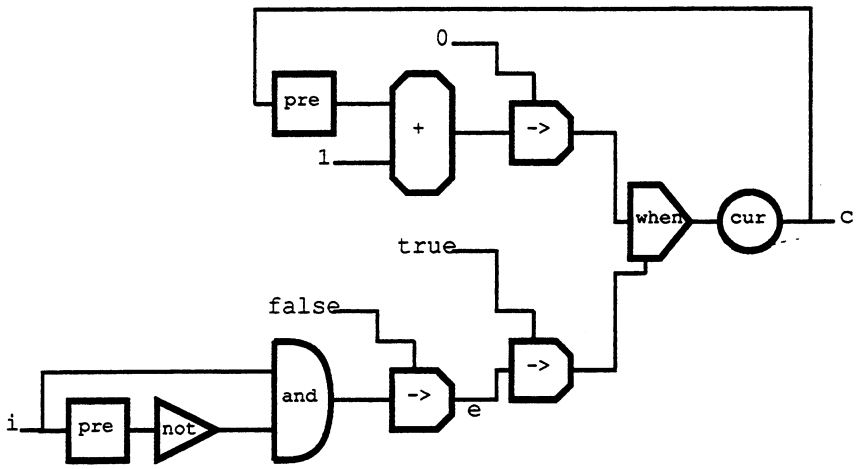


Figure 1.4 *le réseau d'opérateurs du programme edgeCounter*

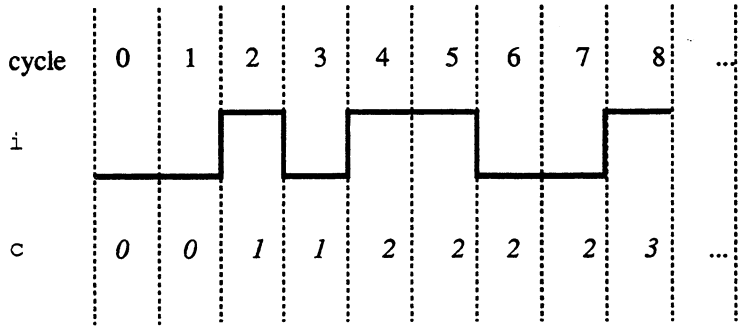


Figure 1.5 *un exemple d'exécution*

Voici un exemple illustrant le fonctionnement de ce programme :

Le rôle de la première équation :

```
e = false -> (i and not pre  i);
```

est de détecter les fronts. La variable `e` vaut *true* s'il vient d'y avoir un front montant, c'est à dire si `i` vaut *true* à l'instant courant et valait *false* l'instant précédent (c'est la signification de l'expression `i and not pre i`). A l'instant initial, l'expression `pre i` est indéfinie et donc l'expression `i and not pre i` aussi. C'est pour cela qu'on force `e` à *false* au premier instant, nous supposons qu'il n'y a pas de front montant au départ.

Voici pour la même séquence d'exécution que précédemment les suites de valeurs associées aux principaux éléments de cette première équation :

i:(	*false*	, *false*	, *true*	, *false*	, *true*	, *true*	, *false*	, *false*	, *true*	,...)
pre i:(	*nil*	, *false*	, *false*	, *true*	, *false*	, *true*	, *true*	, *false*	, *false*	,...)
not pre i:(	*nil*	, *true*	, *true*	, *false*	, *true*	, *false*	, *false*	, *true*	, *true*	,...)
i and not pre i:(	*nil*	, *false*	, *true*	, *false*	, *true*	, *false*	, *false*	, *false*	, *true*	,...)
e:(	*false*	, *false*	, *true*	, *false*	,*true*	, *false*	, *false*	, *false*	, *true*	,...)

Le rôle de la seconde équation :

```
c = current((0 -> (pre c + 1)) when (true -> e));
```

est de compter le nombre de fois qu'un front montant a été détecté depuis l'instant initial, c'est à dire le nombre de fois où la variable `i` a pris la valeur *true* depuis l'instant initial. elle est dérivée de l'équation :

```
x = 0-> (pre x + 1);
```

qui affecte à la variable `x` la suite des entiers naturels. Cette variable a pour valeur *0* à l'instant initial et la valeur de l'instant précédent augmentée de un ailleurs.

Dans notre cas il ne s'agit pas d'incrémenter la valeur de `c` à chaque instant mais seulement lorsque `e` vaut *true*, ce que l'on réalise en échantillonnant le compteur avec cette variable. en fait l'horloge logique utilisée est `true -> e` et non pas `e` pour initialiser le compteur à *0* au premier instant.

L'expression `(0 -> (pre c + 1)) when (true -> e)` définit donc que la valeur de `c` est augmentée de 1 les instants où un front est détecté. Il ne reste plus qu'à définir que la valeur de `c` reste inchangée les autres instants, ce qui est fait en reprojetant l'expression sur l'horloge de base par l'opérateur `current`.

e:(	*false*	, *false*	, *true*	, *false*	, *true*	, *false*	, *false*	, *false*	, *true*	,...)
true -> e:(	*true*	, *false*	, *true*	, *false*	, *true*	, *false*	, *false*	, *false*	, *true*	,...)
x = 0-> pre x + 1:(	*0*	, *1*	, *2*	, *3*	, *4*	, *5*	, *6*	, *7*	, *8*	,...)
(0->(pre c +1)) when (true->e):(	*0*		, *1*		, *2*				, *3*	,...)
c:(	*0*	, *0*	, *1*	, *1*	, *2*	, *2*	, *2*	, *2*	, *3*	,...)

Chapitre 2

Tour d'horizon

Un compteur binaire cascadable est un exemple très simple mais néanmoins :

- très représentatif des opérateurs de base utilisés dans la plupart des circuits.
- suffisant pour se convaincre de la nécessité d'étendre LUSTRE, et permettre ainsi son utilisation dans le domaine de la description de circuits.

Le comportement de la cellule de base de cet opérateur est définie par les deux équations temporelles suivantes :

$$value_t = value_{t-1}\ xor\ count$$

$$carry_t = value_{t-1}\ and\ count$$

Cette cellule se décrit naturellement avec un noeud LUSTRE :

```
   node Cell (count:bool) returns (value, carry:bool);
var
oldvalue:bool;
   let
      oldvalue = false -> pre value;
      value = oldvalue xor count;
      carry = oldvalue and count;
   tel;
```

Construire un compteur n bits consiste à chaîner n cellules, en connectant l'entrée de chacune d'entre elles à la retenue (la sortie `carry`) de la précédente. Mais, la seule solution offerte par LUSTRE V3 pour décrire cet opérateur, consiste à fixer la valeur de n et écrire une équation par cellule. Par exemple, pour n = 4 :

```
node Counter (count:bool) returns (value_0, value_1, value_2, value_3,
   carry:bool);
```

```
var carry_0, carry_1, carry_2 : bool;
let
   (value_0,carry_0) = Cell(count);
   (value_1,carry_1) = Cell(carry_0);
   (value_2,carry_2) = Cell(carry_1);
   (value_3,carry) = Cell(carry_2);
tel;
```

Cette description, déjà lourde et difficile à manipuler pour n valant quatre, devient complètement illisible et inutilisable pour des valeurs plus grandes.

Cet exemple explique pourquoi deux des extensions mentionnées dans l'introduction (§ 1.1) ont été faites :

- L'introduction des tableaux tout d'abord, pour permettre de donner une définition plus concise d'opérateurs ayant une structure régulière. La récursivité bornée a également été introduite pour cette raison.
- L'introduction de noeuds plus généraux ensuite pour ne pas avoir, par exemple, à écrire un noeud pour chaque taille de compteur utilisée, mais un noeud unique paramétré par son nombre de cellules.

Pour définir les différentes extensions de LUSTRE, et plus particulièrement y introduire une notion de tableaux, nous nous sommes intéressés à différents langages. Le but était de voir s'ils avaient eu des problèmes similaires et dans ce cas, les solutions qu'ils avaient adopté ainsi que la mesure dans laquelle ces solutions pouvaient être appliquées à LUSTRE.

Après un bref rappel des extensions de LUSTRE qui avaient été proposées au préalable, le chapitre se découpera de deux parties :

- Un aperçu des langages dont nous nous sommes inspirés.
- Un tour d'horizon rapide des langages de description de matériel les plus proches de LUSTRE ou les plus utilisés.

2.1 Historique des tableaux en LUSTRE

Une notation pour les tableaux a été proposée comme extension future par [Pla88]. Des exemples d'utilisation peuvent d'ailleurs être trouvés :

- dans [HP86] pour la description d'algorithmes systoliques
- dans [Pay91] pour la programmation d'une machine cellulaire.

Avec cette syntaxe, un type tableau est déclaré en donnant la liste de ses bornes, des intervalles d'entiers dont les bornes sont connues à la compilation. Par exemple,

- `type vector = array [1..5] of int;` définit `vector` comme étant le type d'un tableau de 5 entiers indicés de 1 à 5.

- `type matrix = array [1..5][12..24] of bool;` définit `matrix` comme étant le type d'un tableau de booléens à deux dimensions dont le premier indice est compris entre 1 et 5 et le second entre 12 et 24.

Ces types peuvent ensuite être utilisés pour définir des variables. Par exemple, `var x:vector;` définit que x est un tableau de 5 entiers indicés de 1 à 5.

Le seul moyen de manipuler un tableau est élément par élément, Par exemple `x[4] = 12;` définit que la valeur de l'élément d'indice 4 de x vaut 12.

Toutefois, l'opérateur `for` permet de simplifier l'écriture de parties régulières. Par exemple,

```
for i in 1 .. 5
let
   x[i] = 1;
tel;
```

définit que tous les éléments de x valent 1.

Avec cette extension, l'écriture d'un compteur 4 bits se simplifie et devient :

```
node Counter (count:bool) returns (value: array [0..3] of bool; carry:bool);
   var c: array [-1..3] of bool;
   let
      c[-1] = count;
      for i in [0..3]
      let
         (value[i], c[i]) = Cell(c[i-1]);
      tel;
      carry = c[3];
```

Une telle description d'un compteur présente l'avantage, sur la précédente, d'avoir une taille indépendante du nombre de cellules composant le compteur. Le fait que la syntaxe de tableaux employée soit classique peut également être considéré comme un avantage, notamment au niveau de l'apprentissage du langage, mais également, sous certains autres aspects, comme un inconvénient :

- La syntaxe très générale proposée permet d'écrire des dépendances arbitrairement complexes entre éléments de tableaux rendant l'analyse statique et la génération de code difficiles à réaliser autrement qu'en expansant au préalable tous les tableaux.

  Il s'agit là d'un inconvénient majeur, l'introduction de tableaux en LUSTRE ayant pour but de simplifier, non seulement la description d'applications de grande taille, mais également leur compilation et leur exécution.

  Des limitations sur les dépendances possibles ont été cherchées mais aucune n'a donné satisfaction. Toute limitation était trop restrictive ou trop faible et permettait alors d'écrire des programmes comme celui-ci :

```
for i in 0 .. N
let
   x[i,i] = 1;
   x[i,N-i] = 3;
tel;
```

Si N est impair, le programme est correct, s'il est pair l'élément central du tableau est défini deux fois.

- Le `for` n'est pas une construction qui s'intègre bien au langage. La notion de bloc qu'elle introduit se combine mal avec l'aspect flot de données. Par exemple, un programme utilisant des tableaux, définis à l'aide d'un `for`, n'est pas représentable par un réseau d'opérateurs flots de données.
- Le principe de substitution est également perdu.

Ces inconvénients, complétés par le fait, mentionné dans l'introduction, que les applications traitées jusqu'à présent n'avaient pas un besoin impératif de tableaux, font que cette extension n'a jamais été incluse au langage et qu'une autre solution a été recherchée.

2.2 Tableaux

2.2.1 Alpha

ALPHA [Mau89] [DVPY91] est un langage spécialisé dans la description et la manipulation d'architectures parallèles synchrones, plus particulièrement d'algorithmes systoliques [Kun82] [KL80]. D'autres langages du même genre existent [Che86], mais ALPHA, à cause des nombreuses caractéristiques qu'il possède en commun avec LUSTRE, nous a plus particulièrement intéressés.

Ce langage est, comme LUSTRE, un langage déclaratif, un programme se présente sous la forme d'un système d'équations. Mais, à la différence de ce dernier, une variable n'est pas une fonction de $\mathbb{N}$ vers un ensemble de valeurs mais une fonction d'un polyèdre convexe de $\mathbb{Z}^n$ (appelé domaine spatial) vers un ensemble de valeurs. Par exemple, `x:{i|i>=0} of bool` est la déclaration d'une variable `x` qui associe une valeur booléenne à chaque point de $\mathbb{Z}$ ayant une coordonnée positive ou nulle.

Les manipulations des domaines spatiaux se font par deux types d'opérateurs :

- Les opérateurs immobiles, les équivalents des opérateurs sur les valeurs que l'on trouve en LUSTRE, qui s'appliquent terme à terme sur leurs opérandes.
- Des opérateurs spatiaux qui permettent de manipuler des domaines. Nous allons considérer, pour donner une brève description de ces opérateurs, la variable `x` définie sur le domaine `{i|i>=1}`.
  - L'opérateur de dépendance permet d'appliquer une fonction affine à chaque point d'un domaine. Ainsi, dans l'équation `y = x(i->i-1)`, `y` est une variable définie sur le domaine `{i|i>=2}`, telle que $\forall i \geq 2$, $y_i = x_{i-1}$.

- L'opérateur de restriction permet d'extraire une partie d'un domaine. Ainsi, dans l'équation `y = {i|i>=3}:x`, y est une variable définie sur le domaine `{i|i>=3}`, telle que $\forall i \geq 3$, $y_i = x_i$.
- L'opérateur de composition permet de construire un domaine à partir de parties de plusieurs autres. Ainsi, dans l'équation

```
y = case
      {i|i=1} : x
      {i|i>1} : x(i->i-1}
    esac
```

y est une variable définie sur le domaine `{i|i>=1}`, telle que :
$y_1 = x_1$
$\forall i \geq 2$, $y_i = x_{i-1}$.

Un compteur 4 bits s'écrit donc en ALPHA :

```
system Counter (count: {tt>=0} of bool) returns
  (value: {t,i|t>=0, 3>=i>=0} of bool; carry: bool);
var cout, cin: {t,i|t>=0, 3>=i>=0} of bool;
let
   cin = case
            {t,i|i=0}: count(t,i-> t);
            {t,i|i>0}: cout(t,i-> i-1);
         esac;
   (value, cout) = Cell(cin);
   carry = cout (t -> t, 3);
tel;
```

L'expression `count(i->)` étend à $\mathbb{Z}$ la variable `count` définie sur $\mathbb{Z}^0$. `cout (-> 3)` réalise l'opération inverse.

L'approche développée en ALPHA pour manipuler les tableaux, est très intéressante. Elle a pour avantages d'être simple, souple à utiliser, mais également parfaitement adaptée à un langage déclaratif. Appliquée à LUSTRE, elle apporte une réponse à la plupart des objections faites à l'extension précédente.

Le seul point non résolu est la définition exacte des opérateurs à introduire en LUSTRE pour manipuler des données structurées. Les domaines d'applications d'ALPHA et de LUSTRE ne sont pas les mêmes, les opérateurs utilisés dans le premier ne sont par conséquent pas parfaitement adaptés au second. ALPHA fournit des opérateurs de manipulation de domaines spatiaux généraux, l'objectif étant de fournir les moyens de réaliser simplement les transformations de programmes à la base de la plupart des méthodes de systolisation d'algorithmes [P.Q84] [Che88] [Mon85].

Nous désirons disposer en LUSTRE d'opérateurs plus simples, l'objectif étant dans ce cas la compilation efficaces de programmes. Les algorithmes de calcul de polyèdres [Che65] [FP88], utilisés lors de la manipulation de programmes ALPHA, sont trop coûteux pour être utilisés dans la compilation de programmes LUSTRE de grande taille.

2.2.2 Lucid

LUCID [AW85] est un langage à flots de données qui a été la principale source d'inspiration des concepteurs de LUSTRE. La solution choisie en LUCID pour représenter des données complexes est proche de celle employée en ALPHA. Une différence existe cependant au niveau de la distinction entre les dimensions spatiales et temporelles. En ALPHA, toutes les dimensions d'un domaine sont indifférenciées. La notion de temps n'est introduite que lors de l'ultime étape de systolisation d'un programme, lors de la recherche d'un ordonnancement des calculs. Une dimension de chaque domaine est alors choisie pour représenter le temps.

En LUCID en revanche, la dimension temporelle est particularisée dès le début. Une variable `X` est une fonction de $\mathbb{N} \times \mathbb{N}^n$ vers un ensemble de valeurs. Le premier indice est le paramètre temporel, les n suivants les paramètres spatiaux, La valeur de `X` à l'instant t, au point de coordonnées i_0, i_1, i_2 ... est notée $\mathtt{X}_t^{i_0,i_1,i_2\cdots}$.

Cette différenciation se retrouve également au niveau des opérateurs. A la base les manipulations spatiales ou temporelles d'un objet permises sont les mêmes ; elle sont cependant décrites à l'aide de deux jeux d'opérateurs distincts.

Une première catégorie d'opérateurs permet en LUCID de manipuler la dimension temporelle des variables :

- `first X` a pour résultat la valeur `X` au premier instant :
  $(\mathtt{first\ X})^{i_0,i_1,i_2\cdots} = \mathtt{X}_0^{i_0,i_1,i_2\cdots}$,
- L'expression `next X` a pour résultat la valeur `X` à l'instant suivant :
  $(\mathtt{next\ X})_t^{i_0,i_1,i_2\cdots} = \mathtt{X}_{t+1}^{i_0,i_1,i_2\cdots}$,
- L'expression `X fby Y` a pour résultat la valeur `Y` à l'instant précédent et la valeur courante de `X` au premier instant :
  $(\mathtt{X\ fby\ Y})_t^{i_0,i_1,i_2\cdots} = \mathtt{X}_0^{i_0,i_1,i_2\cdots}$, si t = 0
  $(\mathtt{X\ fby\ Y})_t^{i_0,i_1,i_2\cdots} = \mathtt{Y}_{t-1}^{i_0,i_1,i_2\cdots}$, si t > 0

Les dimensions spatiales sont manipulées de façon similaire à la dimension temporelle grâce à une seconde catégorie d'opérateurs.

- La définition de `initial X` est analogue à celle de `first X` :
  $(\mathtt{initial\ X})_t^{i_0,i_1,i_2\cdots} = \mathtt{X}_t^{0,i_0,i_1,i_2\cdots}$,
- La définition de `rest X` est analogue à celle de `next X` :
  $(\mathtt{rest\ X})_t^{i_0,i_1,i_2\cdots} = \mathtt{X}_t^{i_0+1,i_1,i_2\cdots}$,
- La définition de `X cby Y` est analogue à celle de `X fby Y`
  $(\mathtt{X\ cby\ Y})_t^{i_0,i_1,i_2\cdots} = \mathtt{X}_t^{i_1,i_2\cdots}$, si t = 0
  $(\mathtt{X\ fby\ Y})_t^{i_0,i_1,i_2\cdots} = \mathtt{Y}_t^{i_0-1,i_1,i_2\cdots}$, si t > 0

Avec ces primitives le compteur s'écrit ainsi :

```
Counter (count) = (value, carry)
where
   (value, cout) = Cell(cin);
   cin = count cby cout;
   carry = Elem (3, cout);
end;

Elem (n, array) = e
where
   e = if n = 0 then initial array
       else Elem(n-1, rest array);
end;
```

La solution choisie par LUCID est simple et séduisante, mais également très restrictive. Les inconvénients et les avantages d'utiliser ce formalisme en LUSTRE sont inversés par rapport à ceux donnés au sujet de ALPHA. L'accès uniquement séquentiel aux éléments d'une variable réduit son utilisation à une classe restreinte d'applications très régulières. Comment par exemple, décrire simplement l'algorithme de calcul de la transformée de Fourier rapide ?

2.2.3 μFP

Plusieurs langages, dérivés de FP [J.B78], ont été développés et utilisés dans le domaine de la description de circuits comme μFP [She85] [She84], auquel nous allons plus particulièrement nous intéresser, ou νFP [PSE85]. Le domaine d'application visé par μFP est la description de circuits très réguliers, qui se présentent sous la forme de matrices de processeurs. L'originalité de l'approche vient du fait que chaque opérateur du langage est doté d'une double sémantique, un programme FP ne décrit pas uniquement le comportement d'une application mais également sa topologie.

Les primitives les plus simples et leur sémantique sont décrites Figure 2.1 et Figure 2.2 .

Le compteur à partir de tels constructeurs, s'écrit d'une manière remarquablement concise :

```
<value0, value1, value2, value3, carry> = /R Cell : <count>
```

Suivant l'usage qui est fait de la description d'un circuit, la spécification de son seul comportement peut se révéler insuffisante. Par exemple, des informations, même partielles, de placement peuvent permettre d'obtenir une meilleure implantation. Des placeurs automatiques peuvent être employés dans ce cas mais :

- pourquoi utiliser des outils relativement lents pour essayer de retrouver des informations à la disposition du concepteur.
- ces outils sont en général basés sur des méthodes heuristiques, qui n'assurent pas de d'obtenir le résultat souhaité.

Mais les informations peuvent être de toute nature, temps de calcul, longueur de connexion, couleur, ... Il semble donc judicieux de ne pas se limiter en LUSTRE, comme c'est le cas pour μFP, aux informations de placement.

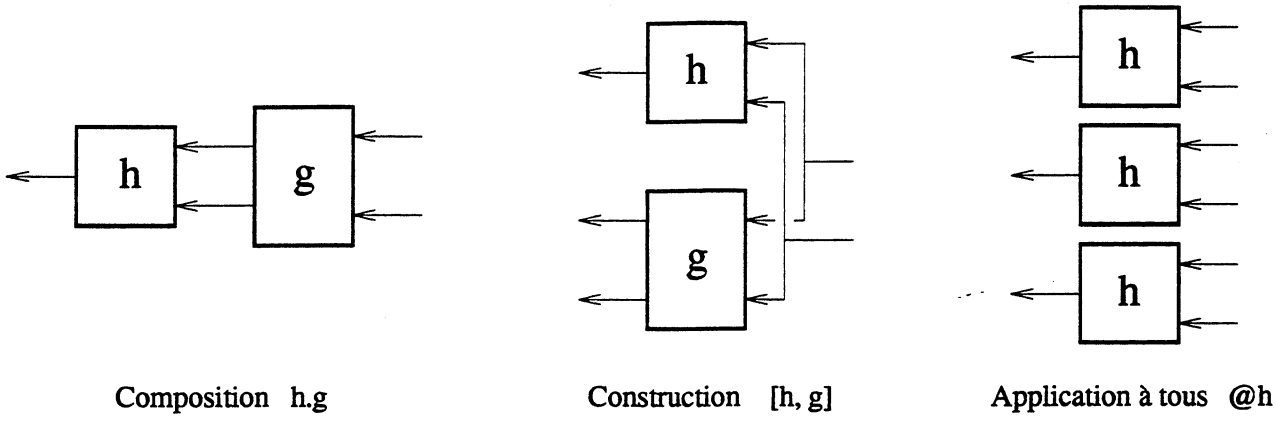
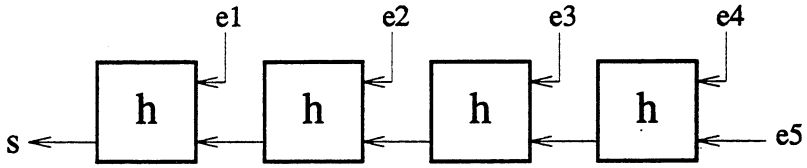
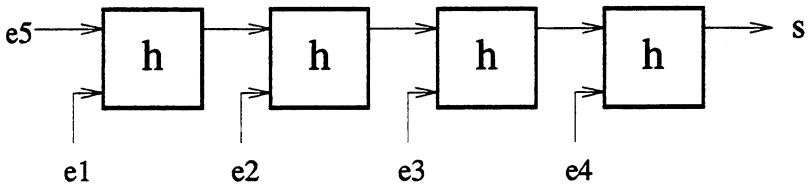


Figure 2.1 *Une première série d'opérateurs de μFP*



Chainage en partant de la droite s = /R h : <e1, e2, e3, e4, e5>



Chainage en partant de la gauche s = /L h : <e1, e2, e3, e4, e5>

Figure 2.2 *Une seconde série d'opérateurs de μFP*

Nous désirons décrire en LUSTRE une classe d'opérateurs plus générale qu'en μFP. Or, dès qu'un circuit n'est plus parfaitement régulier il est beaucoup plus avantageux de décrire sont comportement, le mettre au point et seulement ensuite de rajouter les informations supplémentaires. Pour cette raison nous ne suivrons pas l'exemple de μFP et ne donnerons pas plusieurs sémantiques distinctes aux opérateurs.

2.2.4 APL

APL [Leg79] est sans nul doute le langage le plus complet au niveau des manipulations de tableaux. Le fait que beaucoup d'opérateurs soient représentés à l'aide de caractères spéciaux, imposant l'utilisation d'APL sur du matériel spécialisé a limité son utilisation mais il reste une référence incontournable. Plus récemment, Le langage 81/2 [J.L88], reprenant la même philosophie, a été développé pour programmer des applications sur machines massivement parallèles,

Nous ne décrirons pas, comme nous l'avons fait jusqu'à présent, le compteur 4 bits en APL . Le langage étant relativement ancien, il est un peut frustre par certains aspects. Les fonctions définies, notamment, ne peuvent avoir plus d'un résultat. Cet exemple n'aurait donc pas été simple à décrire et n'aurait de tout façon pas été très représentatif des possibilités offertes par le langage.

Nous allons donc nous contenter de donner un bref aperçu des principales possibilités offertes par APL. Afin d'éviter toute confusion par la suite le terme :

opérateur : désignera une primitive du langage permettant de manipuler des tableaux.

fonction : désignera, un objet primitif ou défini par l'utilisateur permettant de faire un calcul, par exemple la fonction d'addition notée `+`.

Dans les exemples d'illustration nous utiliserons deux variables :

- `A` correspondant au tableau
```
2  3  ¯4  5
1  3   3  4
```
(`¯` désigne le moins unaire)

- `B` correspondant au tableau
```
1   0  2  1
0  ¯1  4  7
```

Les fonctions s'appliquent terme à terme à des données de même dimensions : `A + B` par exemple, a pour résultat le tableau
```
3  3  ¯2  6
1  2   7  11
```

Les fonctions s'appliquent également entre un scalaire et un tableau, le résultat a la dimension du tableau : `A + 2` par exemple, a pour résultat le tableau
```
4  5  ¯2  7
3  5   5  6
```

Différentes fonctionnelles permettent d'autres types d'application des fonctions. En voici quelques unes parmi les plus simples :

- La réduction / applique une fonction à tous les éléments d'un tableau. Par exemple `+/ 2 3 ¯4 5` a pour résultat `2 + 3 + ¯4 + 5 = 6`.

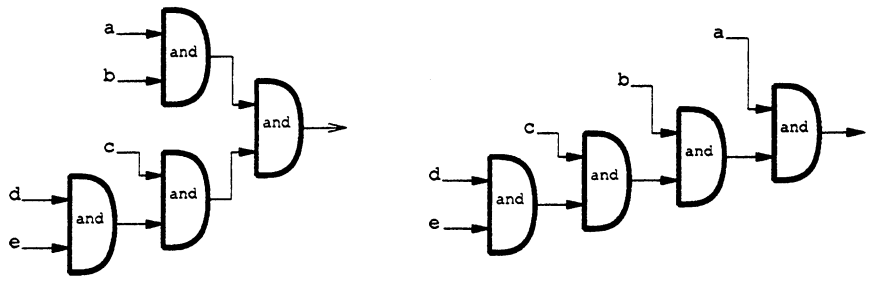


Figure 2.3 *Implantation de l'opérateur de réduction*

- Le balayage produit un tableau de même taille dont la valeur du i^eme élément est la réduction des i premiers éléments du tableau source. Par exemple `+ 2 3 ¯4 5` a pour résultat `2 5 1 6`.

Ces fonctionnelles s'appliquent aussi à des tableaux multidimensionnels, mais il faut indiquer entre crochet après l'opérateur la dimension affectée. Par exemple `+/[1] A` calcule la somme des lignes de `A` le résultat est `6`
`11`

Les opérateurs de manipulation de tableaux sont également nombreux et variés. En voici quelques exemples :

- La sélection d'un partie d'un tableau se fait en indiquant entre crochets, pour chaque dimension, l'ensemble des indices choisis. Les éléments sélectionnés sont ceux appartenant au produit cartésien de ces ensembles par exemple, `A[1 ; 2 3]` sélectionne les deuxièmes et troisièmes éléments de la première ligne de `A` le résultat est `3 ¯2`.

- L'opérateur de linéarisation, met à plat un tableau multidimensionnel. Par exemple le résultat de `,A` est `2 3 ¯4 5 1 3 3 4`.

- De nombreux opérateurs permettent de faire des permutations d'indices, des déplacements d'éléments. Par exemple, l'expressions `1` ϕ`[1]A` effectue une rotation d'un élément vers la droite de chaque ligne de `A` le résultat est `5 2 3 ¯4`
`4 1 3 3`

APL présente l'intérêt de fournir un catalogue très complet d'opérateurs de manipulation de tableaux, fournissant des indications précieuses sur ceux à ajouter en LUSTRE. Tous ne sont pas adaptés à ce langage ou à la description de circuits. La réduction d'une fonction, par exemple, présente l'inconvénient de ne pas avoir une sémantique suffisamment précise. Plusieurs choix de circuits correspondant à l'expression `∧/ a b c d e` se présentent, deux sont représentées Figure 2.3

De manière générale, nous nous efforcerons d'ajouter à LUSTRE des opérateur ne laissant pas de choix d'implantation au compilateur, de façon à permettre une description de circuit la plus précise possible.

2.3 Langages de description de matériel

2.3.1 Silage

SILAGE [Hil87] [HRG+90] est un langage de description d'algorithmes de traitement du signal utilisé en entrée des compilateurs de silicium Cathedral [FC89].

Le langage est très proche de LUSTRE :

- il est basé sur le modèle flot de données. Les opérateurs manipulent des suites infinies de valeurs.
- un programme est décrit par un système d'équations.
- la notation de tableaux employée est similaire à celle initialement proposée pour LUSTRE.

Cette parenté des deux langages est illustrée par la description en SILAGE du compteur :

```
func Cell (count:bool) : value, carry: bool =
begin
   value@@1 = 0;
   value = value@1 != count;
   carry = !value & count;
end;

func Counter (count:bool) : value: bool[4]; carry:bool =
begin
   (value[0], carry[0]) = Cell(count);
   (i: 1 .. 2)::(value[i], carry[i+1]) = Cell(carry[i]);
   (value[3], carry) = Cell(carry[3]);
end;
```

`value@1` est l'équivalent de l'expression LUSTRE `pre value`, l'équation `value@@1 = 0` définit la valeur initiale de value. Les déclarations des variables locales sont implicites.

Un programme SILAGE décrit le comportement d'un circuit indépendamment de l'architecture sur lequel il doit être implanté. Cependant un concepteur peut également à l'aide de la directive `pragma`, fournir des informations supplémentaires au compilateur. La directive `pragma Processor (n, e1, e2, ...)` par exemple, permet d'indiquer que les expressions `e1`, `e2`, ... doivent être calculées sur le processeur n.

2.3.2 Model

MODEL [GBR83] [GBR82] est un autre langage utilisé comme point d'entrée d'un compilateur de silicium [GH85]. Ce langage est de plus bas niveau que SILAGE, un programme MODEL étant principalement constitué d'un assemblage d'éléments en provenance d'une librairie de composants de bas niveaux, localement optimisés. Les circuits décrits sont donc beaucoup plus liés à une technologie donnée.

La description d'un circuit est simplifiée en MODEL grâce à deux constructions. Il est ainsi possible de :

- construire des vecteurs de fils. Par exemple `x(0:7)` définit un vecteur de 8 fils indicés de 0 à 7 et nommé `x`.
- décrire un circuit de façon hiérarchique en définissant de nouveaux composants (appelés `Part`). Ces derniers peuvent être définis en fonction de paramètres entiers, ce qui en permet une utilisation plus générale. Par exemple,

  `Part Counter (n) [clock, count] -> value(0:n-1), carry`

  est l'entête d'un compteur n bits (`clock` est l'horloge globale commandant les points mémoires.

Voici la description MODEL du compteur 4 bits

```
Part  Cell [clock, count] -> value, carry
   xor [bdff[clock, value], count] -> value, --
   and [not [value], count] -> carry
End

Part Counter [clock, count] ->  value(0:3), carry
Signal inter(0:3)
   for i = 0:3
      Cell [If i=0 Then count Else inter(i-1) Endif] -> value(i), inter(i)
   inter(3) -> carry
End;
```

`bdff` est un élément de librairie, une bascule D ayant :

- deux entrées, un signal d'horloge et la valeur à mémoriser, à chaque front montant de l'horloge.
- deux sorties, son état courant et celui complémenté.

2.3.3 VHDL

VHDL [SLM+85] [Ash90] s'impose de plus en plus comme le standard des langages de description de circuits, ce qui était un des principaux objectifs de son développement. Ce langage a été conçu dans le but de rendre la conception d'un circuit indépendante de la technologie employée afin de permettre une description de plus haut niveau ou de faciliter la réutilisation de composants. Toutefois, là encore, la possibilité a été laissée aux utilisateurs de rajouter des informations (les attributs) qui seront utilisées pour des traitements spécifiques.

Un programme VHDL se décompose en un ensemble d'entités (`entity`) représentant chacune un circuit ou une partie de circuit. Une entité se décompose en :

- une interface décrivant les entrées et les sorties d'un circuit, mais aussi les paramètres génériques, permettant de fixer certaines de ses caractéristiques (technologie employée, taille, ...) uniquement lors de son instanciation.
- un ensemble de descriptions du circuit pouvant être :

  comportementale: définissant la fonction réalisée par le circuit

  architecturale: définissant la structure du circuit.

Nous donnons ci-dessous une description architecturale d'une cellule de compteur :

```
entity Cell is
port (count: in bit; value, carry: out bit);
end Cell;

architecture structure of Cell is
   signal oldvalue:bit;
begin
   oldvalue := value;
value <= oldvalue xor count;
carry <= not value and count;
end structure;
```

La philosophie adoptée par VHDL est à l'opposé de celle de LUSTRE, l'objectif a été de concevoir un langage qui soit le plus complet possible, mais sans prendre le soin d'en donner une sémantique précise. Des problèmes d'utilisation de VHDL sont apparus lorsqu'il a commencé à être utilisé dans le domaine de la synthèse et non plus simplement à celui de la simulation auquel il était originellement plus particulièrement destiné. En effet, certaines parties de VHDL, les pointeurs par exemple, ne sont pas synthétisables. Un gros effort est donc réalisé actuellement en vue de définir un sous ensemble standard en entrée d'outils de synthèse.

Contrairement à LUSTRE, aucune hypothèse de synchronisme n'est faite, la prise en compte des temps de propagation des signaux est prise en compte en définissant un modèle comportant une double échelle de temps :

- Un micro instant correspond à un délai de propagation, il n'est pas mesurable par l'utilisateur
- Un macro-instant correspond à une unité mesurable de temps, il est composé d'un ensemble de micro-instants

Plusieurs autres langages peuvent être rangés dans la catégorie des langages de description comportementale/structurelle de circuits auquel appartient VHDL. Parmi les plus proches citons pour mémoire ZEUS [GL85] CONLAN [Pa85].

Chapitre 3

Lustre V4

3.1 Expressions statiques

Les expressions LUSTRE se divisent en trois catégories :

- La plus générale est celle des expressions dont la suite de valeurs n'est calculée qu'à l'exécution.
- La seconde est celle des expressions constantes dont tous les éléments de la suite de valeurs sont identiques. De telles expressions ne sont pas toujours évaluables au moment de la compilation car elles peuvent être définies à partir de constantes externes. Un identificateur de constante est défini en précédant sa déclaration du mot clé `const`. Ce peut être une déclaration de paramètre, comme en LUSTRE V3, mais également une déclaration locale à un noeud. La déclaration `const x:t` définit une constante `x` de type `t`.
- Les expressions statiques, enfin, dont le résultat est une suite de valeurs constantes qui est évaluable au moment de la compilation. Un identificateur de statique est défini de la même façon qu'un identificateur de constante. L'utilité de ces expressions apparaîtra au fur et à mesure de la description des extensions apportées au langage.

Ces catégories sont incluses les unes dans les autres. Une expression constante peut être utilisée comme une expression ayant une suite de valeurs variables, une expression statique comme une expression constante ou variable. Le contraire en revanche n'est pas vrai.

3.2 Les types

3.2.1 Définitions de types

Peu de changement au niveau des types élémentaires qui se composent :

- Des types prédéfinis, qui ont été évoqués lors de la présentation du langage de base (§ 1.4), complétés du type caractère `char`. Une constante caractère est décrite, tout comme en C, par un symbole encadré de quotes.

- Des types externes qui sont définis dans un autre langage et importés, avec leurs opérateurs et leurs constantes associés (§1.8).

Le type caractère était jusqu'à présent un type externe. Toute constante caractère était un identificateur qu'il fallait déclarer explicitement, ce qui rendait leur manipulation lourde et malaisée. Le type caractère est donc devenu prédéfini pour les mêmes raisons que les autres, booléen, entier ou réel : simplifier l'écriture des programmes et fournir une notation plus habituelle pour les constantes.

A partir de ces types élémentaires, de nouveaux types peuvent être dérivés au moyen de deux constructeurs :

- Si `t1`, `t2`, ... `tn` sont n types, `[l1 : t1, l2 : t2 ..., ln : tn]` dénote le type d'une structure de n éléments dont le k^{ieme}, repéré par le label `lk`, est de type `tk`. Tous les labels d'une structure doivent être différents. L'ordre de déclaration des champs d'une structure est sans importance, ils sont repérés uniquement par leur label. Ainsi, les types `[l1 : t1, l2 : t2 ..., ln : tn]` et `[l2 : t2, l1 : t1 ..., ln : tn]` sont identiques.

- Si `t` est un type et si `e1`, `e2`, ... `en` sont n expressions statiques entières, `t^(e1, e2, ... , en)` dénote le type d'un tableau de n dimensions, d'éléments de type `t`. Chaque `ek` est une expression statique qui spécifie la taille de la k^{ieme} dimension.

Ces deux constructeurs peuvent être combinés à volonté. Par exemple,

- `bool^4^7` dénote le type un tableau de tableaux de booléens Le parenthésage implicite est `(bool^4)^7`, il s'agit du type *"tableau de 7 tableaux de 4 booléens"*.

- `bool^(4,7)` quant à lui, dénote le type tableau de booléens à deux dimensions, la première de taille 4, la seconde de taille 7, différent du précédent.

- `[a : bool^3, b : [ba : bool, bb : int] ]` définit le type d'une structure dont le champ de label `a` est un tableau de 3 booléens et l'autre, de label `b`, une structure ayant elle-même deux champs de labels `ba` et `bb`, de types respectifs `bool` et `int`.

Un type est défini par une équation

```
x = t
```

où `x` est un identificateur et `t` un type construit à partir des types élémentaires, d'autres types définis par l'utilisateur et des deux constructeurs. Par exemple :

- `A = [first: int, second: bool]` définit que `A` dénote le type d'une structure possédant deux champs, l'un nommé `first`, de type `int`, l'autre, nommé `second`, est de type `bool`.

- `B = A^7` définit que `B` dénote le type d'un tableau de 7 éléments de type `A`.

3.2.2 Equivalence de types

La relation d'équivalence entre les types est purement structurelle, le nommage de types est juste une facilité syntaxique. Ce choix nous a semble plus simple et le plus adapté

Deux types sont donc dits équivalents (ce qui sera noté `t1` $\sim$ `t2`) :

- Si `t1` et `t2` désignent le même type élémentaire
- S'ils dénotent les types de deux tableaux de même dimension et tailles, dont les éléments sont des types équivalents :

  `t = t1^(e1,e2, ..., en)`

  `u = u1^( f1, f2, ..., fk)`

  `t1` $\sim$ `u1`

  $\forall$ `k`, val(`ek`) = val(`fk`)

- S'ils dénotent les types de deux structures de même taille, ayant le même ensemble de labels et dont les éléments ayant même label sont de types équivalents :

  `t = [i1:t1, i2:t2, ... , in:tn]`,

  `u = [j1:u1, j2:u2, ... , jn:un]`

  $\forall k,\ 1 \leq i \leq n\ \exists l,\ 1 \leq l \leq n$ tel que `ik` = `jl` et `tk` $\sim$ `ul`.

3.2.3 Conversions de type

Aucune conversion de type implicite n'est autorisée en LUSTRE toutes doivent être explicites. A cet effet, deux fonctions prédéfinies permettent de faire les conversions d'entiers en réels et vice-versa.

- la fonction `int` prend un réel comme paramètre d'entrée et fournit en sortie sa partie entière convertie en entier.
- la fonction `real` prend un entier en entrée et le convertit en réel.

3.2.4 Surcharge

Les deux constantes entières `0` et `1` peuvent être utilisées dans des expressions booléennes respectivement en lieu et place des constantes `false` et `true`.

3.3 Décomposition modulaire

Un noeud intermédiaire d'un programme étant lui même un programme, il peut être compilé seul avec les noeuds qu'il utilise et exécuté. La hiérarchie permet donc de simplifier non seulement la description d'un programme mais aussi sa mise au point qui peut être réalisée opérateur par opérateur.

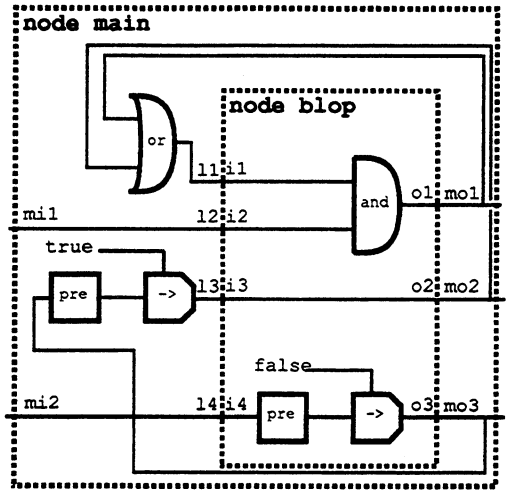


Figure 3.1 *Schéma du programme main*

Il est également intéressant de pouvoir compiler séparément les différentes parties d'un programme, cela évite notamment, lors d'une modification, d'en recompiler l'ensemble. Mais, pour compiler séparément un noeud, il est nécessaire de fournir des informations supplémentaires dans son entête, il s'agit :

- Des dépendances entrées-sorties, c'est à dire, pour chaque sortie, des entrées dont elle dépend sans retard (une variable x dépend sans retard d'une autre variable y s'il est possible que x_i ait besoin de y_i pour être calculée).
- Des dépendances sorties-entrées, c'est à dire, pour chaque entrée, des sorties dont elle peut dépendre sans retard.

Ces dépendances, que l'on retrouve dans SIGNAL [LGLL91], sont nécessaires :

- Pour déterminer si le programme est correct, si le calcul de la valeur d'une variable ne dépend pas d'elle même.
- Pour trouver un ordre d'évaluation des calculs lorsque le programme est séquentialisé.

Par exemple, dans l'exemple suivant :

```
node blop (i1,i2,i3,i4:bool) returns (o1,o2,o3:bool);
let
   o1 = i1 and i2;
   o2 = i3;
   o3 = false -> pre i4;
end;
```

```
node main (mi1,mi2:bool) returns (mo1,mo2,mo3:bool)
var
   l1,l2,l3,l4:bool;
let
  (mo1,mo2,mo3) = blop(l1,l2,l3,l4);
   l1 = mo1 or mo2;
   l2 = mi1;
   l3 = true->pre mo3;
   l4 = mi2;
end;
```

- o1 dépend sans retard de i1 et i2
- o2 dépend sans retard de i3
- i1 dépend sans retard de o1 et o2
- i3 dépend sans retard de o3
- les autres entrées sorties du noeud `blop` n'ont pas de dépendance sans retard.

La déclaration de ce noeud avec ses dépendances est :

```
node blop (i1,i2,i3,i4:bool) returns (o1,o2,o3:bool);
dep
   o1 needs i1,i2;
   o2 needs i3;
   i1 needs o1,o2;
   i3 needs o3;
let
   o1 = i1 and i2;
   o2 = i3;
   o3 = false -> pre i4;
end;
```

Ce noeud peut ainsi être compilé séparément du noeud principal et ce dernier peut l'être également en ne connaissant que l'entête du noeud blop.

```
node blop (i1,i2,i3,i4:bool) returns (o1,o2,o3:bool);
dep
   o1 needs i1,i2;
   o2 needs i3;
   i1 needs o1,o2;
   i3 needs o3;
end;
```

```
node main (mi1,mi2:bool) returns (mo1,mo2,mo3:bool)
var
   l1,l2,l3,l4:bool;
let
  (mo1,mo2,mo3) = blop(l1,l2,l3,l4);
   l1 = mo1 or mo2;
   l2 = mi1;
   l3 = true->pre mo3;
   l4 = mi2;
end;
```

Ainsi les deux parties du programme sont à même de détecter que le programme est incorrect puisqu'il y a une boucle combinatoire entre la première entrée du noeud **blop** et sa première sortie.

Notons au passage une modification de la syntaxe du langage. le mot clé **tel** a été remplacé par **end**. En LUSTRE V3, une déclaration de noeud se compose d'un entête puis de deux blocs facultatifs :

- Un ensemble de déclarations de variables débutant par le mot clé **var**
- Un système d'équations encadré par les mot clés **let** et **tel**. Celui-ci est omis dans la déclaration d'un noeud externe.

En LUSTRE V4, une déclaration de noeud commence par le mot clé **node** et se termine au mot clé **end**. La déclaration débute par l'entête du noeud suivi de trois blocs facultatifs :

- Un ensemble de déclarations de dépendances débutant par le mot clé **dep**
- Un ensemble de déclarations de variables débutant par le mot clé **var**
- Un système d'équations débutant par le mot clé **let**.

3.4 Notation de listes d'expressions

Une liste d'expressions est formée à l'aide de n expressions séparées par des virgules et, si le contexte l'impose, entourée par des parenthèses. A la différence des structures, les listes ne sont pas hiérarchiques. Ainsi, étant donné n expressions **e1**, **e2**, ... , **en**, Les listes d'expressions **((e1, e2), e3, ... , en)** et **(e1, e2, e3, ... , en)** sont identiques.

3.5 Structures et tableaux

Un petit nombre d'opérateurs ont été introduits pour permettre de définir ou manipuler structures et tableaux

3.5.1 Opérateurs de construction

Les deux premiers opérateurs sont des constructeurs, il permettent de définir des structures ou des tableaux.

La structure ou le tableau vide à 0 élément sont représentés par l'expression `[]` (crochet-ouvrant, crochet fermant).

3.5.1.1 Structures

Etant donné une liste de n expressions `e1, e2, ..., en` de types respectifs `t1, t2, ... , tn`, l'expression `[l1: e1, l2: e2, ..., ln: en]` définit une structure de type `[l1: t1, l2: t2, ..., ln: tn]` dont le champ de label `lk`, de type `tk`, est `ek`.

3.5.1.2 Tableaux

Un tableau peut être construit de plusieurs façons :

- Etant donné une liste de n expressions statiques `e1, e2, ..., en` de type `int` et `e` une expression de type `t`, l'expression `e^(e1, e2 , ... , en)` définit un tableau de type `t^(e1, e2 , ... , en)`, dont chacun des éléments est un copie de `e`.
- Etant donné une liste de n expressions `e1, e2, ..., en` de type `t`, l'expression `[e1, e2, ..., en]` définit un tableau de type `t^n` dont le k^{ieme} élément est `ek`.
- Etant données trois expressions statiques `e1`, `e2` et `e3` de type `int` et ayant pour valeurs respectives à tout instant, i, j et k l'expression `e1 .. e2 step e3` définit le tableau formé des entiers $i + k \times x$ compris entre i et j, $x \geq 0$.
  - `[e1]` si $i = j$
  - `[e1, e1 + e3, ... ]` si $i < j$ et $k > 0$
  - `[e1, e1 - e3, ... ]` si $j < i$ et $k < 0$
  - Le tableau vide `[]` sinon, puisque l'intervalle de sélection est vide.

  Dans le cas ou k est égal à 1, la partie `step` peut être omise Par exemple,
  - `1 .. 4` définit le tableau de constantes entières `[1, 2, 3, 4]` de type `int^4`
  - `2 .. 6 step 3` définit le tableau de constantes entières `[2, 5]` de type `int^2`
  - `4 .. 2` définit le tableau vide `[]`

3.5.2 Opérateurs de sélection

3.5.2.1 Structures

Une structure peut être manipulée de deux façons :

- Dans son ensemble : Si `s` est une structure et `e` une expression de type équivalent, l'équation `s = e` est valide, elle définit qu'à chaque instant, la valeur de chaque élément de `s` est égale à la valeur de l'élément ayant même label de `e`.
- Composant par composant. La sélection d'un élément d'une structure se fait en la suffixant par le label du champ sélectionné. Si `s` est une structure de type `[l1 : t1, l2 : t2 ..., ln : tn]`, l'expression `s.l1`, permet de sélectionner le champs de `s` ayant pour label `l1`, de type `t1`, `s.l2` celui ayant pour label `l2`, de type `t2` ...

Si, par exemple, `s` est égale à `[a:e1, b:e2, c:e3 , d:e4]`, le résultat de

`s.a` est égal à `e1`

3.5.2.2 Tableaux

Les manipulations permises sur les tableaux sont plus étendues que celles qui viennent d'être décrites pour les structures. En effet il est possible d'utiliser un tableau dans son ensemble, d'en sélectionner un élément, mais également d'en extraire un sous-tableau.

Intéressons-nous tout d'abord uniquement aux tableaux mono-dimensionnels. Les éléments d'un tableau n'étant pas nommés, ils sont repérés par leur position. Le premier est indicé `0`, le second `1`, etc. Ce choix nous a semblé être, pour LUSTRE, le plus judicieux de ceux envisagés. C'est celui fait dans de nombreux domaines, en description de matériel notamment. Toute nappe de fils, vecteur de bits est numéroté à partir de 0.

Pour que cette numérotation reste cohérente, tout tableau produit par un calcul est implicitement réindicé pour que ses éléments soient dans le bon intervalle. Par exemple, soit un tableau `[e0, e1, e2, e3, e4]`, de type `t^5`, dont est extrait le sous-tableau `[e2,e4]`. Dans ce dernier les éléments `e2` et `e4` ont pour indices respectifs 0 et 1 et non 2 et 4.

Ce choix présente un certain nombre d'avantages :

- Les tableaux manipulés sont tous compacts.
- La gestion des indices est simplifiée, il n'y a notamment pas à se poser la question de savoir comment combiner deux tableaux de même nature mais définis sur des intervalles différents.

Etant donné un tableau de n éléments `a` de type `t^n` et `e` une expression statique, l'accès à une partie de ses éléments est réalisé par une expression de la forme `a[e]` .

- Si `e` est de type `int`, le résultat de l'expression `a[e]` est le $(e+1)^{ieme}$ élément de `a`, *e* devant être compris entre *0* et *n - 1*.
- Si `e` est de type `int^m`, le résultat de l'expression `a[e]` est un tableau de m éléments `[s[e[0]], s[e[1]], ... , s[e[m -1]]` de type `t^m`.

Comme exemple, considérons le tableau a égal à `[e0, e1, e2 , e3, e4, e5, e6]`, le résultat de :

- `a[0]` est l'élément `e0`
- `a[0 .. 3]` ou `a[ [0, 1 ,2, 3] ]` est le tableau `[e0, e1, e2 ,e3]`
- `a[4 .. 1 step -2]` ou `a[ [4,2] ]` est le tableau `[e4,e2]`

Il est important de noter que la sélection d'un élément et la sélection d'un sous-tableau ne comprenant qu'un seul élément n'ont pas le même résultat. Si le $(i+1)^{ieme}$ élément de `a`, `a[i]` est de type `t`, le résultat de l'expression `a[i .. i]` est le tableau `[a[i] ]` de type `t^1`.

Ainsi, si l'on reprend l'exemple précédent.

- `a[3]` sélectionne l'élément `e3`
- `a[3 .. 3]` ou `a[ [3] ]` sélectionnent le tableau `[e3]`

Il est également possible de ne sélectionner aucun élément. Le résultat de l'expression `a[[]]` a pour résultat le tableau vide `[]`.

Considérons maintenant un tableau multi-dimensionnel `a` de type `t^(n1, n2, ..., nk)`. Chacun de ses éléments est repéré par un k-uplet d'indices. L'accès à une partie de ses éléments est réalisé par une expression `a[e1, e2, ..., ek]` où `e1`, `e2`, ... `ek` sont k expressions statiques.

Chaque expression `ei` est soit un entier, soit un tableau d'entiers. Elle définit un ensemble de valeurs, les éléments sélectionnés sont uniquement ceux dont le i^{ieme} indice appartient à cet ensemble. Les éléments extraits du tableau sont donc ceux dont les k-uplets d'indices les repérant, appartiennent au produit cartésien de ces ensembles.

Le résultat est un tableau dont la dimension est égale au nombre d'expressions `ei` qui sont des tableaux d'entiers. Soit q un entier inférieur ou égal à n et *i1, i2, ..., iq* une suite croissante de nombres compris entre *1* et n, tel que `ei1`, `ei2`, ... `eiq` sont de type respectifs `int^mi1`, `int^mi2`, ... , `int^miq` et toutes les autres expressions `ei` sont de type `int` ; alors le type du résultat de la sélection est `t^(mi1, mi2, ..., miq)`. Les deux cas extrêmes sont les suivants :

- Si tous les `ei` sont des entiers alors le résultat est l'élément de type `t` de `a` repéré par le n-uplet des valeurs des `ei` chacune comprise entre 0 et *ni*-1.
- Si chaque `ei` est un tableau d'entiers, le résultat est un tableau de type `t^(m1,m2,...,mk)`, l'élément du tableau résultat d'indices `i1, i2, ...` `ik` est l'élément `a[e1[i1] , e2[i2] , ... , ek[ik] ]`

Par exemple, si l'on considère le tableau à deux dimensions :

$$a = \begin{bmatrix} e00 & e01 & e02 & e03 & e04 \\ e10 & e11 & e12 & e13 & e14 \\ e20 & e21 & e22 & e23 & e24 \\ e30 & e31 & e32 & e33 & e34 \end{bmatrix}$$

- `a[2, 3]` sélectionne l'élément `e23`
- `a[0..1,1..3]` sélectionne le tableau bi-dimensionnel $\begin{bmatrix} e01 & e02 & e03 \\ e11 & e12 & e13 \end{bmatrix}$

- `a[1,2..4]` sélectionne le tableau mono-dimensionnel [`e12`, `e13`, `e14`]

Les opérateurs de sélection s'appliquent non seulement à des variables mais aussi à des expressions. Il est notamment possible, dans le cas de structures de structures, de tableaux de tableaux, ... d'appliquer successivement plusieurs sélections pour extraire les éléments désirés.

3.5.3 Opérateur de concaténation

Ce opérateur permet de définir une structure ou un tableau par juxtaposition de deux autres.

3.5.3.1 Structures

Etant donné deux structures ayant des ensembles disjoints de labels,

- `s0` de n champs, de type `[l1:t1, l2:t2 ..., ln:tn]`
- `s1` de m champs, de type `[k1:v1, k2:v2 ..., km:vm]`

l'expression `s0 | s1` définit une structure de n+m champs, de type
`[l1:t1, l2:t2 ..., ln:tn , k1:v1, k2:v2 ..., km:vm]`

Par exemple si la structure `s0` est égale à

```
[a:[aa:e0, ab:e1], b:e3, [c:e4] ]
```

et la structure `s1` à

```
[h:f0, [x:f1, [xx:f2, xxx:f3] ] ]
```

le résultat de l'expression `s0 | s1` est la structure

```
[a:[aa:e0, ab:e1], b:e3, [c:e4]   h:f0, [x:f1, [xx:f2, xxx:f3] ] ].
```

3.5.3.2 Tableaux

Etant donnés deux tableaux mono-dimensionnels ayant des éléments de même type,

- `a0` de type `t^n`
- `a1` de type `t^n`

l'expression `a0 | a1` définit une tableau de n+m éléments de type `t^(n+m)`

Par exemple si le tableau `a0` est égal à `[e0, e1, e2]` et le tableau `s1` à `[f0, f1]`, le résultat de l'expression `a0 | a1` est le tableau `[e0, e1, e2 , f0, f1]`

3.6 Polymorphisme

Les noeuds et la plupart des opérateurs prédéfinis ont une signature unique, c'est par exemple le cas de l'opérateur *"et logique"* (**and**) qui a pour signature $bool \times bool \rightarrow bool$. Dans le cas d'un noeud, sa signature est définie par son entête.

D'autre opérateurs sont surchargés, c'est le cas notamment de la plupart des opérateurs arithmétiques. Par exemple la signature de l'opérateur d'addition peut être $int \times int \rightarrow int$ ou alors $real \times real \rightarrow real$.

Enfin, les opérateurs restants sont polymorphes. Les types d'une partie, voire de l'ensemble, de leurs opérandes sont variables, ils sont définis pour chacune de leur instance par les types des expressions auxquels ils sont appliqués. Il s'agit, par exemple de `pre` ou `if`.

La façon de noter les types variables en LUSTRE est similaire à ML [Nik85], un identificateur de type variable est une suite d'étoiles. Deux types variables ayant même identificateur doivent forcément toujours désigner un même type. Par exemple, un opérateur dont la signature est $* \times * * \rightarrow *$ peut être appliqué à n'importe quel couple d'expressions de type élémentaire, prédéfinis ou importé, et produit un résultat dont le type est identique à celui de la première expression.

La notion de type variable définie, il est maintenant possible de donner la liste des opérateurs polymorphes :

- `if` dont la signature est $bool \times * \times * \rightarrow *$
- `pre` et `current` dont la signature est $* \rightarrow *$
- `->` dont la signature est $* \times * \rightarrow *$
- `when` dont la signature est $* \times bool \rightarrow *$
- $=$ et $<>$ dont la signature est $* \times * \rightarrow bool$

Les types variables permettent également la définition des noeuds polymorphes. Par exemple le noeud Reg définit un opérateur de retard avec initialisation :

```
node Reg (init, in: *) returns (out: *);
let
   out = init -> pre in;
tel;
```

Une expression ayant un type variable ne peut être l'opérande que d'un opérateur polymorphe, si `x` et `y` sont deux variables de types `*`, l'expression `x or y` est incorrecte. De même, si `x` et `y` sont deux variables de types respectifs `*` et `**`, l'expression `x -> y` est incorrecte.

3.7 Extension homomorphe des opérateurs

Tous les opérateurs prédéfinis, ainsi que les noeuds ont été étendus pour pouvoir être appliqués à tous les éléments d'un tableau ou d'une structure. Ainsi tout opérateur peut avoir des expressions de type structuré en lieu et place de ses opérandes habituels. Par exemple l'expression

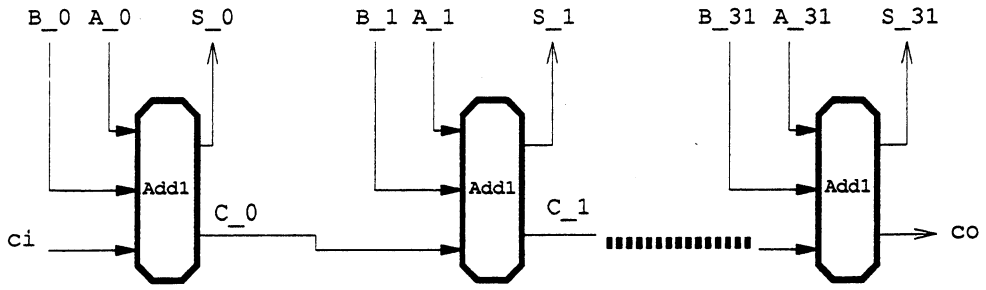


Figure 3.2 *Additionneur combinatoire 32 bits*

```
if [c0, c1, ... cn] then [e0, e1, ... en] else  [f0, f1, ... fn]
```

est équivalente à l'expression

```
[if c0 then e0 else f0, if c1 then e1 else f1, ... if cn then en else fn]
```

Les opérandes doivent avoir la même taille.

Le principal avantage de cette extension homomorphe des opérateurs est qu'elle permet, en conjonction avec l'utilisation des types structurés, de simplifier énormément la description des parties régulières d'un programme. Cette extension est par exemple utilisée dans le programme suivant pour décrire, à partir de la cellule d'additionneur binaire complet, un additionneur combinatoire classique de 32 bits.

```
node Add32 (A,B:bool^32; ci:bool)
returns (S:bool^32; co:bool);
var
   C:bool^32;
let
   (S,C) = Add1(A, B, [ci] | C[0..30]);
   co  = C[31];
end;
```

Cet additionneur est constitué de 32 cellules d'additionneurs (Add1), la $i^{ème}$ cellule recevant les $i^{ème}$ bits des opérandes A et B ainsi que de la retenue (retenue entrante) et produisant en sortie le $i^{ème}$ de la somme et le $i+1^{ème}$ bit de la retenue qui servira de retenue entrante à la cellule suivante. La dernière équation indique quant à elle que la retenue sortante de l'additionneur est la retenue sortante de la dernière cellule.

3.8 Assertions sur les structures

La notion d'assertion a été généralisée, avec l'introduction des types structurés et des expressions évaluables statiquement.

Une assertion usuelle est de la forme **assert e** où **e** est une expression booléenne. De façon similaire à ce qui vient d'être décrit pour les opérateurs, une assertion peut porter sur un tableau ou une structure dont les éléments sont booléens, son écriture est équivalente à celle consistant à écrire une assertion par élément de la structure ou du tableau. Par exemple, si **e** est une structure de n éléments `[l1: e1,l2: e2, ... ,ln: en]`, l'assertion

```
assert e
```

est équivalente aux n assertions

```
assert e1
assert e2
...
assert en
```

ou encore à l'assertion

```
assert e1 and e2 and ...  and en
```

3.9 Assertions Statiques

Si **e** est une expression de type **static bool**, **static assert e** est une assertion évaluable statiquement, le résultat du calcul de **e** par le compilateur doit être *true*.

Ces assertions permettent de spécifier des assertions sur les paramètres de noeuds pour, par exemple dans le cas de paramètres entiers, spécifier l'intervalle dans lequel ils doivent se trouver.

3.10 Paramètres statiques

Dans un entête de noeud, la déclaration d'une entrée ou d'une sortie peut être précédée du mot clé **const** (resp. **static**) pour préciser que dans toutes les instanciations du noeud, l'expression passée en paramètre sera forcément une constante (resp. évaluable statiquement).

L'intérêt est non seulement de permettre de faire des vérifications supplémentaires, mais également, dans le cas des paramètres statiques, de fournir le moyen de décrire des noeuds généraux, indépendant par exemple de la taille des données.

Les paramètres statiques permettent donc de simplifier l'écriture des programmes et il est possible d'écrire des bibliothèques de noeuds générales. La contrepartie est qu'un noeud ayant des paramètres statiques, ne peut être compilé seul, pour le tester par exemple, il doit être nécessairement instancié. Cela signifie que le noeud principal d'un programme ne doit pas avoir de paramètre statique.

Voici par exemple une autre façon de définir un additionneur 32 bits, à partir d'un addtionneur n bits (**AddN**) qui est instancié par le noeud principal du programme (**AddnN32**). Ici le paramètre est **n**, la taille des opérandes.

```
node AddN (static n:int; A, B:bool^n; ci:bool)
returns (S:bool^n; co:bool);
var C:bool^n;
let
   (S, C) = Add1(A, B, [ci] | C[0..n-2]);
   co  = C[n-1];
end;

node AddN32 (A, B:bool^32; ci:bool)
returns (S:bool^32; co:bool);
let
   (S, co) = AddN(32, A, B, i);
end;
```

3.11 Définition récursive d'opérateurs

Les valeurs de sortie d'un noeud LUSTRE , à un instant donné, peuvent dépendre, non seulement des valeurs courantes de ses variables d'entrées, mais aussi de leurs valeurs passées. Aussi, un noeud est à la fois une unité de calcul qui indique de quelle façon calculer les valeurs courantes des sorties, et une unité de mémorisation qui conserve une trace des valeurs passées de ses variables.

La conséquence de ceci est que deux instanciations d'un noeud correspondent à deux copies de celui-ci et non à deux appels. Un programme LUSTRE ne peut donc être compilé que si son arbre d'appel est fini et complètement connu au moment de la compilation. Cela implique qu'un programme, écrit avec la partie du langage présentée jusqu'à présent, ne peut contenir aucun appel récursif. Or, la récursivité présente pour le langage le même intérêt que les types structurés, c'est à dire un moyen de simplifier la description des programmes. De plus, son utilisation est une conséquence naturelle de l'introduction, dans le langage, des paramètres statiques aux noeuds.

La possibilité d'appeler récursivement un noeud a été introduite grâce à l'ajout d'un opérateur particulier, **with**, à trois entrées et une sortie, qui permet de paramétrer les équations. Etant donné une expression booléenne **e0** qui doit pouvoir être évaluée au moment de la compilation, et deux expressions **e1** et **e2** de type **t** quelconque, l'expression **with e0 then e1 else e2** est remplacée dans le programme par l'expression **e1** si **e0** est évaluée à *true*, **e2** sinon.

Voici par exemple un programme, calculant la conjonction des éléments d'un tableau de booléens. Cette opération est réalisée en calculant récursivement les conjonctions des éléments des premières et secondes moitiés des tableaux puis celle des deux résultats obtenus. L'opérateur **with** est utilisé pour arrêter la récursion lorsque les tableaux à comparer n'ont plus qu'un élément.

```
node recand(static n:int;a:bool^n) returns (c:bool);
let
   c = with (n = 1) then
            a[0]
       else
         recand(n/2,a[0..(n/2)-1]]) and recand((n+1)/2,a[(n/2) .. (n-1)]);
```

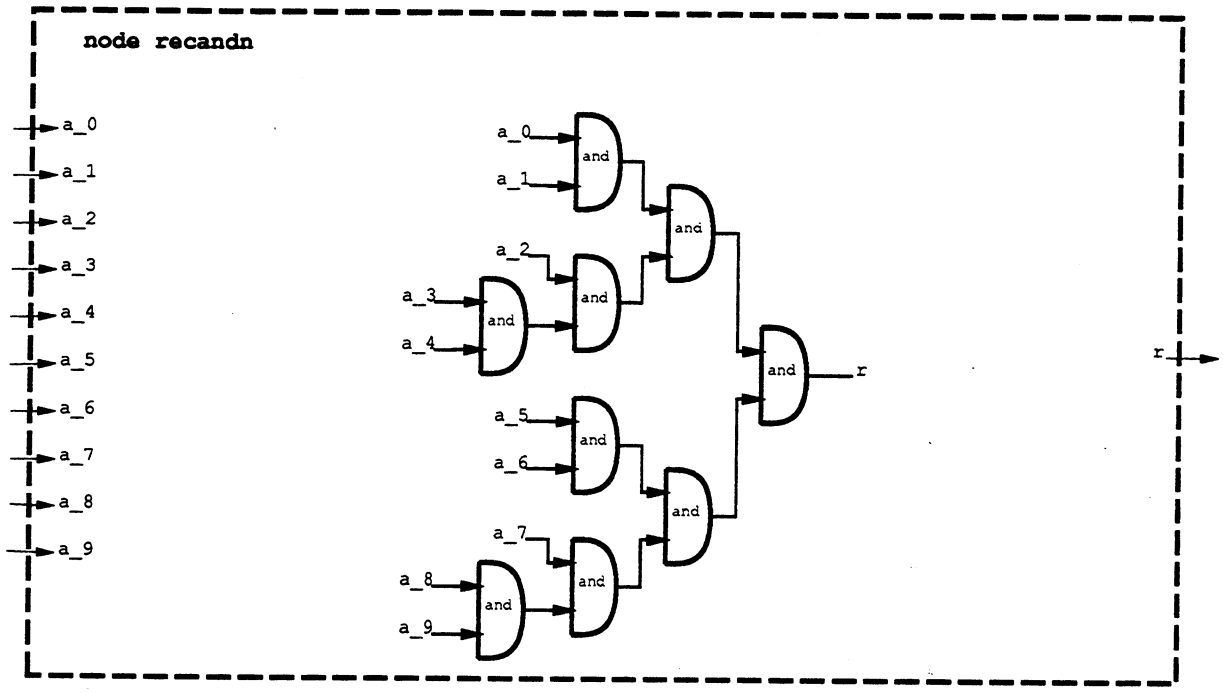


Figure 3.3 *le noeud recand10 après éclatement*

```
end;
```

Le noeud suivant est une instanciation du précédent pour `n = 10`, le résultat de ce programme est représenté Figure 3.3 .

```
node recand10 (a,b:bool^10) returns (r:bool);
let
   r = recand(10,a,b);
end;
```

3.12 Pragmas

Un programme ne fournit par défaut que la description du comportement d'un système. Or, en fonction de l'utilisation qui est faite de ce programme, il peut se révéler nécessaire de donner des informations supplémentaires. Mais LUSTRE étant utilisé dans des domaines différents, la nature de ces informations est susceptible d'être complètement différente d'une application à l'autre.

Un mécanisme de pragmas (ou commentaires exécutables) a donc été introduit pour permettre d'inclure dans un programme des informations qui seront utilisées par les outils LUSTRE capable de les interpréter et ignorées par les autres. Il est ainsi possible d'étendre le langage pour une utilisation particulière, sans pour autant le rendre incompatible avec les outils déjà existant.

Un pragma est une chaîne de caractères entre accolades débutant par un identificateur suivi de":". Cet identificateur permet de reconnaître le type de pragma : si l'outil sait le traiter, le reste de la chaîne est lu, dans le cas contraire, le reste du pragma, quel que soit son contenu, est ignoré.

Citons comme exemple la compilation répartie d'un programme LUSTRE [BCP88]. Il s'agit d'associer à chaque variable, le nom du site sur lequel elle est calculée. Nous verrons d'autres applications des pragmas lorsque nous décrirons l'implémentation de LUSTRE sur circuits.

3.13 Exemple

Voici un exemple de programme LUSTRE illustrant l'utilisation des tableaux à deux dimensions. Ce programme donne la description d'un bus de `size` bits de large, avec `nb_in` entrées et une sortie (noeud Bus), son instanciation dans le cas ou `size` vaut 32 et `nb_in` 3 (noeud `Bus_3_32`) et enfin le noeud permettant de connecter deux bus de même largeur entre eux (noeud `Bus_Connect`).

Nous désirons modéliser un bus ayant le comportement suivant :

Il reçoit en entrée *nb_in* données (`data_in`) et leurs fils de données associées (`enable_in`) ; la $(i+1)^{ieme}$ donnée `data_in[i]` est autorisée à accéder au bus uniquement si son fil de contrôle `enable_in[i]` vaut *true*. Si le bus fonctionne correctement, il y a à chaque instant qu'un fil de contrôle valant *true*, la valeur de la sortie est celle de l'entrée autorisée à accéder au bus.

La description du bus se décompose en deux parties :

- La première détermine la valeur en sortie de bus

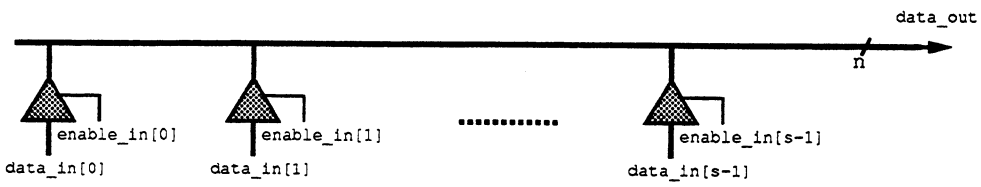


Figure 3.4 *bus de largeur n à s entrées et une sortie*

- La seconde s'assure que le fonctionnement du bus est correct.

La valeur de sortie du bus est déterminée par le noeud récursif **Bus_Value** :

- Si aucune entrée n'a accès au bus, sa valeur par défaut est **true^size**, tous ses fils sont à 1. C'est notamment le cas d'un bus sans entrée.
- Si une ou plusieurs entrées ont accès au bus la valeur du bus est égale à celle de l'entrée autorisée de plus haut rang. Ceci peut se reformuler en : la valeur d'un bus à n entrées est celle de sa dernière entrée si elle peut accéder au bus, celle du bus formé de ses n-1 premières entrées sinon.

L'état du bus est défini par trois booléens **good_drive**, **no_drive** et **drive_conflict**.

- Si tous les éléments de **enable_in** valent *false*, le bus est en haute impédance. Il est dans l'état no_drive
- Si un seul élément de **enable_in** vaut *true* le bus est normalement utilisé. Il est dans l'état good_drive
- Si plusieurs éléments de **enable_in** valent *true* il y a conflit d'accès au bus. Il est dans l'état drive_conflict

Cet état est déterminé par le noeud récursif **Bus_Status** chargé de compter en unaire de zéro à deux le nombre d'entrées ayant accès au bus (0 correspondant à l'état **no_drive**, 1 à l'état **good_drive** et 2 à l'état **drive_conflict**).

- Un bus sans entrée est forcément dans l'état *no_drive*.
- Si une entrée est rajoutée à un bus de n entrées
  - son état ne change pas si cette entrée est inhibée.
  - son état passe de *no_drive* (resp. *good_drive*) à *good_drive* (resp. *drive_conflict*) ou reste à *drive_conflict*, si cette entrée a accès au bus.

Un bus ayant un fonctionnement correct est toujours dans l'état *good_drive*, ce qui est spécifié par l'assertion du noeud **Bus**.

```
node UnaryCounter3 (up,i0,i1,i2 : bool)
returns (o0,o1,o2 : bool);
let
   (o0, o1, o2) = if up then (false, i0, i1 or i2)
                  else (i0, i1, i2);
end;

node Bus_Status (static n : int; enable : bool^n)
returns (no_drive, good_drive, drive_conflict : bool);
let
   (no_drive, good_drive, drive_conflict) =
      with (n = 0) then (true, false, false)
      else
         UnaryCounter3(enable[n-1], Bus_Status (n-1, enable[0..n-2 step 1]));
end;

node Bus_Value (static size,nb_in : int; data_in: bool^size^nb_in;
   enable_in: bool^nb_in) returns (data_out: bool^size);
let
   data_out =  with (nb_in = 0)^size then true^size
               else if (enable_in[nb_in-1])^size then data_in[nb_in-1]
                    else  Bus_Value (size, nb_in - 1,
                                     data_in[0..nb_in-2 step 1],
                                     enable_in[0..nb_in-2 step 1]);
end;

node Bus (static size,nb_in: int; data_in: bool^size^nb_in;
   enable_in: bool^nb_in) returns (data_out : bool^size);
var
   no_drive,good_drive,drive_conflict : bool;
let
   data_out = Bus_Value(size, nb_in, data_in, enable_in);
   (no_drive,good_drive,drive_conflict)= Bus_Status(nb_in,enable_in);
   assert good_drive and (not no_drive) and (not drive_conflict);
end;

static size = 32;
static nb_in = 3;

type mot = bool^size;

node Bus_3_32 (data_in : word^3; enable_in : bool^3)
returns (data_out : word);
let
   data_out = Bus(nb_in,size,data_in,enable_in);
end;
```

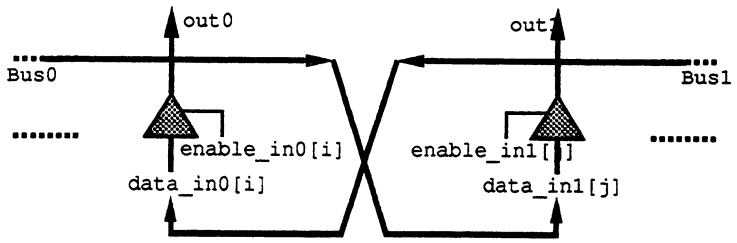


Figure 3.5 *Connexion incorrecte de deux bus entre eux*

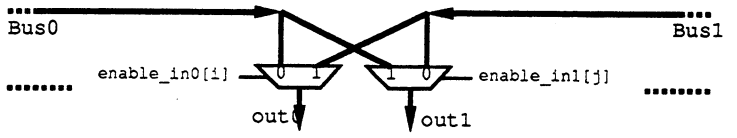


Figure 3.6 *Connexion correcte de deux bus entre eux*

Un pont, permettant à deux bus d'échanger des informations, est intuitivement réalisé en reliant la sortie de l'un à l'entrée de l'autre et vice versa comme représenté sur la Figure 3.5 . Mais cette connexion a le défaut de ne marcher que si `enable_in0[i]` et `enable_in1[j]` sont des signaux exclusifs ; il y a court circuit autrement. L'analyse de ce genre de problèmes étant très stricte en LUSTRE, un tel programme est systématiquement considéré comme incorrect.

Le noeud `Bus_Connect`, schématisé par la Figure 3.6 , fournit un moyen équivalent de connecter deux bus, sans se heurter au problème de rebouclage. Si les deux commandes `c0` et `c1` sont à *false*, les deux bus sont séparés, `s0` (resp. `s1`) sort la valeur sur le bus 0 (resp. 1). Une commande `ci` valant *true* signifie que `si` prend la valeur de l'autre bus.

c0	c1	
false	*false*	`s0 = e0; s1 = e1;`
true	*false*	`s0 = e1; s1 = e1;`
false	*true*	`s0 = e0; s1 = e0;`
true	*true*	`s0 = e1; s1 = e0;`

La valeur de `ci` définit sur quel bus la sortie `si` prend sa valeur .

```
   node BusConnect (static n:int; c0,c1:bool; e0,e1:bool^n)
   returns (s0,s1: bool^n);
let
   s0 = if c0 then e1 else e0;
   s1 = if c1 then e0 else e1;
end;
```

3.14 Horloges

3.14.1 Modification du calcul d'horloges

Le défaut du calcul d'horloge des versions précédentes de LUSTRE (§ 1.9.4) est :

- qu'il s'accorde mal avec la notion de hiérarchie,
- que la notion d'horloge n'est alors pas transparente pour un utilisateur écrivant des programmes purement synchrones.

En effet, si l'on considère le noeud `toggle` décrit ci-dessous, l'horloge de la variable `o` est `true`.

```
node toggle () returns (o:bool);
let
   o = false-> pre not o;
end;
```

Si maintenant l'on réécrit ce noeud en utilisant un noeud intermédiaire, il est nécessaire de déclarer l'horloge de la variable `o` pour que le programme ne soit pas rejeté.

```
node freg(v:bool) returns (p:bool);
let
   p = false -> pre v;
end;

node toggle () returns ((o:bool) when true);
let
   o = freg (not o);
end;
```

Pour résoudre ce problème, le calcul d'horloge a été modifié. Initialement, les horloges de toutes les constantes et de toutes les variables sont connues. La déclaration de la variable `x`

- `x:t` définit que son horloge est `true`
- `(x:t) when h` définit que son horloge est `h`

`t` étant un type et `h` une autre variable booléenne.

Le calcul d'horloge est alors simplifié, une itération du précédent calcul suffit pour calculer toutes les horloges possibles.

3.14.2 Horloge d'un noeud sans entrée

En LUSTRE V3, puisque les horloges étaient par défaut indéfinies, tout noeud devait avoir forcément une entrée, pour permettre de déterminer son horloge de base. En LUSTRE V4,

un noeud sans entrée peut être défini, son horloge de base est, par défaut, l'horloge de base du noeud appelant.

Si l'on reprend l'exemple précédent, l'équation `t = toggle ()` définit une variable booléenne `t` sur l'horloge de base et ayant pour suite de valeurs :

`true` : (	*true*	, *true*	, *true*	, *true*	, *true*	, *true*	, *true*	, *true*	, *true*	, ...)
`t` : (	*false*	, *true*	, *false*	, *true*	, *false*	, *true*	, *false*	, *true*	, *false*	, ...)

un noeud sans entrée peut cependant quand même être échantillonné sur une autre horloge. Si h est une variable booléenne alors, l'équation `t = toggle (() when h)` définit une variable booléenne `t` ayant pour horloge logique `h`, et pour suite de valeurs :

`true` : (	*true*	, *true*	, *true*	, *true*	, *true*	, *true*	, *true*	, *true*	, *true*	, ...)
`h` : (	*true*	, *false*	, *true*	, *false*	, *true*	, *true*	, *false*	, *true*	, *true*	, ...)
`t` : (	*false*		, *true*		, *false*	, *true*		, *false*	, *true*	, ...)

Partie II

Sémantique naturelle

Chapitre 4

Définitions

4.1 Introduction

Le domaine d'application initialement visé par LUSTRE étant la programmation de systèmes nécessitant une grande sûreté de fonctionnement, un des principaux soucis des concepteurs de ce langage a été de lui donner une définition claire et précise. Pour cette raison, la description de la sémantique formelle de LUSTRE s'est faite en parallèle avec sa conception.

Le but recherché était double :

- Faire en sorte que le comportement de tout programme soit clairement défini sans aucune ambigüité possible.
- Simplifier la réalisation de compilateurs ou d'outils.

Cette sémantique initiale est restée inchangée. Cependant, de façon à être la plus complète possible et couvrir tous les aspects du langage, elle a été décrite sous différentes formes :

- La sémantique opérationnelle dans [Buo88] [HL91]
- La sémantique dénotationnelle dans [Ber86] [Pla88]

Lorsqu'il s'est révélé nécessaire d'étendre LUSTRE, nous avons avant tout recherché a conserver les caractéristiques fondamentales du langage. Nous nous sommes donc attachés à donner aux extensions définies une sémantique rigoureuse et la plus simple possible que nous allons maintenant décrire.

La sémantique naturelle nous a semblé la plus adaptée pour donner une définition aussi claire et complète que possible de ces extensions sans entrer dans les détails du processus de compilation, comme avec la sémantique opérationnelle.

Les deux points qui vont être abordés sont :

- Le calcul des types
- L'évaluation des expressions

en fonction desquels il sera possible de définir ce qu'est un programme LUSTRE correct.

4.2 Notations

Les notations utilisées dans ce chapitre sont proches de celles employées par [Pla88].

La sémantique naturelle de LUSTRE est décrite par un ensemble de règles d'inférence structurelles à la Plotkin [Plo81]. Chacune d'entre elles est de la forme :

$$\frac{P_1,\ P_2,\ \ldots\ P_n}{P}$$

P_1, P_2, ... P_n sont n prédicats formant la prémisse de la règle. S'ils sont tous vérifiés, alors le prédicat P l'est également. Les règles les plus contraintes s'appliquent en priorité.

Les prédicats sont définis à partir de la syntaxe abstraite du langage. Voici quelques exemples typiques de prédicats employés :

- $\alpha \vdash o : i$ signifie que, dans l'environnement α, l'objet o (une expression de la syntaxe abstraite dans notre cas) synthétise l'information i.
- $\alpha \vdash o \rightarrow o'$ signifie que, dans l'environnement α, l'objet o se réécrit en l'objet o'.

Un environnement désigne une fonction qui associe des informations à des objets. Ce peut être par exemple l'environnement qui associe sa valeur à une expression.

Les notations suivantes seront employées pour les environnements ou les fonctions :

- $\alpha(o) = i$ indique que l'information i est associée à l'expression o (le fait qu'aucune information ne soit attaché à un objet o est noté $\alpha(o) = \bot$)
- i/o désigne la fonction :
$$\lambda o'.\ si\ o' = o\ alors\ i\ sinon\ \bot$$
- $\alpha[\alpha']$ désigne l'environnement défini par :
$$\alpha''(o) = si\ \alpha'(o) = \bot\ alors\ \alpha(o)\ sinon\ \alpha'(o)$$
- π_k sera utilisé pour désigner la k^{eme} composante d'un produit cartésien :
$$\pi_k(o_1,\ o_2,\ \ldots\ , o_n) = o_k$$
- Si o est un élément d'un ensemble O, o^n désigne le n-uplet $\underbrace{(o,\ \ldots\ ,\ o)}_{n}$ du produit cartésien $\underbrace{O \times \ldots \times O}_{n}$ également noté O^n.
- Les ensembles de départ et d'arrivée d'une fonction f sont respectivement notés $Dom(f)$ et $Ran(f)$.

4.3 Langage réduit

Dans un premier temps, afin de simplifier la description de sa sémantique, nous allons nous intéresser à un sous-ensemble du langage LUSTRE V4 dans lequel les seules extensions prises en compte sont les expressions statiques. Les horloges ne seront également pas prises en compte.

4.3.1 Syntaxe Abstraite

4.3.1.1 Méta-syntaxe

Dans la grammaire suivante,

- Les terminaux sont représentés en gras
- *id* désigne un identificateur et *val* une valeur.
- ‖ définit un choix
- [[et]] encadrent une information optionnelle.

4.3.1.2 Listes

La règle générique pour toute liste d'objets *obj* est :

$$obj\_list ::= \varepsilon \parallel obj\ obj\_list$$

4.3.1.3 Programmes

Un programme LUSTRE_V4 se présente sous la forme d'une liste de déclarations définissant une hiérarchie d'opérateurs dont la racine est nommée *id*. Pour des raisons de commodité uniquement, l'ordre des déclarations étant sans importance, nous avons séparé cette liste en deux :

- une liste de déclarations d'objets : types, constantes et statiques.
- une liste de déclaration d'opérateurs.

$$pgm ::= \texttt{prog}\ id\ decl_{obj}\_list\ decl_{op}\_list$$

4.3.1.4 Déclarations

Les définitions des types, des constantes, tout comme les opérateurs peuvent être locales ou externes, seuls les statiques (des constantes particulières dont la valeur est évaluée à la compilation) sont forcément définis localement.

$$\begin{aligned} decl_{obj} ::= {} & \texttt{type}\ id\ type \parallel \texttt{static}\ id\ exp \parallel \texttt{const}\ id\ exp \parallel \\ & \texttt{type}\ id \parallel \texttt{const}\ id\ type \end{aligned}$$

decl$_{op}$ ::= **node** *id* **in** *var_list* **out** *var_list* **loc** *var_list dep_list eq_list* ‖
node *id* **in** *var_list* **out** *var_list dep_list* ‖
function *id* **in** *var_list* **out** *var_list*

4.3.1.5 Dépendances

dep ::= *left* **needs** *exp_list*

4.3.1.6 Variables

var ::= ⟦**const** ‖ **static** ⟧*id type*

4.3.1.7 Types

type ::= *id* ‖ $*_k$ ‖ *type* ^ *exp_list* ‖ [*type_list*]

4.3.1.8 Equations

eq ::= *left_list* = *exp_list* ‖ **assert** *exp* ‖ **static assert** *exp*

4.3.1.9 Expressions

exp ::= *id* ‖ *val* ‖ *id* (*exp_list*) ‖ *exp1* | *exp2* ‖ *exp* ^ *exp_list*
‖ *exp* [*exp_list*] ‖ [*exp_list*] ‖ *exp1* .. *exp2* **step** *exp3* ‖ *exp3*
‖ *exp_list*

4.3.1.10 Expressions en partie gauche d'équations

left ::= *left* [*exp_list*] ‖ [*left_list*] ‖ *left_list* ‖
with *exp* **then** *left_list* **else** *left_list*

4.3.2 Domaines syntaxiques

Chaque règle de la syntaxe abstraite

non_term ::= ...

définit un domaine syntaxique noté $\mathcal{NON\_TERM}$. Ainsi, le domaine des expressions est noté $\mathcal{EXP}$.

4.3.3 Domaines de base

$\mathcal{ID} \ni id, i$: Identificateurs

$\mathcal{R} \ni r$: Réels

$\mathcal{N} \ni n$: Entiers Naturels

$\mathcal{B} \ni b$: Booléens

$\mathcal{C} \ni c$: Caractères

$\mathcal{VAL} = \mathcal{B} \cup \mathcal{N} \cup \mathcal{R} \ni v$: Valeurs

Chapitre 5

Programme bien construit

Dans un premier temps nous allons nous intéresser uniquement à la structure d'un programme et spécifier sous quelles conditions le comportement qu'il décrit est bien défini et unique.

5.1 Genres

5.1.1 Domaine des genres

$\mathcal{GENRE} = \{ \perp \} \cup \{static,\ const\ , var\} \cup \mathcal{GENRE}^k \ni g$

Le genre d'une expression est

- $\perp$ s'il est indéfini
- *static* si la suite de valeurs qu'elle prend est constante et peut être connue au moment de la compilation.
- *const* si la suite de valeurs qu'elle prend est constante mais n'est connue qu'à l'exécution.
- *var* si la suite de valeurs qu'elle prend est variable.

Tous les éléments d'un tableau ont le même genre, en revanche, chaque élément d'une liste d'expressions peut avoir le sien. Le genre d'une liste de n expressions est donc un n-uplet de genres.

5.1.2 Environnements

5.1.2.1 Environnement des genres des objets

$\mathcal{GOBJENV} = \mathcal{ID} \rightarrow \mathcal{GENRE} \ni \gamma$

γ est l'environnement associant son genre à une un identificateur de variable, de constante ou de statique.

5.1.2.2 Environnement des genres des opérateurs

$\mathcal{GFUNENV} = \mathcal{ID} \rightarrow \mathcal{GENRE} \times \mathcal{GENRE} \ni \psi$

ψ est l'environnement associant au nom d'un opérateur le genre de la liste de ses paramètres d'entrées et celui de la liste de ses paramètres de sortie.

5.1.3 Fonctions utiles

Les trois catégories d'expressions, statiques, constantes et variables sont incluses les unes dans les autres, une expression statique peut être utilisée comme une expression constante ou variable et une expression constante comme une expression variable.

Plus formellement, ceci signifie que le domaine des genres est muni d'une relation d'ordre $<$ vérifiant :

$$\bot < static < const < var$$

Cette relation nous permet de définir la fonction $\sqcup$ de $\mathcal{GENRE} \times \mathcal{GENRE} \rightarrow \mathcal{GENRE}$ qui calcule la borne supérieure de deux genres élémentaires :

$$\sqcup(g,\ g') \ =\ si\ g < g'\ alors\ g'\ sinon\ g$$

$$\sqcup(g_1 \ldots g_k,\ g'_1..g'_k) \ =\ \sqcup(g_1,\ g'_1) \ldots \sqcup(g_k,\ g'_k)$$

Nous définissons également la fonction $\bigsqcup$ de $\mathcal{GENRENLIST} \rightarrow \mathcal{GENRE}$ dont le résultat est le plus grand élément d'une liste de genres élémentaires :

$$\bigsqcup(g_1 \ldots g_k) = \sqcup(g_1,\ \bigsqcup(g_2 \ldots g_k)$$

$$\bigsqcup(static) = static$$

$$\bigsqcup(const) = const$$

$$\bigsqcup(var) = var$$

5.1.4 Prédicats

$$\gamma,\ \psi \overset{\substack{check\\genre}}{\vdash} e : g$$

$$\gamma,\ \psi \overset{\substack{check\\genre}}{\vdash} o$$

5.1.5 Programmes

$$\frac{\gamma,\ \psi \overset{\substack{check\\genre}}{\vdash} decl_{obj}\_list \qquad \gamma,\ \psi \overset{\substack{check\\genre}}{\vdash} decl_{op}\_list}{\gamma,\ \psi \overset{\substack{check\\genre}}{\vdash} prog\ id\ decl_{obj}\_list\ decl_{op}\_list}$$

La définition de ψ_{predef}, l'environnement ψ limité aux seuls opérateurs prédéfinis est la suivante :

$$\begin{array}{[lllll} (g & \rightarrow & \sqcup(g)) & / & op_{data} \\ (g & \rightarrow & var) & / & op_{data} \\ (static\ g_1\ g_2 & \rightarrow & \sqcup(g_1, g_2))) & / & with \quad] \end{array}$$

$op_{data} \in \{and,\ or,\ xor,\ plus,\ moins,\ mul,\ div,\ sup,\ inf,\ supeg,\ infeg,\ egal,\ dif\}$

$op_{temp} \in \{arrow,\ pre,\ when,\ current\}$

5.1.6 Déclarations

5.1.6.1 Types

$$\gamma,\ \psi \overset{\substack{check\\genre}}{\vdash} \texttt{type}\ id\ type$$

$$\gamma,\ \psi \overset{\substack{check\\genre}}{\vdash} \texttt{type}\ id$$

5.1.6.2 Statiques

$$\frac{\gamma,\ \psi \overset{genre}{\vdash} exp : static \qquad \gamma(id) = static}{\gamma,\ \psi \overset{\substack{check\\genre}}{\vdash} \texttt{static}\ id\ exp}$$

5.1.6.3 Constantes

$$\frac{\gamma,\ \psi \overset{genre}{\vdash} exp : g \qquad g \leq const \qquad \gamma(id) = const}{\gamma,\ \psi \overset{\substack{check\\genre}}{\vdash} \texttt{const}\ id\ exp}$$

$$\frac{\gamma(id) \;=\; const}{\gamma,\ \psi \overset{\substack{check\\genre}}{\vdash} \texttt{const}\ id\ \ type}$$

5.1.6.4 Noeuds locaux

5.1.7 Expressions

5.1.7.1 Identificateurs

$$\gamma,\ \psi \overset{\text{genre}}{\vdash} id\ :\ \gamma(id)$$

5.1.7.2 Valeurs

$$\gamma,\ \psi \overset{\text{genre}}{\vdash} b\ :\ static$$

$$\gamma,\ \psi \overset{\text{genre}}{\vdash} n\ :\ static$$

$$\gamma,\ \psi \overset{\text{genre}}{\vdash} r\ :\ static$$

$$\gamma,\ \psi \overset{\text{genre}}{\vdash} c\ :\ static$$

5.1.7.3 Opérateurs

Un opérateur id est appliqué à une liste d'expressions de genre g et en produit une autre en sortie de type g'. Ces genres sont déterminés grâce à l'environnement des fonctions, une fois les valeurs des entrées statiques déterminée.

$$\frac{\begin{array}{c}\gamma,\ \psi \overset{\text{genre}}{\vdash} exp\_list\ :\ h \\ \psi(id) \;=\; g \;\rightarrow\; g' \\ h \leq g\end{array}}{\gamma,\ \psi \overset{\text{genre}}{\vdash} id\ (exp\_list)\ :\ g'}$$

5.1.7.4 Opérateurs de concaténation

$$\frac{\gamma,\ \psi \overset{\text{genre}}{\vdash} \mathit{exp1} : g \qquad \gamma,\ \psi \overset{\text{genre}}{\vdash} \mathit{exp2} : g'}{\gamma,\ \psi \overset{\text{genre}}{\vdash} \mathit{exp1} \mid \mathit{exp2} : \sqcup(g,\ g')}$$

5.1.7.5 Opérateurs dextension

$$\frac{\gamma,\ \psi \overset{\text{genre}}{\vdash} \mathit{exp} : g \qquad \gamma,\ \psi \overset{\text{genre}}{\vdash} \mathit{exp\_list} : \mathit{static}^k}{\gamma,\ \psi \overset{\text{genre}}{\vdash} \mathit{exp}\ \hat{}\ \mathit{exp\_list} : g}$$

5.1.7.6 Opérateurs de sélection

$$\frac{\gamma,\ \psi \overset{\text{genre}}{\vdash} \mathit{exp} : g \qquad \gamma,\ \psi \overset{\text{genre}}{\vdash} \mathit{exp\_list} : \mathit{static}^k}{\gamma,\ \psi \overset{\text{genre}}{\vdash} \mathit{exp}\ [\mathit{exp\_list}] : g}$$

5.1.7.7 Opérateurs de construction

$$\frac{\gamma,\ \psi \overset{\text{genre}}{\vdash} \mathit{exp\_list} : g}{\gamma,\ \psi \overset{\text{genre}}{\vdash} [\mathit{exp\_list}] : \sqcup(g)}$$

5.1.7.8 Opérateurs d'énumération

$$\frac{\gamma,\ \psi \overset{\text{genre}}{\vdash} \mathit{exp1} : \mathit{static} \qquad \gamma,\ \psi \overset{\text{genre}}{\vdash} \mathit{exp2} : \mathit{static} \qquad \gamma,\ \psi \overset{\text{genre}}{\vdash} \mathit{exp3} : \mathit{static}}{\gamma,\ \psi \overset{\text{genre}}{\vdash} \mathit{exp1}\ ..\ \mathit{exp2}\ \texttt{step}\ \mathit{exp3} : \mathit{static}}$$

5.1.7.9 Listes d'expressions

$$\frac{\gamma,\ \psi \overset{\text{genre}}{\vdash} exp\ :\ g \qquad exp \neq exp\_list' \qquad \gamma,\ \psi \overset{\text{genre}}{\vdash} exp\_list\ :\ g_1\ ...\ g_k}{\gamma,\ \psi \overset{\text{genre}}{\vdash} exp\ exp\_list:\ g g_1\ ...\ g_k}$$

Les listes d'expressions ne sont pas hiérarchiques en LUSTRE, le genre d'une liste *exp_list* dont certains de ses éléments sont eux-mêmes des listes est le même que celui la liste obtenu une fois *exp_list* mise à plat.

$$\frac{\gamma,\ \psi \overset{\text{genre}}{\vdash} exp\ :\ g_1\ ...\ g_k \qquad \gamma,\ \psi \overset{\text{genre}}{\vdash} exp\_list\ :\ g'_1\ ...\ g'_{k'}}{\gamma,\ \psi \overset{\text{genre}}{\vdash} exp\ exp\_list:\ g'_1\ ...\ g'_k g_1\ ...\ g'_{k'}}$$

5.1.8 Déclarations d'un noeud

Dans un noeud, Il y a trois types de déclarations :

- Les paramètres d'entrée.
- Les paramètres de sortie.
- Les variables, constantes et statiques locaux au noeud.

Le type des déclarations de constantes et de statiques, dans un noeud, est donné explicitement et leur définition repoussée dans le système d'équations. Toutes ces déclarations ont ainsi un format similaire.

5.1.8.1 Variables

Toute variable d'un programme LUSTRE est définie à l'intérieur d'un noeud, il n'existe aucune variable globale.

$$\frac{\gamma,\ \psi \overset{\text{genre}}{\vdash} type\ :\ t \qquad \gamma(id)\ =\ (t,\ var)}{\gamma,\ \psi \overset{\text{genre}}{\vdash} id\ type\ :\ \gamma(id)}$$

5.1.8.2 Constantes

$$\frac{\gamma, \psi \overset{\text{genre}}{\vdash} \mathit{type} : t \quad \gamma(\mathit{id}) = (t, \mathit{const})}{\gamma, \psi \overset{\text{genre}}{\vdash} \texttt{const}\ \mathit{id}\ \mathit{type} : \gamma(\mathit{id})}$$

5.1.8.3 Statiques

$$\frac{\gamma, \psi \overset{\text{genre}}{\vdash} \mathit{type} : t \quad \gamma(\mathit{id}) = (t, \mathit{static})}{\gamma, \psi \overset{\text{genre}}{\vdash} \texttt{static}\ \mathit{id}\ \mathit{type} : \gamma(\mathit{id})}$$

5.1.8.4 Liste de variables

$$\frac{\gamma, \psi, \mu \overset{\text{genre}}{\vdash} \mathit{var} : g \quad \gamma, \psi, \mu \overset{\text{genre}}{\vdash} \mathit{var\_list} : g_1, \ldots, g_k}{\gamma, \psi \overset{\text{genre}}{\vdash} \mathit{var}\ \mathit{var\_list} : (g, g_1, \ldots, g_k)}$$

5.1.9 Dépendances

Une dépendance sert à indiquer que les résultats de différents calculs sont nécessaires à un autre. Le type de la partie gauche de la dépendance et celui de chaque élément de la liste en partie droite doivent être identiques. De plus, l'évaluation d'un statique ne peut dépendre de celle d'une constante ou d'une variable, l'évaluation d'une constante de celle d'une variable.

$$\frac{\gamma, \psi \overset{\text{genre}}{\vdash} \mathit{left} : t, \gamma \quad \gamma, \psi \overset{\text{genre}}{\vdash} \mathit{exp\_list} : nt, n\gamma \quad \forall n\ (nt(n) = t) \quad \forall n\ (n\gamma(n) \leq \gamma)}{\gamma, \psi \overset{\text{check}}{\vdash} \mathit{left\_list}\ \texttt{needs}\ \mathit{exp\_list'}}$$

Un ensemble de dépendances est correctement typé si chacune d'entre-elles l'est.

$$\frac{\gamma,\ \psi \overset{\text{check}}{\vdash} dep \qquad \gamma,\ \psi \overset{\text{check}}{\vdash} dep\_list}{\gamma,\ \psi \overset{\text{check}}{\vdash} dep\ dep\_list}$$

5.1.10 Equations

Une équation est correctement typée si ses deux membres sont de même-type et leur genres compatibles, un statique ne peut être défini en fonction d'une constante ou d'une variable et un constante en fonction d'une variable.

Une équation se compose d'une liste d'objets en partie gauche et d'une liste d'expression, donnant leur définition, en partie droite.

$$\frac{\begin{array}{c}\gamma,\ \psi \overset{\text{genre}}{\vdash} left\_list\ :\ nt,\ n\gamma \\ \gamma,\ \psi \overset{\text{genre}}{\vdash} exp\_list\ :\ nt,\ n\gamma' \\ \forall n\ (n\gamma'(n)\ \leq\ n\gamma(n))\end{array}}{\gamma,\ \psi \overset{\text{check}}{\vdash} left\_list = exp\_list}$$

La sémantique de la partie gauche d'une équation n'étant qu'un cas particulier de celle des expressions, les règles la décrivant ne seront pas données.

5.1.11 Assertions

Une assertion est correctement typée si le type de *exp* est dérivé de *bool*, c'est à dire s'il est obtenu en appliquant un certain nombre de fois les constructeurs de types au type *bool*.

De façon générale, nous dirons qu'un type t est dérivé d'un type t' si le premier peut être obtenu à partir du second en lui appliquant un certain nombre de fois les constructeurs de types. Par exemple, `(int^5)^4` et `((int^5)^(4,3))^12` sont deux types dérivés `(int^5)`.

Une définition complète de cette notion de type dérivé est donné par le prédicat *derived_type*

$$\overset{\text{derived\_type}}{\vdash}\ (t,\ t')$$

est vérifié si t est un type dérivé de t'

La dérivation de type la plus simple dérive un type en lui même

$$\overset{\text{derived\_type}}{\vdash}\ (t,\ t)$$

Un type est dérivé d'un type structuré si ses éléments le sont également

$$\frac{\overset{\text{derived\_type}}{\vdash \quad (t,\ t')}}{\overset{\text{derived\_type}}{\vdash \quad (t,\ (t',\ \delta))}}$$

La règle définissant qu'une assertion est correctement typée est alors :

$$\frac{\begin{array}{c}\overset{\text{genre}}{\gamma,\ \psi \quad \vdash \quad exp\ :\ t,\ n\gamma} \\ \overset{\text{derived\_type}}{\vdash \quad (bool, t)}\end{array}}{\overset{\text{check}}{\gamma,\ \psi \quad \vdash \quad \texttt{assert}\ exp}}$$

Pour une assertion statique, on impose en plus que le genre soit statique, et que sa valeur soit *true* ou dérivée de *true*. Leur rôle est de donner des contraintes sur les valeurs des statiques d'un noeud, de façon à restreindre ses instanciations possibles. Une valeur fausse signifie que l'instanciation est interdite.

La notion de valeur dérivée se définit de façon similaire à celle de type dérivé :

$$\overset{\text{derived\_val}}{\vdash \quad (\nu,\ \nu)}$$

$$\frac{\forall n((n\nu(n) \neq \bot)) \Rightarrow \overset{\text{derived\_val}}{\vdash \quad (\nu,\ \nu(n))}}{\overset{\text{derived\_val}}{\vdash \quad (\nu,\ n\nu')}}$$

$$\frac{\begin{array}{c}\overset{\text{genre}}{\gamma,\ \psi \quad \vdash \quad exp\ :\ t,\ static} \\ \overset{\text{derived\_type}}{\vdash \quad (bool, t)} \\ \overset{\text{genre}}{\gamma,\ \psi \quad \vdash \quad exp : \nu} \\ \overset{\text{derived\_val}}{\vdash \quad (true,\ \nu)}\end{array}}{\overset{\text{check}}{\gamma,\ \psi \quad \vdash \quad \texttt{static assert}\ exp}}$$

5.1.12 Liste d'équations

Un ensemble d'équations est correctement typé si chacune d'entre elle l'est.

$$\frac{\begin{array}{c}\overset{\text{check}}{\gamma,\ \psi \quad \vdash \quad eq} \\ \overset{\text{check}}{\gamma,\ \psi \quad \vdash \quad eq\_list}\end{array}}{\overset{\text{check}}{\gamma,\ \psi \quad \vdash \quad eq\ eq\_list}}$$

5.2 Calcul des types

5.2.1 Types

Les règles suivantes définissent la construction du représentant canonique d'un type.

Le type associé à un identificateur *id* déclaré est fourni par l'environnement des types

$$\frac{(id) \neq \perp}{\gamma, \overset{\text{genre}}{\vdash} id : (id)}$$

$$\overset{\text{genre}}{\vdash} *_k : *_k$$

Les bornes d'un tableau doivent forcément être connues au moment de la compilation. Le type d'un tableau à k dimensions est donc défini à partir du type t de ses éléments et de k expressions statiques, les tailles des dimensions successives.

$$\frac{\begin{array}{c}\gamma, \psi \overset{\text{genre}}{\vdash} type : t \\ \gamma, \psi \overset{\text{genre}}{\vdash} exp\_list : \lambda p.\ (si\ p < k\ alors\ int\ sinon\ \perp),\ \lambda p.\ (si\ p < k\ alors\ static\ sinon\ \perp) \\ \gamma, \psi \overset{\text{val}}{\vdash} exp\_list : n\nu \\ \forall p (n\nu(p) \neq \perp \Leftrightarrow p < k)\end{array}}{\gamma, \psi \overset{\text{genre}}{\vdash} type\hat{}exp\_list : (t, n\nu)}$$

Le type associé à un identificateur *id* déclaré est fourni par l'environnement des types Θ

$$\alpha, \varphi, \Theta \overset{\text{type}}{\vdash} id : \Theta(id)$$

$$\alpha, \varphi, \Theta \overset{\text{type}}{\vdash} *_k : *_k$$

Les bornes d'un tableau doivent forcément être connues au moment de la compilation. Le type d'un tableau à k dimensions est donc défini à partir du type t de ses éléments et de k expressions statiques, les tailles des dimensions successives.

$$\frac{\begin{array}{c}\alpha, \varphi, \Theta \overset{\text{type}}{\vdash} type : t \\ \alpha, \varphi, \Theta \overset{\text{type}}{\vdash} exp\_list : int^k\end{array}}{\alpha, \varphi, \Theta \overset{\text{type}}{\vdash} type\hat{}exp\_list : (t, k)}$$

5.2.2 Opérateurs

Un opérateur *id* est appliqué à une liste d'expressions de type *nt* et en produit une autre en sortie de type *nt'*. Ces types sont déterminés grâce à l'environnement des fonctions, une fois les valeurs des entrées statiques déterminée.

Dans le cas d'un opérateur non polymorphe, l'application de *id* à *exp_list* est correcte si la liste des types des paramètres formels d'entrée est égale à *nt*.

$$\frac{\alpha,\ \varphi,\ \Theta \overset{\text{type}}{\vdash} exp\_list\ :\ nt \qquad \varphi(id)\ =\ nt\ \rightarrow\ nt'}{\alpha,\ \varphi,\ \Theta \overset{\text{type}}{\vdash} id\ (exp\_list)\ :\ nt'}$$

Le cas d'un opérateur polymorphe est plus complexe et généralise le précédent, puisque $n\theta$ ne désigne plus un type unique mais un ensemble de types. L'opérateur *id* peut être appliqué à n'importe quelle liste d'expressions s'il est possible d'unifier son type *nt* avec celui des paramètres formels $n\theta$.

L'unificateur, noté Φ, est un ensemble de couples $(*_k,\ t)$ associant à tout type variable $*_k$, défini dans $n\theta$, un type t élémentaire ou un autre type variable $*_p$ par lequel il faut remplacer toute ses occurrences dans $n\theta$ pour obtenir *nt*.

Φ est défini comme une fonction des types dans les types. Le prédicat suivant indique que deux types θ et t peuvent être unifiés en utilisant Φ comme unificateur.

$$\Phi \overset{\text{unify}}{\vdash} \theta,\ t$$

Un type variable $*_k$ s'unifie avec un type élémentaire (dont la représentation canonique est un identificateur) ou $*_p$. Toutes les occurrences de $*_k$ doivent être unifiées avec le même type.

$$\frac{\Phi(k) = id}{\Phi \overset{\text{unify}}{\vdash} *_k,\ id}$$

$$\frac{\Phi(k) = *p}{\Phi \overset{\text{unify}}{\vdash} *_k,\ *_p}$$

deux types identiques s'unifient

$$\Phi \overset{\text{unify}}{\vdash} t,\ t$$

Deux tableaux s'unifient seulement s'ils ont les même dimensions et si les types de leurs éléments s'unifient

$$\frac{\Phi \overset{\text{unify}}{\vdash} \theta,\ t}{\Phi \overset{\text{unify}}{\vdash} (\theta,\ n),\ (t,\ n)}$$

Deux listes s'unifient seulement si leurs éléments s'unifient deux à deux.

$$\frac{\forall n \ \Phi \overset{\text{unify}}{\vdash} n\theta(n),\ nt(n)}{\Phi \overset{\text{unify}}{\vdash} n\theta,\ nt}$$

$$\Phi \overset{\text{unify}}{\vdash} \bot,\ \bot$$

Pour pouvoir déterminer le type résultat d'une instanciation d'opérateur polymorphe, il est nécessaire de déterminer l'unificateur Φ permettant le passage du type de ses paramètres formels d'entrée $n\theta$ à celui de ses paramètres effectifs d'entrée nt. L'unificateur Φ', permettant le passage du type de ses paramètres formels de sortie $n\theta'$ à celui de ses paramètres effectifs de sortie nt', s'en déduit alors. Tout type variable $*_k$ défini dans $n\theta'$, doit l'être également dans $n\theta$ pour permettre au type résultat d'être complètement spécifié. Ainsi, $\Phi'(*_k)$ est défini comme étant à $\Phi(*_k)$

Un unificateur n'est calculable que si le types des paramètres formels est strictement plus général que celui des paramètres effectifs, tout type variable du second n'est unifiable qu'avec un autre type variable. Par exemple un opérateur de type $int \rightarrow int$ ne peut être appliqué à un opérande de type $*_k$.

$$\frac{\begin{array}{c}\alpha,\ \varphi,\ \Theta \overset{\text{type}}{\vdash} exp\_list\ :\ nt \\ \varphi(id)\ =\ n\theta \rightarrow n\theta' \\ \Phi \vdash n\theta,\ nt \\ \Phi' \vdash n\theta',\ nt' \\ \forall k\ (\Phi'(k) \neq \bot \Rightarrow (\Phi'(k) = \Phi(k)))\end{array}}{\alpha,\ \gamma,\ \Theta \overset{\text{type}}{\vdash} id\ (exp\_list)\ :\ n\theta'}$$

Pour tout opérateur *id*, le genre $n\gamma$ de *exp_list* doit être compatible avec sa définition. Le genre du résultat est également fourni par l'environnement φ, il dépend de $n\gamma$ pour la plupart des opérateur prédéfinis, il est fixé pour les autres.

L'opérateur `with`, de par son utilisation principale dans la description d'opérateurs récursifs, a une définition particulière. En fonction de la valeur booléenne de sa première entrée, seul une des deux autres est utilisée.

$$\frac{\begin{array}{c}\alpha,\ \varphi,\ \Theta \overset{\text{type}}{\vdash} exp\ :\ bool \\ \alpha,\ \varphi,\ \Theta \overset{\text{type}}{\vdash} exp'\ :\ t \\ \alpha,\ \varphi,\ \Theta \overset{\text{type}}{\vdash} exp''\ :\ t\end{array}}{\alpha,\ \gamma,\ \Theta \overset{\text{type}}{\vdash} \texttt{with}\ exp\ \texttt{then}\ exp'\ \texttt{else}\ exp''\ :\ nt}$$

L'opérateur `..` construit un tableaux mono-dimensionnel d'entiers dont la taille dépend de la valeur de ses opérandes.

$$\frac{\alpha,\ \varphi,\ \Theta \overset{\text{type}}{\vdash} exp\ :\ int \quad \alpha,\ \varphi,\ \Theta \overset{\text{type}}{\vdash} exp'\ :\ int \quad \alpha,\ \varphi,\ \Theta \overset{\text{type}}{\vdash} exp''\ :\ int}{\alpha,\ \gamma,\ \Theta \overset{\text{type}}{\vdash} exp\ ..\ exp'\ \texttt{step}\ exp''\ :\ (int, 1)}$$

5.2.3 Concaténation de tableaux

L'opérateur de concaténation s'applique uniquement à deux tableaux mono-dimensionnels, ayant des éléments de même type. Ces tableaux sont mis bout à bout pour en former un troisième dont la taille est la somme de celle des deux premiers.

$$\frac{\alpha,\ \varphi,\ \Theta \overset{\text{type}}{\vdash} exp\ :\ (t,1) \quad \alpha,\ \varphi,\ \Theta,\ \mu \overset{\text{type}}{\vdash} exp'\ :\ (t,1)}{\alpha,\ \varphi,\ \Theta,\ \mu \overset{\text{type}}{\vdash} exp\ \mid\ exp'\ :\ (t,\ 1)}$$

5.2.3.1 Extension

L'opérateur d'extension construit un tableau à k dimensions dont tous les éléments sont une copie de *exp*. Les tailles des dimensions successives sont données par *exp_list*.

Cet opérateur ne peut être appliqué à une liste d'expressions.

$$\frac{\alpha,\ \varphi,\ \Theta \overset{\text{type}}{\vdash} exp\ :\ t \quad t \in ID \cup \mathcal{STRUCTYPE} \quad \alpha,\ \varphi,\ \Theta \overset{\text{type}}{\vdash} exp\_list\ :\ int^k}{\alpha,\ \varphi,\ \Theta \overset{\text{type}}{\vdash} exp\hat{}exp\_list\ :\ (t,\ k)}$$

5.2.3.2 Construction

L'opérateur de construction permet de définir par extension un tableau mono-dimensionnel à partir d'une liste de k éléments de même type. L'indice des différents éléments est donné par leur position dans la liste.

Il n'est pas possible de construire des tableaux de listes.

$$\frac{\alpha,\ \varphi,\ \Theta \overset{\text{type}}{\vdash} exp\_list\ :\ t^k) \qquad t \in\ ID\ \cup STRUCTYPE)}{\alpha,\ \varphi,\ \Theta \overset{\text{type}}{\vdash} [\ exp\_list\ ] : (t,\ 1)}$$

5.2.3.3 Sélection

L'opérateur de sélection permet d'extraire un sous tableau d'un tableau à k dimensions. *exp_list* est une liste de k éléments statiques entiers ou tableaux mono-dimensionnels d'entiers.

Chaque élément du tableau source est repéré par k entiers. Si le i^{eme} élément de la suite *exp_list* est un entier n, les éléments sélectionnés auront tous n comme p^{ieme} indice. S'il s'agit d'un tableau t, les éléments sélectionnés auront tous leur p^{ieme} indice égal à la valeur d'un élément de t.

Un simple élément de *exp* est sélectionné si tous les éléments de *exp_list* sont des entiers.

$$\frac{\alpha,\ \varphi,\ \Theta \overset{\text{type}}{\vdash} exp\ :\ (t,\ k) \qquad \alpha,\ \varphi,\ \Theta \overset{\text{type}}{\vdash} exp\_list\ :\ int^k}{\alpha,\ \varphi,\ \Theta \overset{\text{type}}{\vdash} exp[exp\_list]\ :\ t}$$

Dans le cas contraire le résultat est un sous-tableau

$$\frac{\alpha,\ \varphi,\ \Theta \overset{\text{type}}{\vdash} exp\ :\ (t,\ k) \qquad \alpha,\ \varphi,\ \Theta \overset{\text{type}}{\vdash} exp\_list\ :\ (t_1,\ \ldots,t_k) \qquad \forall p < k\ (t_p\ =\ int)\ \vee\ (t_p)\ =\ (int,\ 1)}{\alpha,\ \varphi,\ \Theta \overset{\text{type}}{\vdash} exp[exp\_list]\ :\ (t,\ \Sigma_{p=1}^{k}(t_p\ =\ int))}$$

5.2.4 Listes d'expressions

$$\frac{t \notin TYPELIST \qquad \alpha,\ \varphi,\ \Theta \overset{\text{type}}{\vdash} exp\ :\ t \qquad \alpha,\ \varphi,\ \Theta \overset{\text{type}}{\vdash} exp\_list\ :\ (t_1,\ldots,t_k)}{\alpha,\ \varphi,\ \Theta \overset{\text{type}}{\vdash} exp\ exp\_list\ :\ (t,\ t_1,\ \ldots,\ t_k)}$$

Les listes d'expressions ne sont pas hiérarchiques en LUSTRE, le type d'une liste *exp_list* dont certains de ses éléments sont eux-mêmes des listes est le même que celui la liste obtenu une fois *exp_list* mise à plat.

$$\frac{\alpha,\ \varphi,\ \Theta \overset{\text{type}}{\vdash} exp\ :\ (t_1,\ \ldots,\ t_k) \qquad \alpha,\ \varphi,\ \Theta \overset{\text{type}}{\vdash} exp\_list\ :\ (t'_1,\ \ldots,\ t'_{k'})}{\alpha,\ \varphi,\ \Theta \overset{\text{type}}{\vdash} exp\ exp\_list :\ (t_1,\ \ldots,\ t_k,\ t'_1,\ \ldots,\ t'_{k'})}$$

5.2.5 Déclarations d'un noeud

Dans un noeud, Il y a trois types de déclarations :

- Les paramètres d'entrée.
- Les paramètres de sortie.
- Les variables, constantes et statiques locaux au noeud.

Le type des déclarations de constantes et de statiques, dans un noeud, est donné explicitement et leur définition repoussée dans le système d'équations. Toutes ces déclarations ont ainsi un format similaire.

5.2.5.1 Variables

Toute variable d'un programme LUSTRE est définie à l'intérieur d'un noeud, il n'existe aucune variable globale.

$$\frac{\alpha,\ \varphi,\ \Theta \overset{\text{type}}{\vdash} type\ :\ t \qquad \alpha(id)\ =\ t}{\alpha,\ \varphi,\ \Theta \overset{\text{type}}{\vdash} id\ type\ :\ \alpha(id)}$$

5.2.5.2 Constantes

$$\frac{\alpha,\ \varphi,\ \Theta \overset{\text{type}}{\vdash} type\ :\ t \qquad \alpha(id)\ =\ t}{\alpha,\ \varphi,\ \Theta \overset{\text{type}}{\vdash} \texttt{const}\ id\ type\ :\ \alpha(id)}$$

5.2.5.3 Statiques

$$\frac{\alpha,\ \varphi,\ \Theta \overset{\text{type}}{\vdash} type\ :\ t \qquad \alpha(id)\ =\ t}{\alpha,\ \varphi,\ \Theta \overset{\text{type}}{\vdash} \texttt{static}\ id\ type\ :\ \alpha(id)}$$

5.2.5.4 Liste de variables

$$\frac{\alpha,\ \varphi,\ \Theta,\ \mu \overset{\text{type}}{\vdash} var\ :\ t \qquad \alpha,\ \varphi,\ \Theta,\ \mu \overset{\text{type}}{\vdash} var\_list\ :\ t_1,\ \ldots,\ t_k}{\alpha,\ \varphi,\ \Theta \overset{\text{type}}{\vdash} var\ var\_list :\ (t,\ t_1,\ \ldots,\ t_k)}$$

Chapitre 6

Correction des programmes

6.1 Vérification des programmes

6.1.1 Programme

Un programme LUSTRE est considéré comme correct s'il est possible de définir les quatre environnements α, φ, Θ et μ de telle sorte que le programme vérifie le prédicat *check*.

- α, l'environnement des objets, fournit le type de n'importe quel constante ou statique global du programme. Chaque noeud complète localement cette environnement en fonctions de ses déclarations de variables.
- Θ, l'environnement de types, fournit la représentation canonique des types prédéfinis ($\Theta(int) = int$, $\Theta(bool) = bool$, $\Theta(real) = real$) ou définis par l'utilisateur.
- φ, l'environnement des opérateurs, fournit la nature et la signature des opérateurs prédéfinis ou définis par l'utilisateur.

  Par exemple,

  $$\varphi(or) = (pred,\ ((bool\ /\ 0,\ bool\ /\ 1),\ n\gamma)\ \rightarrow\ ((bool\ /\ 0),\ (\bigsqcup(n\gamma)\ /\ 0)))$$

  est la définition de l'opérateur booléen à deux entrées `or` ; il s'agit d'un opérateur prédéfini qui reçoit en entrée une liste de deux booléens, ayant un genre quelconque et produit en sortie un booléen dont le genre est la borne supérieure du genre des entrées.

  Le genre de la plupart des opérateurs est défini ainsi, seul les opérateurs temporels et `with` font exception.

  - `with` est un opérateur dont la condition est évaluée à la compilation, elle doit donc être une expression statique.
  - Les opérateurs temporels ne peuvent être employés dans des expressions statiques ou constantes, leur résultat est donc toujours de genre **var**.

  voici la définition de φ_{pred}, l'environnement φ restreint aux seuls opérateur prédéfinis.

$$
\begin{array}{|llllllllll|}
\hline
not & ,n\nu & \rightarrow & (pred, & (bool/0, & n\gamma &) \rightarrow & (bool/0, & \bigsqcup(n\gamma)/0)) & ,\lambda n.\bot \\
b\_op2 & ,n\nu & \rightarrow & (pred, & (bool/0,\ bool/1, & n\gamma &) \rightarrow & (bool/0, & \bigsqcup(n\gamma)/0)) & ,\lambda n.\bot \\
\# & ,n\nu & \rightarrow & (pred, & (bool^+, & n\gamma &) \rightarrow & (bool/n, & \bigsqcup(n\gamma)/0) & ,\lambda n.\bot \\
\% & ,n\nu & \rightarrow & (pred, & (int/0,\ int/1, & n\gamma &) \rightarrow & (int/0, & \bigsqcup(n\gamma)/0)) & ,\lambda n.\bot \\
umoins & ,n\nu & \rightarrow & (pred, & (int/0, & n\gamma &) \rightarrow & (int/0, & \bigsqcup(n\gamma)/0)) & ,\lambda n.\bot \\
ir\_op2 & ,n\nu & \rightarrow & (pred, & (int/0,\ int/1, & n\gamma &) \rightarrow & (int/0, & \bigsqcup(n\gamma)/0)) & ,\lambda n.\bot \\
ir\_op2 & ,n\nu & \rightarrow & (pred, & (real/0,\ real/1, & n\gamma &) \rightarrow & (real/0, & \bigsqcup(n\gamma)/0)) & ,\lambda n.\bot \\
pow & ,n\nu & \rightarrow & (pred, & (int/0,\ int/1, & n\gamma &) \rightarrow & (int/0, & \bigsqcup(n\gamma)/0)) & ,\lambda n.\bot \\
ir2\_op2 & ,n\nu & \rightarrow & (pred, & (real/0,\ int/1, & n\gamma &) \rightarrow & (real/0, & \bigsqcup(n\gamma)/0)) & ,\lambda n.\bot \\
p\_op2 & ,n\nu & \rightarrow & (pred, & (*_k/0,\ *_k/1, & n\gamma &) \rightarrow & (*_k/0, & \bigsqcup(n\gamma)/0)) & ,\lambda n.\bot \\
if & ,n\nu & \rightarrow & (pred, & (bool/0,\ *_k/1,\ *_k/2, & n\gamma &) \rightarrow & (*_k/0, & \bigsqcup(n\gamma)/0)) & ,\lambda n.\bot \\
with & ,n\nu & \rightarrow & (pred, & (bool/0,\ *_k/1,\ *_p/2, & n\gamma[static/0] &) \rightarrow & (\lambda n.\bot & \lambda n.\bot) & ,\lambda n.\bot \\
t\_op1 & ,n\nu & \rightarrow & (pred, & (*_k/0, & n\gamma &) \rightarrow & (*_k/0, & var/0\ ) & ,\lambda n.\bot \\
fby & ,n\nu & \rightarrow & (pred, & (*_k/0,\ *_k/1, & n\gamma &) \rightarrow & (*_k/0, & var/0\ ) & ,\lambda n.\bot \\
when & ,n\nu & \rightarrow & (pred, & (*_k/0,\ bool/1, & n\gamma &) \rightarrow & (*_k/0, & var/0\ ) & ,\lambda n.\bot \\
.. & ,n\nu & \rightarrow & (pred, & (int/0,\ int/1,\ int/2 & static/0 &) \rightarrow & (\lambda n.\bot, & static) & ,\lambda n.\bot \\
 & & & & & static/1 & & & & \\
 & & & & & static/2 & & & & \\
\hline
\end{array}
$$

$b\_op2 \in \{and,\ or,\ xor\}$
$ir\_op2 \in \{plus,\ moins,\ mul,\ div,\ sup,\ inf,\ supeg,\ infeg\}$
$p\_op2 \in \{egal,\ dif\}$
$t\_op1 \in \{pre,\ current\}$

Le type, le genre et la valeur du résultat d'un opérateur `with` est indéfini, il est calculé et non tiré de l'environnement. La type et la valeur du u résultat d'un opérateur `..` sont indéfinis pour la même raison.

- μ l'environnement des valeurs, qui fournit la valeur associée à tous les identificateurs de statiques.

Puisque nous avons choisi de donner une sémantique naturelle de LUSTRE, la définition des prédicats ne va donner aucune information sur le moyen de construire ces environnements. A l'inverse, les quatre environnements vont être supposés connus et les règles vont permettre, à l'aide des prédicats, de définir des propriétés que doivent vérifier les environnements. Un programme sera donc correct si l'ensemble des propriétés qu'il induit sur les environnements n'est pas contradictoire.

La sémantique décrite en utilisant cette approche sera peut être moins complète que ce que l'on aurait pu obtenir en donnant une sémantique opérationnelle, mais en revanche elle va permettre de donner une description plus simple et compréhensible du langage.

Un programme LUSTRE est considéré être correct si l'ensemble de ses déclarations d'objets l'est et si les différentes instanciations (au sens n'ayant pas les mêmes entrées statiques) de ses opérateurs le sont également dans les mêmes environnements.

$$\frac{\alpha,\ \varphi, \Theta,\ \mu \overset{\text{check}}{\vdash} decl_{obj}\_list \qquad \alpha,\ \varphi, \Theta,\ \mu \overset{\text{check}}{\vdash} decl_{op}\_list}{\alpha,\ \varphi,\ \Theta \overset{\text{check}}{\vdash} \texttt{prog}\ id\ decl_{obj}\_list\ decl_{op}\_list}$$

6.1.2 Constantes

6.1.2.1 Constante externe

La déclaration d'une constante externe associe à l'identificateur *id* :

- Le type τ, la représentation canonique de *type*.
- Le genre const.

Le type des constantes externes n'est pas forcément élémentaire, il est tout à fait possible en LUSTRE de déclarer des tableaux externes de constantes. Cette règle assure juste dans ce cas que leurs bornes sont calculables.

$$\frac{\alpha,\ \varphi,\ \Theta,\ \mu \overset{\text{type}}{\vdash} type : \tau \qquad \alpha(id) = (\tau,\ const)}{\alpha,\ \varphi,\ \Theta,\ \mu \overset{\text{check}}{\vdash} \texttt{const}\ id\ type}$$

6.1.2.2 Constante locale

Le type d'une constante locale n'est pas donné explicitement mais déduit de son expression de définition.

Un identificateur de constante ne peut être associé qu'à une certaine catégorie d'expression. *exp* ne peut désigner une liste d'expressions et doit être construite uniquement à partir de constantes et de statiques.

$$\frac{\alpha,\ \varphi,\ \Theta,\ \mu \overset{\text{type}}{\vdash} exp : \tau,\ \gamma \qquad \alpha(id) = (\tau,\ const) \qquad \tau \in ID \cup \mathcal{STRUCTYPE} \qquad \bot < \gamma \leq const}{\alpha,\ \varphi,\ \Theta,\ \mu \overset{\text{check}}{\vdash} \texttt{const}\ id\ exp}$$

6.1.3 Statiques

La valeur d'un statique n'est calculable que si son type a pu être défini.

$$\frac{\begin{array}{c}\alpha,\ \varphi,\ \Theta,\ \mu \overset{\text{type}}{\vdash} exp\ :\ \tau,\ static \\ \alpha(id) = (\tau,\ static) \\ \tau\ \in\ \mathcal{ID}\ \cup \mathcal{STRUCTYPE} \\ \alpha,\ \varphi,\ \Theta,\ \mu \overset{\text{val}}{\vdash} exp : \mu(id)\end{array}}{\alpha,\ \varphi,\ \Theta \overset{\text{check}}{\vdash} \texttt{static}\ id\ exp}$$

6.1.4 Types

6.1.4.1 Types externe

Un type externe est traité comme un nom, sa structure étant inconnue.

$$\frac{\Theta(id) = id}{\alpha,\ \varphi,\ \Theta,\ \mu \overset{\text{check}}{\vdash} \texttt{type}\ id}$$

6.1.4.2 Types locaux

La déclaration d'un type local permet d'associer dans l'environnement des types Θ à l'identificateur *id* la représentation canonique τ de *type*.

Les types des listes ne sont pas dénotables, il n'est pas possible de les nommer.

$$\frac{\begin{array}{c}\alpha,\ \varphi,\ \Theta,\ \mu \overset{\text{type}}{\vdash} type\ :\ \Theta(id) \\ \Theta(id)\ \in\ \mathcal{ID}\ \cup \mathcal{STRUCTYPE}\end{array}}{\alpha,\ \varphi,\ \Theta,\ \mu \overset{\text{check}}{\vdash} \texttt{type}\ id\ type}$$

6.1.5 Noeuds

6.1.5.1 Noeuds locaux

L'environnement d'objets α d'un noeud se décompose en deux sous-environnements :

- $g\alpha$ associant son type à tout identificateur de variable, constante ou statique global
- $l\alpha$ associant son type à tout identificateur de variable, constante ou statique déclaré dans le noeud local.

Ces deux sous-environnements sont disjoints, un même identificateur ne pouvant être utilisé pour désigner à la fois un objet global et un autre local (pas de structure de bloc).

L'environnement de valeurs se décompose lui en quatre sous-environnements disjoints $g\mu$ l'environnement des valeurs des objects globaux et trois environnement pour les valeurs des différent statiques définis dans le noeud.

- $i\mu$ celui des valeurs des entrées statiques.
- $o\mu$ celui des valeurs des sorties statiques.
- $l\mu$ celui des valeurs des statiques locaux.

Un noeud est correct si chacune des instanciations qui en est faite dans le programme l'est. Une instanciation d'un noeud est définie par une liste de valeurs de ses entrées statiques qui est fournie par l'environnement avec la liste des valeurs des sorties statiques correspondantes.

Les types des paramètres d'entrée ne doivent être définis qu'en fonction des entrées statiques, ceux des paramètres de sortie ou des locaux en fonction des entrées et des sorties statiques. Les types des paramètres ne peuvent pas dépendre des valeurs des statiques locaux utilisables uniquement à l'intérieur du système d'équation. Les dépendances ne peuvent également dépendre des valeurs des statiques locaux.

La valeurs des locaux et des sorties statiques sont définies par l'évaluation statique du système d'équation. Les valeurs de ces dernier doivent correspondre avec celle spécifiées par l'environnement.

$$\frac{\forall n\nu\ (\varphi(id, n\nu) = loc,\ (n\tau,\ n\gamma)\ \rightarrow\ (n\tau',\ n\gamma'),\ n\nu') \quad \exists l\alpha, l\mu \left(\begin{array}{c} l\alpha,\ \varphi,\ \Theta,\ g\mu[i\mu] \overset{\text{type}}{\vdash} var\_list : n\tau,\ n\gamma \\ l\alpha,\ \varphi,\ \Theta,\ g\mu[i\mu[o\mu]] \overset{\text{type}}{\vdash} var\_list' : n\tau',\ n\gamma' \\ l\alpha,\ \varphi,\ \Theta,\ g\mu[i\mu[o\mu]] \overset{\text{type}}{\vdash} var\_list'' : n\tau'',\ n\gamma'' \\ \forall id\ (g\alpha(id) = \bot \vee l\alpha(id) = \bot) \\ i\mu \overset{\text{val}}{\vdash} var\_list : n\nu \\ o\mu \overset{\text{val}}{\vdash} var\_list' : n\nu' \\ g\alpha[l\alpha],\ \varphi,\ \Theta,\ \mu[i\mu[o\mu]] \overset{\text{check}}{\vdash} dep\_list \\ g\alpha[l\alpha],\ \varphi,\ \Theta,\ \mu[i\mu[o\mu[l\mu]]] \overset{\text{check}}{\vdash} eq\_list \\ \forall id\ (i\mu(id) = \bot \vee o\mu(id) = \bot) \\ \forall id\ (i\mu(id) = \bot \vee l\mu(id) = \bot) \\ \forall id\ (o\mu(id) = \bot \vee l\mu(id) = \bot) \\ g\alpha[l\alpha],\ \varphi,\ \Theta,\ \mu[i\mu[o\mu[l\mu]]] \overset{\text{val}}{\vdash} eq\_list \end{array} \right)}{g\alpha,\ \varphi,\ \Theta,\ \mu \overset{\text{check}}{\vdash} \texttt{node}\ id\ \texttt{in}\ var\_list\ \texttt{out}\ var\_list'\ \texttt{loc}\ var\_list''\ dep\_list\ eq\_list}$$

6.1.5.2 noeuds externe

La déclaration d'un noeud externe ne comportant pas de système d'équations, les vérifications assurent simplement que ses déclarations et ses dépendances sont correctement définies. Une contrainte supplémentaire est ajoutée par rapport aux noeuds localement définis : aucun de ses paramètres de sortie ne peut être statique, leur valeur en effet ne serait pas évaluable par le compilateur. En revanche certains de ses paramètres d'entrée peuvent être statiques si la compilation du noeud externe peut être réalisée sans connaître leur valeur.

$$\frac{\forall n\nu\ (\varphi(id, n\nu) = loc,\ (n\tau,\ n\gamma)\ \rightarrow\ (n\tau',\ n\gamma'),\ \lambda n.\ \bot) \quad \exists l\alpha \left(\begin{array}{c} l\alpha,\ \varphi,\ \Theta,\ g\mu[i\mu] \overset{\text{type}}{\vdash} var\_list : n\tau,\ n\gamma \\ l\alpha,\ \varphi,\ \Theta,\ g\mu[i\mu[o\mu]] \overset{\text{type}}{\vdash} var\_list' : n\tau',\ n\gamma' \\ \forall id\ (g\alpha(id) = \bot \vee l\alpha(id) = \bot) \\ i\mu \overset{\text{val}}{\vdash} var\_list : n\nu \\ g\alpha[l\alpha],\ \varphi,\ \Theta,\ \mu[i\mu] \overset{\text{check}}{\vdash} dep\_list \\ \forall n\ (n\gamma'(n) \neq static) \end{array} \right)}{g\alpha,\ \varphi,\ \Theta,\ \mu \overset{\text{check}}{\vdash} \texttt{node}\ id\ \texttt{in}\ var\_list\ \texttt{out}\ var\_list'\ dep\_list}$$

6.1.6 Fonctions

Les fonctions sont juste des opérateurs externes ayant la particularité d'être purement combinatoires, c'est à dire que chaque sortie dépend sans retard de toutes les entrées. Les limitations s'appliquant aux opérateurs externes s'appliquent également aux fonctions.

$$\frac{\forall n\nu\ (\varphi(id, n\nu) = loc,\ (n\tau,\ n\gamma)\ \rightarrow\ (n\tau',\ n\gamma'),\ \lambda n.\ \bot) \quad \exists l\alpha \left(\begin{array}{c} l\alpha,\ \varphi,\ \Theta,\ g\mu[i\mu] \overset{\text{type}}{\vdash} var\_list : n\tau,\ n\gamma \\ l\alpha,\ \varphi,\ \Theta,\ g\mu[i\mu[o\mu]] \overset{\text{type}}{\vdash} var\_list' : n\tau',\ n\gamma' \\ \forall id\ (g\alpha(id) = \bot \vee l\alpha(id) = \bot) \\ i\mu \overset{\text{val}}{\vdash} var\_list : n\nu \\ g\alpha[l\alpha],\ \varphi,\ \Theta,\ \mu[i\mu] \overset{\text{check}}{\vdash} dep\_list \\ \forall n\ (n\gamma'(n) \neq static) \end{array} \right)}{g\alpha,\ \varphi,\ \Theta,\ \mu \overset{\text{check}}{\vdash} \texttt{node}\ id\ \texttt{in}\ var\_list\ \texttt{out}\ var\_list'\ dep\_list}$$

6.1.7 Déclaration vide

Une déclaration vide est toujours correcte.

$$\alpha,\ \varphi, \Theta,\ \mu \overset{\text{check}}{\vdash} \varepsilon$$

6.1.8 Liste de déclarations d'objets

$$\frac{\alpha,\ \varphi,\Theta,\ \mu \overset{\text{check}}{\vdash} decl \qquad \alpha,\ \varphi,\Theta',\ \mu \overset{\text{check}}{\vdash} decl_{obj}\_list}{\alpha,\ \varphi,\ \Theta,\ \mu \overset{\text{check}}{\vdash} decl\ decl_{obj}\_list}$$

6.1.9 Liste de déclarations d'opérateurs

$$\frac{\alpha,\ \varphi,\Theta,\ \mu \overset{\text{check}}{\vdash} decl \qquad \alpha,\ \varphi,\Theta',\ \mu \overset{\text{check}}{\vdash} decl_{obj}\_list}{\alpha,\ \varphi,\ \Theta,\ \mu \overset{\text{check}}{\vdash} decl\ decl_{obj}\_list}$$

6.1.10 Dépendances

Une dépendance sert à indiquer que les résultats de différents calculs sont nécessaires à un autre. Le type de la partie gauche de la dépendance et celui de chaque élément de la liste en partie droite doivent être identiques. De plus, l'évaluation d'un statique ne peut dépendre de celle d'une constante ou d'une variable, l'évaluation d'une constante de celle d'une variable.

$$\frac{\begin{array}{c}\alpha,\ \varphi,\ \Theta,\ \mu \overset{\text{type}}{\vdash} left\ :\ \tau,\ \gamma \\ \alpha,\ \varphi,\ \Theta,\ \mu \overset{\text{type}}{\vdash} exp\_list\ :\ n\tau,\ n\gamma \\ \forall n\ (n\tau(n)\ =\ \tau) \\ \forall n\ (n\gamma(n)\ \leq\ \gamma)\end{array}}{\alpha,\ \varphi,\ \Theta \overset{\text{check}}{\vdash} left\_list\ \textbf{needs}\ exp\_list'}$$

Un ensemble de dépendances est correctement typé si chacune d'entre-elles l'est.

$$\frac{\alpha,\ \varphi,\ \Theta,\ \mu \overset{\text{check}}{\vdash} dep \qquad \alpha,\ \varphi,\ \Theta,\ \mu \overset{\text{check}}{\vdash} dep\_list}{\alpha,\ \varphi,\ \Theta,\ \mu \overset{\text{check}}{\vdash} dep\ dep\_list}$$

6.1.11 Equations

Une équation est correctement typée si ses deux membres sont de même type et leur genres compatibles, un statique ne peut être défini en fonction d'une constante ou d'une variable et un constante en fonction d'une variable.

Une équation se compose d'une liste d'objets en partie gauche et d'une liste d'expression, donnant leur définition, en partie droite.

$$\frac{\alpha,\ \varphi,\ \Theta,\ \mu \overset{\text{type}}{\vdash} left\_list\ :\ n\tau,\ n\gamma \qquad \alpha,\ \varphi,\ \Theta,\ \mu \overset{\text{type}}{\vdash} exp\_list\ :\ n\tau,\ n\gamma' \qquad \forall n\ (n\gamma'(n)\ \leq\ n\gamma(n))}{\alpha,\ \varphi,\ \Theta,\ \mu \overset{\text{check}}{\vdash} left\_list = exp\_list}$$

La sémantique de la partie gauche d'une équation n'étant qu'un cas particulier de celle des expressions, les règles la décrivant ne seront pas données.

6.1.12 Assertions

Une assertion est correctement typée si le type de *exp* est dérivé de *bool*, c'est à dire s'il est obtenu en appliquant un certain nombre de fois les constructeurs de types au type *bool*.

De façon générale, nous dirons qu'un type τ est dérivé d'un type τ' si le premier peut être obtenu à partir du second en lui appliquant un certain nombre de fois les constructeurs de types. Par exemple, `(int^5)^4` et `((int^5)^(4,3))^12` sont deux types dérivés `(int^5)`.

Une définition complète de cette notion de type dérivé est donné par le prédicat *derived_type*

$$\overset{\text{derived\_type}}{\vdash}\ (\tau,\ \tau')$$

est vérifié si τ est un type dérivé de τ'

La dérivation de type la plus simple dérive un type en lui même

$$\overset{\text{derived\_type}}{\vdash}\ (\tau,\ \tau)$$

Un type est dérivé d'un type structuré si ses éléments le sont également

$$\frac{\overset{\text{derived\_type}}{\vdash}\ (\tau,\ \tau')}{\overset{\text{derived\_type}}{\vdash}\ (\tau,\ (\tau',\ \delta))}$$

La règle définissant qu'une assertion est correctement typée est alors :

$$\frac{\alpha,\ \varphi,\ \Theta,\ \mu \overset{\text{type}}{\vdash} exp\ :\ \tau,\ n\gamma \qquad \overset{\text{derived\_type}}{\vdash}\ (bool, \tau)}{\alpha,\ \varphi,\ \Theta,\ \mu \overset{\text{check}}{\vdash}\ \texttt{assert}\ exp}$$

Pour une assertion statique, on impose en plus que le genre soit statique, et que sa valeur soit *true* ou dérivée de *true*. Leur rôle est de donner des contraintes sur les valeurs des statiques

d'un noeud, de façon à restreindre ses instanciations possibles. Une valeur fausse signifie que l'instanciation est interdite.

La notion de valeur dérivée se définit de façon similaire à celle de type dérivé :

$$\overset{\text{derived\_val}}{\vdash} \quad (\nu,\ \nu)$$

$$\frac{\forall n((n\nu(n) \neq \bot)) \Rightarrow \overset{\text{derived\_val}}{\vdash} \quad (\nu,\ \nu(n))}{\overset{\text{derived\_val}}{\vdash} \quad (\nu,\ n\nu')}$$

$$\frac{\begin{array}{c}\alpha,\ \varphi,\ \Theta,\ \mu \overset{\text{type}}{\vdash} exp\ :\ \tau,\ static \\ \overset{\text{derived\_type}}{\vdash} \quad (bool, \tau) \\ \alpha,\ \varphi,\ \Theta,\ \mu \overset{\text{type}}{\vdash} exp : \nu \\ \overset{\text{derived\_val}}{\vdash} \quad (true,\ \nu)\end{array}}{\alpha,\ \varphi,\ \Theta,\ \mu \overset{\text{check}}{\vdash} \texttt{static assert}\ exp}$$

6.1.13 Liste d'équations

Un ensemble d'équations est correctement typé si chacune d'entre elle l'est.

$$\frac{\begin{array}{c}\alpha,\ \varphi,\ \Theta,\ \mu \overset{\text{check}}{\vdash} eq \\ \alpha,\ \varphi,\ \Theta,\ \mu \overset{\text{check}}{\vdash} eq\_list\end{array}}{\alpha,\ \varphi,\ \Theta,\ \mu \overset{\text{check}}{\vdash} eq\ eq\_list}$$

6.2 Calcul des types

6.2.1 Types

Les règles suivantes définissent la construction du représentant canonique d'un type.

Le type associé à un identificateur *id* déclaré est fourni par l'environnement des types Θ

$$\frac{\Theta(id) \neq \bot}{\alpha,\ \Theta \overset{\text{type}}{\vdash} id\ :\ \Theta(id)}$$

$$\Theta \overset{\text{type}}{\vdash} *_k\ :\ *_k$$

Les bornes d'un tableau doivent forcément être connues au moment de la compilation. Le type d'un tableau à k dimensions est donc défini à partir du type τ de ses éléments et de k expressions statiques, les tailles des dimensions successives.

$$\frac{\begin{array}{c}\alpha,\ \varphi,\ \Theta,\ \mu \overset{\text{type}}{\vdash} type\ :\ \tau \\ \alpha,\ \varphi,\ \Theta,\ \mu \overset{\text{type}}{\vdash} exp\_list\ :\ \lambda p.\ (si\ p < k\ alors\ int\ sinon\ \bot),\ \lambda p.\ (si\ p < k\ alors\ static\ sinon\ \bot) \\ \alpha,\ \varphi,\ \Theta,\ \mu \overset{\text{val}}{\vdash} exp\_list\ :\ n\nu \\ \forall p(n\nu(p) \neq \bot \Leftrightarrow p < k)\end{array}}{\alpha,\ \varphi,\ \Theta,\ \mu \overset{\text{type}}{\vdash} type\hat{}exp\_list\ :\ (\tau,\ n\nu)}$$

6.2.2 Opérateurs

Un opérateur *id* est appliqué à une liste d'expressions de type $n\tau$ et en produit une autre en sortie de type $n\tau'$. Ces types sont déterminés grâce à l'environnement des fonctions, une fois les valeurs des entrées statiques déterminée.

Dans le cas d'un opérateur non polymorphe, l'application de *id* à *exp_list* est correcte si la liste des types des paramètres formels d'entrée est égale à $n\tau$.

$$\frac{\begin{array}{c}\alpha,\ \varphi,\ \Theta,\ \mu \overset{\text{type}}{\vdash} exp\_list\ :\ n\tau,\ n\gamma \\ \alpha,\ \varphi,\ \Theta,\ \mu \overset{\text{val}}{\vdash} exp\_list\ :\ n\nu \\ \varphi(id,\ n\nu)\ =\ \iota,\ (n\tau,\ n\gamma)\ \rightarrow\ (n\tau',\ n\gamma'),n\nu'\end{array}}{\alpha,\ \gamma,\ \Theta,\ \mu \overset{\text{type}}{\vdash} id\ (exp\_list)\ :\ n\tau',\ n\gamma'}$$

Le cas d'un opérateur polymorphe est plus complexe et généralise le précédent, puisque $n\theta$ ne désigne plus un type unique mais un ensemble de types. L'opérateur *id* peut être appliqué à n'importe quelle liste d'expressions s'il est possible d'unifier son type $n\tau$ avec celui des paramètres formels $n\theta$.

L'unificateur, noté Φ, est un ensemble de couples $(*_k,\ \tau)$ associant à tout type variable $*_k$, défini dans $n\theta$, un type τ élémentaire ou un autre type variable $*_p$ par lequel il faut remplacer toute ses occurrences dans $n\theta$ pour obtenir $n\tau$.

Φ est défini comme une fonction des types dans les types. Le prédicat suivant indique que deux types θ et τ peuvent être unifiés en utilisant Φ comme unificateur.

$$\Phi \overset{\text{unify}}{\vdash} \theta,\ \tau$$

Un type variable $*_k$ s'unifie avec un type élémentaire (dont la représentation canonique est un identificateur) ou $*_p$. Toutes les occurrences de $*_k$ doivent être unifiées avec le même type.

$$\frac{\Phi(k) = id}{\Phi \overset{\text{unify}}{\vdash} *_k,\ id}$$

$$\frac{\Phi(k) = *p}{\Phi \overset{\text{unify}}{\vdash} *_k,\ *_p}$$

deux types identiques s'unifient

$$\Phi \overset{\text{unify}}{\vdash} \tau,\ \tau$$

Deux tableaux s'unifient seulement s'ils ont les même dimensions et si les types de leurs éléments s'unifient

$$\frac{\Phi \overset{\text{unify}}{\vdash} \theta,\ \tau}{\Phi \overset{\text{unify}}{\vdash} (\theta,\ \delta),\ (\tau,\ \delta)}$$

Deux listes s'unifient seulement si leurs éléments s'unifient deux à deux.

$$\frac{\forall n\ \Phi \overset{\text{unify}}{\vdash} n\theta(n),\ n\tau(n)}{\Phi \overset{\text{unify}}{\vdash} n\theta,\ n\tau}$$

$$\Phi \overset{\text{unify}}{\vdash} \bot,\ \bot$$

Pour pouvoir déterminer le type résultat d'une instanciation d'opérateur polymorphe, il est nécessaire de déterminer l'unificateur Φ permettant le passage du type de ses paramètres formels d'entrée $n\theta$ à celui de ses paramètres effectifs d'entrée $n\tau$. L'unificateur Φ', permettant le passage du type de ses paramètres formels de sortie $n\theta'$ à celui de ses paramètres effectifs de sortie $n\tau'$, s'en déduit alors. Tout type variable $*_k$ défini dans $n\theta'$, doit l'être également dans $n\theta$ pour permettre au type résultat d'être complètement spécifié. Ainsi, $\Phi'(*_k)$ est défini comme étant à $\Phi(*_k)$

Un unificateur n'est calculable que si le types des paramètres formels est strictement plus général que celui des paramètres effectifs, tout type variable du second n'est unifiable qu'avec un autre type variable. Par exemple un opérateur de type $int \rightarrow int$ ne peut être appliqué à un opérande de type $*_k$.

$$\frac{\begin{array}{c}\alpha,\ \varphi,\ \Theta,\ \mu \overset{\text{type}}{\vdash} exp\_list\ :\ n\tau,\ n\gamma \\ \alpha,\ \varphi,\ \Theta,\ \mu \overset{\text{val}}{\vdash} exp\_list\ :\ n\nu \\ \varphi(id,\ n\nu) = \iota,\ (n\theta,\ n\gamma) \rightarrow (n\theta',\ n\gamma') \\ \Phi \vdash n\theta,\ n\tau \\ \Phi' \vdash n\theta',\ n\tau' \\ \forall k\ (\Phi'(k) \neq \bot \Rightarrow (\Phi'(k) = \Phi(k)))\end{array}}{\alpha,\ \gamma,\ \Theta,\ \mu \overset{\text{type}}{\vdash} id\ (exp\_list)\ :\ n\theta',\ n\gamma'}$$

Pour tout opérateur *id*, le genre $n\gamma$ de *exp_list* doit être compatible avec sa définition. Le genre du résultat est également fourni par l'environnement φ, il dépend de $n\gamma$ pour la plupart des opérateur prédéfinis, il est fixé pour les autres.

L'opérateur **with**, de par son utilisation principale dans la description d'opérateurs récursifs, a une définition particulière. En fonction de la valeur booléenne de sa première entrée, seul une des deux autres est utilisée.

$$\frac{\begin{array}{c}\alpha,\ \varphi,\ \Theta,\ \mu \overset{\text{type}}{\vdash} exp\ :\ bool,\ static \\ \alpha,\ \varphi,\ \Theta,\ \mu \overset{\text{val}}{\vdash} exp\ :\ true \\ \alpha,\ \varphi,\ \Theta,\ \mu \overset{\text{type}}{\vdash} exp'\ :\ \tau,\ \gamma\end{array}}{\alpha,\ \gamma,\ \Theta,\ \mu \overset{\text{type}}{\vdash} \texttt{with}\ exp\ \texttt{then}\ exp'\ \texttt{else}\ exp''\ :\ n\tau,\ n\gamma}$$

$$\frac{\begin{array}{c}\alpha,\ \varphi,\ \Theta,\ \mu \overset{\text{type}}{\vdash} exp\ :\ bool,\ static \\ \alpha,\ \varphi,\ \Theta,\ \mu \overset{\text{val}}{\vdash} exp\ :\ false \\ \alpha,\ \varphi,\ \Theta,\ \mu \overset{\text{type}}{\vdash} exp''\ :\ \tau,\ \gamma\end{array}}{\alpha,\ \gamma,\ \Theta,\ \mu \overset{\text{type}}{\vdash} \texttt{with}\ exp\ \texttt{then}\ exp'\ \texttt{else}\ exp''\ :\ n\tau,\ n\gamma}$$

L'opérateur `..` construit un tableaux mono-dimensionnel d'entiers dont la taille dépend de la valeur de ses opérandes.

$$\frac{\begin{array}{c}\alpha,\ \varphi,\ \Theta,\ \mu \overset{\text{type}}{\vdash} exp\ :\ int,\ static \\ \alpha,\ \varphi,\ \Theta,\ \mu \overset{\text{type}}{\vdash} exp'\ :\ int,\ static \\ \alpha,\ \varphi,\ \Theta,\ \mu \overset{\text{type}}{\vdash} exp''\ :\ int,\ static \\ \alpha,\ \varphi,\ \Theta,\ \mu \overset{\text{val}}{\vdash} exp\ :\ k \\ \alpha,\ \varphi,\ \Theta,\ \mu \overset{\text{val}}{\vdash} exp'\ :\ k' \\ \alpha,\ \varphi,\ \Theta,\ \mu \overset{\text{val}}{\vdash} exp''\ :\ k''\end{array}}{\alpha,\ \gamma,\ \Theta,\ \mu \overset{\text{type}}{\vdash} exp\ ..\ exp'\ \texttt{step}\ exp''\ :\ (int, (\mid k - k' \mid /k'') + 1),\ static}$$

6.2.3 Concaténation de tableaux

L'opérateur de concaténation s'applique uniquement à deux tableaux mono-dimensionnels, ayant des éléments de même type. Ces tableaux sont mis bout à bout pour en former un troisième dont la taille est la somme de celle des deux premiers.

$$\frac{\begin{array}{c}\alpha,\ \varphi,\ \Theta,\ \mu \overset{\text{type}}{\vdash} exp\ :\ (\tau, k\ /\ 0),\ \gamma \\ \alpha,\ \varphi,\ \Theta,\ \mu \overset{\text{type}}{\vdash} exp'\ :\ (\tau, k'\ /\ 0),\ \gamma'\end{array}}{\alpha,\ \varphi,\ \Theta,\ \mu \overset{\text{type}}{\vdash} exp\ \mid\ exp'\ :\ (\tau,\ k\ +\ k'\ /\ 0),\ \sqcup(\gamma,\ \gamma')}$$

6.2.3.1 Extension

L'opérateur d'extension construit un tableau à k dimensions dont tous les éléments sont une copie de *exp*. Les tailles des dimensions successives sont données par *exp_list*.

Cet opérateur ne peut être appliqué à une liste d'expressions.

$$\frac{\begin{array}{c}\alpha,\ \varphi,\ \Theta,\ \mu \overset{\text{type}}{\vdash} exp\ :\ \tau \\ \tau\ \in\ ID \cup \mathcal{STRUCTYPE} \\ \alpha,\ \varphi,\ \Theta,\ \mu \overset{\text{type}}{\vdash} exp\_list\ :\ \lambda p.\ (si\ p < k\ alors\ int\ sinon\ \bot),\ \lambda p.\ (si\ p < k\ alors\ static\ sinon\ \bot) \\ \alpha,\ \varphi,\ \Theta,\ \mu \overset{\text{val}}{\vdash} exp\_list\ :\ n\nu\end{array}}{\alpha,\ \varphi,\ \Theta,\ \mu \overset{\text{type}}{\vdash} exp\hat{}exp\_list\ :\ (\tau,\ n\nu),\ \gamma}$$

6.2.3.2 Construction

L'opérateur de construction permet de définir par extension un tableau mono-dimensionnel à partir d'une liste de k éléments de même type. L'indice des différents éléments est donné par leur position dans la liste.

Il n'est pas possible de construire des tableaux de listes.

$$\frac{\begin{array}{c}\alpha,\ \varphi,\ \Theta,\ \mu \overset{\text{type}}{\vdash} exp\_list\ :\ n\tau,\ n\gamma \\ \tau\ \in\ ID \cup \mathcal{STRUCTYPE}) \\ \forall p < k\ (n\tau(p)\ =\ \tau) \\ \forall p \geq k\ (n\tau(p) = \bot)\end{array}}{\alpha,\ \varphi,\ \Theta,\ \mu \overset{\text{type}}{\vdash} [\ exp\_list\ ] : (\tau,\ k/0),\ \bigsqcup(n\gamma)}$$

6.2.3.3 Sélection

L'opérateur de sélection permet d'extraire un sous tableau d'un tableau à k dimensions. *exp_list* est une liste de k éléments statiques entiers ou tableaux mono-dimensionnels d'entiers.

Chaque élément du tableau source est repéré par k entiers. Si le i^{eme} élément de la suite *exp_list* est un entier n, les éléments sélectionnés auront tous n comme p^{ieme} indice. S'il s'agit d'un tableau t, les éléments sélectionnés auront tous leur p^{ieme} indice égal à la valeur d'un élément de t.

Pour définir l'opérateur de sélection, deux prédicats intermédiaires sont nécessaires. Le premier, *bound* est vérifié si l'expression de sélection est correcte.

$$\overset{\text{bound}}{\vdash} \; n\nu, \delta$$

$n\nu$ est la valeur résultant de l'évaluation de l'expression de sélection et δ désigne les bornes du tableau source.

La vérification est réalisée pour chaque dimension

$$\frac{\forall n \overset{\text{bound}}{\vdash} n\nu(n),\ \delta(n)}{\overset{\text{bound}}{\vdash} n\nu,\ \delta}$$

Tout k-uplet d'indices construit par cette expression doit désigner un élément du tableau source, la valeur de son p^{ieme} indice doit être comprise entre 0 et la taille de la p^{ieme} dimension diminuée de 1 de ce tableau.

$$\overset{\text{bound}}{\vdash} \; \bot, \bot$$

$$\frac{0 \leq p < k}{\overset{\text{bound}}{\vdash} p,\ k}$$

$$\frac{\forall n\ (0 \leq n\nu(n) < k)}{\overset{\text{bound}}{\vdash} n\nu,\ k}$$

Le second prédicat, *sel type* définit les valeurs des bornes du tableau résultat.

$$\overset{\text{sel type}}{\vdash} \; n\tau,\ \delta\ :\ \delta'$$

Il est vérifié si expression de sélection de type $n\tau$ appliquée a un tableau de bornes δ a pour résultat un tableau ayant des éléments de même type et δ' pour bornes.

Le nombre de dimensions du tableau résultat est égal au nombre d'éléments de *exp_list* qui sont des tableaux d'entiers, la taille de la p^{ieme} dimension égale à celle du p^{ieme} tableau d'entiers de *exp_list*.

$$\frac{\begin{array}{c} n\tau(0) = int, (n_k/0) \\ \delta'(0) = n_k \\ \overset{\text{sel type}}{\vdash} \; n\tau \circ (\lambda n.n+1),\ \delta \circ (\lambda n.n+1),\ \delta' \circ (\lambda n.n+1) \end{array}}{\overset{\text{sel type}}{\vdash} \; n\tau,\ \delta\ :\ \delta'}$$

Si le p^{ieme} élément de la suite *exp_list* est un entier, la valeur du p^{ieme} indice est la même pour tous les éléments sélectionnés. Cet indice, n'étant pas discriminant dans le tableau résultat, il est supprimé.

$$\frac{n\tau(0) = int \qquad \overset{\text{sel type}}{\vdash} \quad n\tau \circ (\lambda n.n+1),\ \delta \circ (\lambda n.n+1),\ \delta'}{\overset{\text{sel type}}{\vdash} \quad n\tau,\ \delta,\ \delta'}$$

$$\overset{\text{sel type}}{\vdash} \quad \lambda n.\ \bot,\ \lambda n.\ \bot,\ \lambda n.\ \bot$$

Un simple élément de *exp* est sélectionné si tous les éléments de *exp_list* sont des entiers.

$$\frac{\begin{array}{c}\overset{\text{type}}{\alpha,\ \varphi,\ \Theta,\ \mu \vdash}\ exp\ :\ (\tau,\ \delta),\ \gamma \\ \forall p(\delta(p) = \bot \Leftrightarrow p \geq k) \\ \overset{\text{type}}{\alpha,\ \varphi,\ \Theta,\ \mu \vdash}\ exp\_list\ :\ n\tau,\ \lambda n.\ (si\ n < k\ alors\ static\ sinon\ \bot) \\ n\tau = \lambda n.\ (si\ n < k\ alors\ int\ sinon\ \bot) \\ \overset{\text{bound}}{\vdash}\ n\tau,\ \delta\end{array}}{\overset{\text{type}}{\alpha,\ \varphi,\ \Theta,\ \mu \vdash}\ exp[exp\_list]\ :\ \tau,\ \gamma}$$

Dans le cas contraire le résultat est un sous-tableau

$$\frac{\begin{array}{c}\overset{\text{type}}{\alpha,\ \varphi,\ \Theta,\ \mu \vdash}\ exp\ :\ (\tau,\ \delta),\ \gamma \\ \forall p(\delta(p) = \bot \Leftrightarrow p \geq k) \\ \overset{\text{type}}{\alpha,\ \varphi,\ \Theta,\ \mu \vdash}\ exp\_list\ :\ n\tau,\ \lambda n.\ (si\ n < k\ alors\ static\ sinon\ \bot) \\ \forall p < k\ (n\tau(p) = int) \vee (n\tau(p) = (int,\ n_p\ /\ 0)) \\ \overset{\text{sel type}}{\vdash}\ n\tau,\ \delta,\ \delta' \\ \overset{\text{bound}}{\vdash}\ n\tau,\ \delta\end{array}}{\overset{\text{type}}{\alpha,\ \varphi,\ \Theta,\ \mu \vdash}\ exp[exp\_list]\ :\ (\tau,\ \delta'),\ \gamma}$$

6.2.4 Listes d'expressions

$$\frac{\begin{array}{c}\tau \notin \mathcal{TYPELIST} \\ \overset{\text{type}}{\alpha,\ \varphi,\ \Theta,\ \mu \vdash}\ exp\ :\ \tau,\ \gamma \\ \overset{\text{type}}{\alpha,\ \varphi,\ \Theta,\ \mu \vdash}\ exp\_list\ :\ n\tau,\ n\gamma\end{array}}{\begin{array}{l}\overset{\text{type}}{\alpha,\ \varphi,\ \Theta,\ \mu \vdash}\ exp\ exp\_list:\ \lambda n.\ (si\ n = 0\ alors\ \tau\ sinon\ n\tau(n-1)), \\ \lambda n.\ (si\ n = 0\ alors\ \gamma\ sinon\ n\gamma(n-1))\end{array}}$$

Les listes d'expressions ne sont pas hiérarchiques en LUSTRE, le type d'une liste *exp_list* dont certains de ses éléments sont eux-mêmes des listes est le même que celui la liste obtenu une fois *exp_list* mise à plat.

$$\frac{\alpha,\ \varphi,\ \Theta,\ \mu \overset{\text{type}}{\vdash} exp\ :\ \lambda n.\ si\ n\ <\ k\ alors\ \tau_i\ sinon\ \bot,\ n\gamma \qquad \alpha,\ \varphi,\ \Theta,\ \mu \overset{\text{type}}{\vdash} exp\_list\ :\ n\tau,\ n\gamma}{\alpha,\ \varphi,\ \Theta,\ \mu \overset{\text{type}}{\vdash} exp\ exp\_list :\ \lambda n.\ (si\ n\ <\ k\ alors\ n\tau'(n)\ sinon\ n\tau(n-k)),\ \lambda n.\ (si\ n\ <\ k\ alors\ n\gamma'(n)\ sinon\ n\gamma(n-k))}$$

6.2.5 Déclarations d'un noeud

Dans un noeud, Il y a trois types de déclarations :

- Les paramètres d'entrée.
- Les paramètres de sortie.
- Les variables, constantes et statiques locaux au noeud.

Le type des déclarations de constantes et de statiques, dans un noeud, est donné explicitement et leur définition repoussée dans le système d'équations. Toutes ces déclarations ont ainsi un format similaire.

6.2.5.1 Variables

Toute variable d'un programme LUSTRE est définie à l'intérieur d'un noeud, il n'existe aucune variable globale.

$$\frac{\alpha,\ \varphi,\ \Theta,\ \mu \overset{\text{type}}{\vdash} type\ :\ \tau \qquad \alpha(id)\ =\ (\tau,\ var)}{\alpha,\ \varphi,\ \Theta,\ \mu \overset{\text{type}}{\vdash} id\ type\ :\ \alpha(id)}$$

6.2.5.2 Constantes

$$\frac{\alpha,\ \varphi,\ \Theta,\ \mu \overset{\text{type}}{\vdash} type\ :\ \tau \qquad \alpha(id)\ =\ (\tau,\ const)}{\alpha,\ \varphi,\ \Theta,\ \mu \overset{\text{type}}{\vdash} \texttt{const}\ id\ type\ :\ \alpha(id)}$$

6.2.5.3 Statiques

$$\frac{\alpha,\ \varphi,\ \Theta,\ \mu \overset{\text{type}}{\vdash} type\ :\ \tau \qquad \alpha(id)\ =\ (\tau,\ static)}{\alpha,\ \varphi,\ \Theta,\ \mu \overset{\text{type}}{\vdash} \texttt{static}\ id\ type\ :\ \alpha(id)}$$

6.2.5.4 Liste de variables

$$\frac{\alpha,\ \varphi,\ \Theta,\ \mu \overset{\text{type}}{\vdash} var\ :\ \tau,\ \gamma \qquad \alpha,\ \varphi,\ \Theta,\ \mu \overset{\text{type}}{\vdash} var\_list\ :\ n\tau,\ n\gamma}{\alpha,\ \varphi,\ \Theta \overset{\text{type}}{\vdash} var\ var\_list :\ \lambda n.\ (si\ n\ =\ 0\ alors\ \tau\ sinon\ n\tau(n-1)),\ \lambda n.\ (si\ n\ =\ 0\ alors\ \gamma\ sinon\ n\gamma(n-1))}$$

6.3 Evaluation des expressions statiques

6.3.1 Equations

La valeur de chaque objet statique en partie gauche doit être identique à celle de l'expression correspondante en partie droite.

$$\frac{\alpha,\ \varphi,\ \Theta,\ \mu \overset{\text{val}}{\vdash} left\_list\ :\ n\nu \qquad \alpha,\ \varphi,\ \Theta,\ \mu \overset{\text{val}}{\vdash} exp\_list\ :\ n\nu' \qquad \forall n\ ((n\gamma(n) =\ static)\ \Rightarrow\ (n\nu(n) = n\nu'(n)))}{\alpha,\ \varphi,\ \Theta,\ \mu \overset{\text{val}}{\vdash} left\_list = exp\_list}$$

6.3.2 Opérateurs

L'évaluation statique d'un opérateur prédéfini n'est possible que si le genre de sa sortie, fonction de celui de ses entrées, est `static`.

On suppose disposer d'une fonction *eval* qui, à partir d'un nom d'opérateur prédéfini et d'un vecteur de valeurs d'entrée, retourne le résultat correspondant.

$$\frac{\begin{array}{c}\alpha,\ \varphi,\ \Theta,\ \mu \overset{\text{type}}{\vdash} exp\ :\ \tau,\ \gamma \\ \alpha,\ \varphi,\ \Theta,\ \mu \overset{\text{val}}{\vdash} exp\ :\ n\nu \\ \varphi(id,\ \lambda n.\bot)\ =\ predef,\ (n\tau,\ n\gamma)\ \rightarrow\ (n\tau',\ static), \lambda n.\bot\end{array}}{\alpha,\ \gamma,\ \Theta,\ \mu \overset{\text{val}}{\vdash} id\ (exp\_list)\ :\ eval(id,\ n\nu)}$$

L'évaluation statique d'un opérateur `with` consiste à calculer la valeur de sa deuxième ou troisième opérande suivant celle de sa condition.

$$\frac{\begin{array}{c}\alpha,\ \varphi,\ \Theta,\ \mu \overset{\text{val}}{\vdash} exp\ :\ true \\ \alpha,\ \varphi,\ \Theta,\ \mu \overset{\text{val}}{\vdash} exp'\ :\ \nu\end{array}}{\alpha,\ \gamma,\ \Theta,\ \mu \overset{\text{type}}{\vdash} \texttt{with}\ exp\ \texttt{then}\ exp'\ \texttt{else}\ exp''\ :\ \nu}$$

$$\frac{\begin{array}{c}\alpha,\ \varphi,\ \Theta,\ \mu \overset{\text{val}}{\vdash} exp\ :\ false \\ \alpha,\ \varphi,\ \Theta,\ \mu \overset{\text{val}}{\vdash} exp''\ :\ \nu\end{array}}{\alpha,\ \gamma,\ \Theta,\ \mu \overset{\text{type}}{\vdash} \texttt{with}\ exp\ \texttt{then}\ exp'\ \texttt{else}\ exp''\ :\ \nu}$$

Le résultat de l'opérateur `..` est le tableau de tous les entiers compris entre n et n' espacés de n''.

$$\frac{\begin{array}{c}\alpha,\ \varphi,\ \Theta,\ \mu \overset{\text{val}}{\vdash} exp\ :\ k \\ \alpha,\ \varphi,\ \Theta,\ \mu \overset{\text{val}}{\vdash} exp'\ :\ k' \\ \alpha,\ \varphi,\ \Theta,\ \mu \overset{\text{val}}{\vdash} exp''\ :\ k''\end{array}}{\alpha,\ \gamma,\ \Theta,\ \mu \overset{\text{type}}{\vdash} exp\ ..\ exp'\ \texttt{step}\ exp''\ :\ \lambda n.si\ (n < (\mid k - k' \mid /k'') + 1)\ alors\ k + n * k''\ sinon\ \bot}$$

Les valeurs des sorties statiques d'un noeud sont fournies par l'environnement

$$\frac{\begin{array}{c}\alpha,\ \varphi,\ \Theta,\ \mu \overset{\text{val}}{\vdash} exp\_list\ :\ n\nu \\ \varphi(id, n\nu)\ =\ ((n\tau,\ n\gamma)\ \rightarrow\ (n\tau',\ static), n\nu'\end{array}}{\alpha,\ \gamma,\ \Theta,\ \mu \overset{\text{type}}{\vdash} id\ (exp\_list)\ :\ n\nu'}$$

6.3.3 Concaténation de tableaux

Le résultat d'un opérateur de concaténation est obtenu en mettant simplement bout à bout les valeurs de ses deux opérandes.

$$\frac{\begin{array}{c}\alpha,\ \varphi,\ \Theta,\ \mu \overset{val}{\vdash} exp\ :\ n\nu \\ n\nu\ =\ \lambda p.\ (si\ p\ <\ k\ alors\ \nu_p\ sinon\ \bot) \\ \alpha,\ \varphi,\ \Theta,\ \mu \overset{val}{\vdash} exp'\ :\ n\nu'\end{array}}{\alpha,\ \varphi,\ \Theta,\ \mu \overset{val}{\vdash} exp\ \mid\ exp'\ :\ \lambda p.\ (si\ p < k\ alors\ n\nu(p)\ sinon\ n\nu'(p-k))}$$

6.3.3.1 Extension

Le prédicat *ext* permet de construire la valeur résultat d'un opérateur d'extension à partir de celle qui lui est fournie en entrée.

$$\overset{ext}{\vdash}\ \nu,\ \delta,\ n\nu'$$

Il est vérifié si la valeur ν est transformée par l'opérateur en $n\nu'$, un tableau d'éléments de même valeur, dont la liste des dimensions est donné par δ.

$n\nu''$ est la valeur d'entrée étendue en un tableau à n-1 dimensions. L'extension à n dimensions est obtenue en construisant un tableau dont chaque élément est une copie de $n\nu''$.

$$\frac{\begin{array}{c}n\nu' = \lambda p.\ si\ p < n\nu(0)\ alors\ n\nu''\ sinon\ \bot \\ \overset{ext}{\vdash}\ \nu,\ \delta \circ (\lambda n.\ n+1),\ n\nu''\end{array}}{\overset{ext}{\vdash}\ \nu,\ \delta,\ n\nu'}$$

Si aucune extension n'est faite, c'est à dire si la valeur en entrée de l'opérateur est étendue en un tableau sans dimension, la valeur inchangée est produite en résultat.

$$\overset{ext}{\vdash}\ \nu,\ \lambda n.\bot,\ \nu$$

La règle d'évaluation d'un opérateur d'extension est alors :

$$\frac{\begin{array}{c}\alpha,\ \varphi,\ \Theta,\ \mu \overset{val}{\vdash} exp\ :\ \nu \\ \alpha,\ \varphi,\ \Theta,\ \mu \overset{val}{\vdash} exp\_list\ :\ n\nu \\ \overset{ext}{\vdash}\ \nu,\ n\nu,\ n\nu'\end{array}}{\alpha,\ \varphi,\ \Theta,\ \mu \overset{type}{\vdash} exp\hat{}exp\_list\ :\ n\nu'}$$

6.3.3.2 Construction

La valeur d'une liste de n éléments et celle d'un tableau de même taille sont représentés de la même manière.

$$\frac{\alpha,\ \varphi,\ \Theta,\ \mu \overset{\text{val}}{\vdash} exp\_list\ :\ n\nu}{\alpha,\ \varphi,\ \Theta,\ \mu \overset{\text{type}}{\vdash} [\ exp\_list\ ] : n\nu}$$

6.3.3.3 Sélection

Le prédicat *sel val* permet de construire la valeur résultat d'un opérateur de sélection à partir du tableau de valeurs qui lui est fourni en entrée.

$$\overset{\text{sel val}}{\vdash}\ n\nu,\ n\nu',\ \nu$$

Il est vérifié si le tableau de valeurs $n\nu$ est transformé en la valeur ν, $n\nu'$ étant la valeur de l'expression de sélection.

Si le premier élément de l'expression de sélection est un entier n, alors le résultat de l'opérateur est obtenu en appliquant le reste de l'expression de sélection au sous-tableau composé des éléments dont le premier indice est n.

$$\frac{n\nu'(0) = n \qquad \overset{\text{sel val}}{\vdash}\ n\nu(n),\ n\nu' \circ (\lambda n.n+1),\ \nu}{\overset{\text{sel val}}{\vdash}\ n\nu,\ n\nu',\ \nu}$$

Si le premier élément de l'expression de sélection est un tableau de k entiers n_1, ... , n_k, le p^{ieme} élément du résultat est obtenu en appliquant le reste de l'expression de sélection au sous-tableau composé des éléments dont le premier indice est n_p

$$\frac{\begin{array}{c} n\nu'(0) = \lambda p.\ si\ p < k\ alors\ n_p\ sinon\ \bot \\ \forall p < k\ (\overset{\text{sel val}}{\vdash}\ n\nu(n_p),\ n\nu' \circ (\lambda n.n+1),\ \nu(p)) \\ \forall p < k\ (\nu(p)\ =\ \bot) \end{array}}{\overset{\text{sel val}}{\vdash}\ n\nu,\ n\nu',\ \nu}$$

S'il n'y a plus rien à sélectionner alors tout ce qui reste est pris.

$$\overset{\text{sel val}}{\vdash}\ \nu,\ \lambda n.\ \bot,\ \nu$$

L'évaluation d'un opérateur de sélection est alors :

$$\frac{\begin{array}{c}\alpha,\ \varphi,\ \Theta,\ \mu \overset{val}{\vdash} exp\ :\ n\nu \\ \alpha,\ \varphi,\ \Theta,\ \mu \overset{val}{\vdash} exp\_list\ :\ n\nu' \\ \overset{sel\ val}{\vdash}\ \nu,\ n\nu,\ n\nu'\end{array}}{\alpha,\ \varphi,\ \Theta,\ \mu \overset{type}{\vdash} exp[exp\_list]\ :\ \nu}$$

6.3.4 Listes d'expressions

Une liste d'expressions peut avoir certains éléments statiques et d'autres non. $\perp$ est la valeur associée aux éléments non statiques.

$$\frac{\begin{array}{c}\alpha,\ \varphi,\ \Theta,\ \mu \overset{val}{\vdash} exp\ :\ \nu \\ \alpha,\ \varphi,\ \Theta,\ \mu \overset{val}{\vdash} exp\_list\ :\ n\nu\end{array}}{\alpha,\ \varphi,\ \Theta,\ \mu \overset{val}{\vdash} exp\ exp\_list :\ \lambda n.\ (si\ n\ =\ 0\ alors\ \nu\ sinon\ n\nu(n-1))}$$

$$\frac{\begin{array}{c}\alpha,\ \varphi,\ \Theta,\ \mu \overset{type}{\vdash} exp\ :\ \tau,\ \gamma \\ \gamma > static \\ \alpha,\ \varphi,\ \Theta,\ \mu \overset{val}{\vdash} exp\_list\ :\ n\nu\end{array}}{\alpha,\ \varphi,\ \Theta,\ \mu \overset{val}{\vdash} exp\ exp\_list :\ \lambda n.\ (si\ n\ =\ 0\ alors\ \perp\ sinon\ n\nu(n-1))}$$

La valeur d'une liste hiérarchique est mise à plat.

$$\frac{\begin{array}{c}\alpha,\ \varphi,\ \Theta,\ \mu \overset{val}{\vdash} exp\ :n\nu \\ n\nu\ =\ \lambda p.\ (si\ p\ <\ k\ alors\ \nu_p\ sinon\ \perp) \\ \alpha,\ \varphi,\ \Theta,\ \mu \overset{type}{\vdash} exp\_list\ :\ n\nu'\end{array}}{\alpha,\ \varphi,\ \Theta,\ \mu \overset{type}{\vdash} exp\ exp\_list :\ \lambda n.\ (si\ n\ <\ k\ alors\ n\nu'(n)\ sinon\ n\nu'(n-k))}$$

6.3.5 Déclarations d'un noeud

L'évaluation d'une variable ou d'une constante est impossible, le résultat est $\perp$.

6.3.5.1 Variables

$$\alpha,\ \varphi,\ \Theta,\ \mu \overset{val}{\vdash} id\ type\ :\ \perp$$

6.3.5.2 Constantes

$$\alpha,\ \varphi,\ \Theta,\ \mu \overset{\text{val}}{\vdash} \texttt{const}\ id\ type\ :\ \bot$$

6.3.5.3 Statiques

$$\frac{\mu(id)\ \neq\ \bot}{\alpha,\ \varphi,\ \Theta,\ \mu \overset{\text{val}}{\vdash} \texttt{static}\ id\ type\ :\ \mu(id)}$$

6.3.5.4 Liste de variables

L'évaluation d'une liste de déclarations retourne la liste des valeurs de chaque déclaration.

$$\frac{\begin{array}{c}\alpha,\ \varphi,\ \Theta,\ \mu \overset{\text{val}}{\vdash} var\ :\ \nu \\ \alpha,\ \varphi,\ \Theta,\ \mu \overset{\text{val}}{\vdash} var\_list\ :\ n\nu\end{array}}{\alpha,\ \varphi,\ \Theta,\ \mu \overset{\text{val}}{\vdash} var\ var\_list :\ \lambda n.\ (si\ n\ =\ 0\ alors\ \nu\ sinon\ n\nu(n-1))}$$

Chapitre 7

Introduction des structures nommées

La sémantique naturelle de LUSTRE V4 va maintenant être complétée en y incluant celle des deux parties du langage non encore décrites. La première concerne une autre catégorie de types composés , les structures à champs nommés

7.1 Syntaxe abstraite

La syntaxe abstraite est complétée au niveau des déclarations de types et des expressions.

Nous noterons par des points de suspension les parties de règles déjà données.

7.1.1 Types

Un constructeur est ajouté. Le type d'une structure définit le type de chacun de ses champs et leur associe un identificateur de façon à pouvoir y accéder.

field ::= id:type

type ::= ... ‖ [field_list]

7.1.2 Expressions

Un opérateur de construction de structure et un autre de sélection sont ajoutés aux expressions.

named_exp ::= id:exp

exp ::= ... ‖ exp.id ‖ [named_exp]

7.1.3 Expressions en partie gauche d'équations

Les opérateurs rajoutés en partie droite d'équation le sont également en partie gauche.

named_left ::= id:left

left ::= ... ‖ exp.id ‖ [named_left]

7.2 Domaines

7.2.1 Types

$\mathcal{STRUCTYPE} = \mathcal{TYPE} \times \mathcal{DIM} \cup \mathcal{TYPEILIST}$

La définition des types structurés est complétée pour prendre en compte les structures.

$\mathcal{TYPEILIST} = \mathcal{ID} \rightarrow \mathcal{TYPE} \ni i\tau,\ i\theta$

Le type d'une structure est décrit par une fonction $i\tau$ des identificateurs vers les types. $i\tau$(i) est le type du champ associé à l'identificateur i, $\perp$ si la structure ne comporte pas de champ ayant un tel nom.

7.2.2 Valeurs

$\mathcal{VALUE} = \{\perp\}\ \mathcal{ID} \cup \mathcal{VAL} \cup \mathcal{VALUENLIST} \cup \mathcal{VALUEILIST} \ni \nu$

Le domaine des valeurs est également complété :

$\mathcal{VALUEILIST} = \mathcal{N} \rightarrow \mathcal{VALUE} \ni i\nu$

La valeur d'une structure est une fonction associant à chaque identificateur désignant un champ de la structure sa valeur et $\perp$ aux autres.

Rien n'est changé dans les autres domaines. Notamment, tous les éléments d'une structure, comme c'est le cas avec les tableaux, ont même genre

7.3 Calcul des types

7.3.1 Types

Le type d'une structure est défini à partir d'une liste de déclarations de champs définissant leurs types et leurs identificateurs associés. Il est important de noter que l'ordre des déclarations des champs est sans importance. Deux champs d'une même structure ne doivent pas avoir le même nom.

A la différence des listes, les structures sont hiérarchiques, aucune mise à plat n'est faite si un champ d'une structure en est lui même une.

$$\frac{\begin{array}{c}\alpha,\ \varphi,\ \Theta,\ \mu \overset{\text{type}}{\vdash} field\_list : i\tau \\ i\tau(id) = \bot \\ \alpha,\ \varphi,\ \Theta,\ \mu \overset{\text{type}}{\vdash} type : \tau\end{array}}{\alpha,\ \varphi,\ \Theta,\ \mu \overset{\text{type}}{\vdash} id : type\ field\_list\ :\ \lambda i.(si\ i = id\ alors\ \tau\ sinon\ i\tau(i))}$$

$$\frac{\alpha,\ \varphi,\ \Theta,\ \mu \overset{\text{type}}{\vdash} named\_type\_list : i\tau}{\alpha,\ \varphi,\ \Theta,\ \mu \overset{\text{type}}{\vdash} [named\_type\_list]\ :\ i\tau}$$

7.3.2 Equations

La notion de type et de valeur dérivée s'étend aux structures.

$$\frac{\forall i\ (i\tau(i) \neq \bot) \Rightarrow \overset{\text{derived\_type}}{\vdash}\ \tau,\ i\tau(i)}{\overset{\text{derived\_type}}{\vdash}\ \tau,\ i\tau}$$

$$\frac{\forall i\ (i\nu(i) \neq \bot) \Rightarrow \overset{\text{derived\_val}}{\vdash}\ (\nu,\ i\nu(i))}{\overset{\text{derived\_val}}{\vdash}\ \nu,\ i\nu'}$$

7.3.3 Opérateurs

Le prédicat *unify* définissant la substitution permettant l'instanciation d'un opérateur polymorphe est également étendu.

$$\frac{\forall i\ \Phi \overset{\text{unify}}{\vdash} i\theta(i),\ i\tau(i)}{\Phi \overset{\text{unify}}{\vdash} i\theta,\ i\tau}$$

7.3.4 Concaténation

L'opérateur de concaténation est applicable à deux structures dont les ensembles de labels sont disjoints. Le résultat est une troisième structure composée des éléments des deux autres.

$$\frac{\begin{array}{c}\alpha,\ \varphi,\ \Theta,\ \mu \overset{\text{type}}{\vdash} exp\ :\ i\tau,\ \gamma \\ \alpha,\ \varphi,\ \Theta,\ \mu \overset{\text{type}}{\vdash} exp'\ :\ i\tau',\ \gamma' \\ \forall i\ (i\tau(i) = \bot) \vee (i\tau'(i) = \bot)\end{array}}{\alpha,\ \varphi,\ \Theta,\ \mu \overset{\text{type}}{\vdash} exp \mid exp'\ :\ \lambda i.\ (i\tau[i\tau'],\ \sqcup(\gamma,\ \gamma'))}$$

7.3.5 Construction

Un opérateur de définition par extension de structure, voisin de celui pour les tableaux, est disponible. Les champs de la structure produite sont ceux définis par *named_exp_list* et leur type, celui de l'expression qui leur est associée.

$$\frac{\alpha,\ \varphi,\ \Theta,\ \mu \overset{\text{type}}{\vdash} named\_exp\_list : i\tau,\ \gamma}{\alpha,\ \varphi,\ \Theta,\ \mu \overset{\text{type}}{\vdash} [named\_exp\_list]\ :\ i\tau,\ \gamma}$$

$$\frac{\begin{array}{c}\alpha,\ \varphi,\ \Theta,\ \mu \overset{\text{type}}{\vdash} named\_exp\_list : i\tau,\ \gamma \\ i\tau(id)\ =\ \bot \\ \alpha,\ \varphi,\ \Theta,\ \mu \overset{\text{type}}{\vdash} exp : \tau,\ \gamma'\end{array}}{\alpha,\ \varphi,\ \Theta,\ \mu \overset{\text{type}}{\vdash} id : exp\ named\_exp\_list\ :\ \lambda i.(si\ i = id\ alors\ \tau\ sinon\ i\tau(i)),\ \sqcup(\gamma,\ \gamma')}$$

7.3.6 Sélection

L'opération de sélection dans une structure est plus limitée que dans dans un tableau, puisqu'elle permet seulement la sélection d'un élément en spécifiant le nom auquel il est associé. *id* doit bien sûr désigner un champ de la structure.

$$\frac{\begin{array}{c}i\tau(id)\ \neq\ \bot \\ \alpha,\ \varphi,\ \Theta,\ \mu \overset{\text{type}}{\vdash} exp : i\tau,\ \gamma\end{array}}{\alpha,\ \varphi,\ \Theta,\ \mu \overset{\text{type}}{\vdash} exp.id\ :\ i\tau(id),\ \gamma}$$

7.4 Evaluation des expressions statiques

7.4.1 Concaténation

$$\frac{\begin{array}{c}\alpha,\ \varphi,\ \Theta,\ \mu \overset{\text{val}}{\vdash} exp\ :\ i\nu \\ \alpha,\ \varphi,\ \Theta,\ \mu \overset{\text{val}}{\vdash} exp'\ :\ i\nu'\end{array}}{\alpha,\ \varphi,\ \Theta,\ \mu \overset{\text{val}}{\vdash} exp\ |\ exp'\ :\ \lambda i.\ i\nu[i\nu']}$$

7.4.2 Construction

$$\frac{\alpha,\ \varphi,\ \Theta,\ \mu \overset{\text{val}}{\vdash} named\_exp\_list : i\nu,\ \gamma}{\alpha,\ \varphi,\ \Theta,\ \mu \overset{\text{val}}{\vdash} [named\_exp\_list]\ :\ i\nu}$$

$$\frac{\alpha,\ \varphi,\ \Theta,\ \mu \overset{\text{val}}{\vdash} named\_exp\_list : i\nu \qquad \alpha,\ \varphi,\ \Theta,\ \mu \overset{\text{val}}{\vdash} exp : \nu}{\alpha,\ \varphi,\ \Theta,\ \mu \overset{\text{val}}{\vdash} id : exp\ named\_exp\_list\ :\ \lambda i.(si\ i = id\ alors\ \nu\ sinon\ i\nu(i))}$$

7.4.3 Sélection

$$\frac{\alpha,\ \varphi,\ \Theta,\ \mu \overset{\text{val}}{\vdash} exp : i\nu}{\alpha,\ \varphi,\ \Theta,\ \mu \overset{\text{val}}{\vdash} exp.id\ :\ i\nu(id)}$$

Chapitre 8

Extension homomorphe des opérateurs

Le dernier élément du langage qui n'avait jusqu'à présent pas été pris en compte est l'extension homomorphe des opérateurs. Intuitivement, tout opérateur prédéfini, local ou externe peut être appliqué à des tableaux ou des structures de paramètres et produire en sorties des tableaux ou des structures de résultats.

Nous allons voir maintenant les modifications à apporter aux règles de typage ou de calcul des valeurs pour prendre en compte cette notion.

8.1 Calcul des types

Pour définir comment est typé un opérateur homomorphe, nous définissons un nouveau prédicat *apply_type*.

$$\alpha,\ \varphi,\ \Theta,\ \mu \overset{\text{apply\_type}}{\vdash} id,\ n\nu,\ n\tau\ ,\ n\gamma\ :\ n\tau',\ n\gamma'$$

Celui-ci est vérifié si l'instanciation de l'opérateur id, définie en fonction de la liste des valeurs de ses entrées statiques $n\nu$, a respectivement pour type et genre d'entrée $n\tau$ et γ et pour type et genre de sortie $n\tau'$ et $n\gamma'$.

Une instanciation d'opérateur est correcte si elle est définie, modulo une substitution sur les types des paramètres pour résoudre le polymorphisme, dans l'environnement φ. Il s'agit là de l'utilisation "normale" de l'opérateur qui a été employée jusqu'à présent.

$$\frac{\begin{array}{c}\varphi(id,\ n\nu) = (n\theta, n\gamma)\ \rightarrow\ (n\theta', n\gamma'),\ n\nu' \\ \Phi\ \vdash\ n\theta,\ n\tau \\ \Phi'\ \vdash\ n\theta',\ n\tau' \\ \forall k\ (\Phi'(k)\ \neq\ \perp\ \Rightarrow\ (\Phi'(k) = \Phi(k)))\end{array}}{\alpha,\ \varphi,\ \Theta,\ \mu \overset{\text{apply\_type}}{\vdash} id,\ n\nu,\ n\tau\ ,\ n\gamma\ :\ n\tau',\ n\gamma'}$$

Les opérateurs rendus homomorphes, une instanciation d'opérateur peut être, dans deux cas, correcte bien que n'étant pas définie dans l'environnement φ .

Dans le premier cas, tous les paramètres de l'opérateur sont des structures ayant les même noms de champs. De plus, si l'on considère n'importe lequel de ces champs de nom i, l'instanciation de l'opérateur en remplaçant chaque paramètre par son champ i est correcte. C'est par exemple le cas de l'expression `[l1:x1, l2:x2] or [l1:y1, l2:y2]` où `x1`, `x2`, `y1`, `y2` sont quatre variables booléennes.

$$\frac{\begin{array}{c}(id,\ n\nu) \notin \varphi \\ \forall p\ ((n\tau(p) \neq \bot) \Leftrightarrow (p < k)) \\ \forall p\ ((n\tau'(p) \neq \bot) \Leftrightarrow (p < k')) \\ \forall p < k\ n\tau(p) \in TYPEILIST \\ \forall i,\ p < k,\ q < k,\ p' < k'\ ((n\tau(p)(i) = \bot) \Leftrightarrow (n\tau(q)(i) = \bot) \Leftrightarrow (n\tau'(p')(i) = \bot) \\ \forall i\ (\alpha,\ \varphi,\ \Theta,\ \mu \overset{\text{apply\_type}}{\vdash} id,\ \lambda p.\ (si\ p < k\ alors\ n\nu(p)(i)\ sinon\ \bot), \\ \lambda p.\ (si\ p < k\ alors\ n\tau(p)(i)\ sinon\ \bot),\ n\gamma,\ \lambda p.\ (si\ p < k'\ alors\ n\tau'(p)(i)\ sinon\ \bot),\ n\gamma')\end{array}}{\alpha,\ \varphi,\ \Theta,\ \mu \overset{\text{apply\_type}}{\vdash} id,\ n\nu,\ n\tau,\ n\gamma\ :\ n\tau',\ n\gamma'}$$

$$\alpha,\ \varphi,\ \Theta,\ \mu \overset{\text{apply\_type}}{\vdash} id, \lambda n.\bot,\ \lambda n.\bot,\ n\gamma,\ \lambda n.\bot,\ n\gamma'$$

Dans le second cas, tous les paramètres sont des tableaux k-dimensionnels de même taille et l'instanciation de l'opérateur en remplaçant chaque paramètre par son élément d'indices $n_1, .., n_k$ est correcte.

$$\frac{\begin{array}{c}(id,\ n\nu) \notin \varphi \\ \forall p\ ((n\tau(p) \neq \bot) \Leftrightarrow (p < k)) \\ \forall p\ ((n\tau'(p) \neq \bot) \Leftrightarrow (p < k')) \\ \forall p < k\ (n\tau(p) = (\tau_p, \delta)) \\ \forall p < k'\ (n\tau'(p) = (\tau'_p, \delta)) \\ \alpha,\ \varphi,\ \Theta,\ \mu \overset{\text{apply\_type}}{\vdash} id,\ \lambda p.\ (si\ p < k\ alors\ n\nu(p)\ sinon\ \bot), \\ \lambda p.\ (si\ p < k\ alors\ \tau_p\ sinon\ \bot),\ n\gamma,\ \lambda p.\ (si\ p < k'\ alors\ \tau'_p\ sinon\ \bot),\ n\gamma\end{array}}{\alpha,\ \varphi,\ \Theta,\ \mu \overset{\text{apply\_type}}{\vdash} id,\ n\nu,\ n\tau\ ,\ n\gamma\ :\ n\tau',\ n\gamma'}$$

Avec ce prédicat qui définit quelles sont les instanciations correctes d'un opérateur, la règle de calcul des types d'un opérateur devient simple à écrire.

$$\frac{\begin{array}{c}\alpha,\ \varphi,\ \Theta,\ \mu \overset{\text{type}}{\vdash} exp\_list\ :\ n\tau,\ n\gamma \\ \alpha,\ \varphi,\ \Theta,\ \mu \overset{\text{val}}{\vdash} exp\_list\ :\ n\nu \\ \alpha,\ \varphi,\ \Theta,\ \mu \overset{\text{apply\_type}}{\vdash} n\nu,\ n\tau,\ n\gamma\ :\ n\tau',\ n\gamma'\end{array}}{\alpha,\ \gamma,\ \Theta,\ \mu \overset{\text{type}}{\vdash} id\ (exp\_list)\ :\ n\tau',\ n\gamma'}$$

8.2 Evaluation des expressions statiques

De façon similaire à ce qui vient d'être fait pour le calcul des types, il s'agit maintenant de la régle d'évaluation des statiques.

Pour cela nous définissons un nouveau prédicat *apply_val*

$$\alpha,\ \varphi,\ \Theta,\ \mu \overset{\text{apply\_val}}{\vdash} id,\ n\nu,\ n\tau\ :\ n\nu'$$

Celui-ci est vérifié si l'instanciation de l'opérateur id, définie en fonction de la liste des valeurs de ses entrées statiques $n\nu$, a respectivement pour type d'entrée et produit $n\nu'$ comme valeur résultat.

Les deux règles suivantes définissent le prédicat pour les instanciations usuelles de l'opérateur.

$$\frac{\varphi(id,\ \lambda n.\bot)\ =\ predef,\ (n\tau,\ n\gamma)\ \rightarrow\ (n\tau',\ static)}{\alpha,\ \varphi,\ \Theta,\ \mu \overset{\text{apply\_val}}{\vdash} id,\ n\nu,\ n\tau\ :\ eval(id,\ n\nu')}$$

$$\frac{\varphi(id, n\nu)\ =\ ((n\tau,\ n\gamma)\ \rightarrow\ (n\tau',\ static), n\nu'}{\alpha,\ \varphi,\ \Theta,\ \mu \overset{\text{apply\_val}}{\vdash} id,\ n\nu,\ n\tau\ :\ n\nu'}$$

Si l'extension homomorphe de l'opérateur est utilisée, il s'agit d'appliquer récursivement ce prédicat sur tous les éléments des structures ou des tableaux de paramètres.

Si tous les paramètres sont des structures ayant les mêmes champs, l'opérateur est appliqué respectivement sur chacun de ces derniers.

$$\frac{\begin{array}{c}(id,\ n\nu) \notin \varphi \\ \forall p < k\ n\tau(p) \in TYPEILIST \\ \forall p \geq k\ (n\tau(p)\ =\ \bot) \\ \forall i\alpha,\ \varphi,\ \Theta,\ \mu \overset{\text{apply\_val}}{\vdash} id,\ \lambda p.\ (si\ p < k\ alors\ n\nu(p)(i)\ sinon\ \bot), \\ \lambda p.\ (si\ p < k\ alors\ n\tau(p)(i)\ sinon\ \bot),\ \lambda p.\ (si\ p < k'\ alors\ n\nu'(p)(i)\ sinon\ \bot)\end{array}}{\alpha,\ \varphi,\ \Theta,\ \mu \overset{\text{apply\_val}}{\vdash} id,\ n\nu,\ n\tau\ :\ n\nu'}$$

$$\alpha,\ \varphi,\ \Theta,\ \mu \overset{\text{apply\_val}}{\vdash} id, \lambda n.\bot,\ \lambda n.\bot,\ \lambda n.\bot$$

Si tous les paramètres sont des tableaux ayant les même q dimensions, l'opérateur est appliqué respectivement à chacun de leurs éléments.

$$\frac{\begin{array}{c}(id,\ n\nu) \notin \varphi \\ \forall p\ n\tau(p) \neq \bot) \Leftrightarrow (p < k) \\ \forall p < k\ (n\tau(p) = (\tau_p, \delta)); \\ \forall p < k'\ (n\tau'(p) = (\tau'_p, \delta)); \\ \forall p\ ((\delta(p) \neq \bot) \Leftrightarrow (p < q)) \\ \forall p_0 < \delta(0), .., p_{q-1} < n_{q-1} \\ \alpha,\ \varphi,\ \Theta,\ \mu \overset{\text{apply\_val}}{\vdash} id,\ \lambda p.\ (si\ p < k\ alors\ n\nu(p)(p_1)...(p_q)\ sinon\ \bot), \\ \lambda p.\ (si\ p < k\ alors\ \tau_p(p_1)...(p_q)\ sinon\ \bot),\ \lambda p.\ (si\ p < k'\ alors\ n\nu'(p)(p_1)...(p_q)\ sinon\ \bot))\end{array}}{\alpha,\ \varphi,\ \Theta,\ \mu \overset{\text{apply\_val}}{\vdash} id,\ n\nu,\ n\tau\ :\ n\nu'}$$

Avec le prédicat *apply_val*, la règle d'évaluation d'un opérateur devient, elle aussi, simple à écrire.

$$\frac{\begin{array}{c}\alpha,\ \varphi,\ \Theta,\ \mu \overset{\text{type}}{\vdash} exp\ :\ \tau,\ \gamma \\ \alpha,\ \varphi,\ \Theta,\ \mu \overset{\text{val}}{\vdash} exp\ :\ n\nu \\ \alpha,\ \varphi,\ \Theta,\ \mu \overset{\text{apply\_val}}{\vdash} n\nu,\ n\tau\ :\ n\nu'\end{array}}{\alpha,\ \gamma,\ \Theta,\ \mu \overset{\text{type}}{\vdash} id\ (exp\_list)\ :\ n\nu'}$$

Partie III

Le système Pollux

Chapitre 9

Phase haute de la compilation

9.1 Introduction

L'étude initiale de techniques de compilation de programmes LUSTRE a été faite par [Pla88] [Ray91] dans le cadre du développement des compilateurs LUSTRE V2 et V3.

Grossièrement, la structure du nouveau compilateur V4 se décompose en deux parties :

- La première est réalisée par un pré-compilateur chargé d'analyser le programme source et, si celui-ci est jugé correct, de le traduire en un programme LUSTRE noyau, c'est à dire un programme formé uniquement à partir :
  - de variables de type élémentaire. Les structures ou les tableaux ont été éclatés.
  - d'opérateurs prédéfinis ou externes. Le programme source a été mis à plat, toutes les instances de noeuds locaux ont été remplacées par leur définition.
- La seconde, en fonction du résultat désiré, invoque le post-processeur adéquat, chargé de la partie spécifique de la compilation. Il s'agit, par exemple, d'un générateur de code séquentiel plus ou moins optimisé, d'un générateur de circuits ... .

Le pré-compilateur doit effectuer de nombreuses opérations et vérifications dont certaines ne sont, dans la plupart des autres langages, effectuées qu'à l'exécution. C'est par exemple l'évaluation de l'arbre des appels, pour permettre la mise à plat du programme. L'avantage est que le code produit pour un programme LUSTRE est plus simple (donc plus rapide) mais, en contrepartie, le travail du compilateur est plus délicat. Il doit, face à un programme erroné, être à même de détecter la source et la nature de l'erreur pour en aviser l'utilisateur. Heureusement, la définition rigoureuse de LUSTRE et sa simplicité permettent de réaliser cela sans avoir à recourir à des outils algorithmiques sophistiqués et lourds à manipuler.

Cette partie va présenter les différents éléments composant le compilateur baptisé POLLUX, le pré-compilateur en premier, les principaux processeurs ensuite.

Le langage qui a été décrit dans la partie précédente (§ 3) est plus complet que celui accepté par la version actuelle du compilateur. La réalisation puis l'utilisation intensive de ce dernier ont permis de vérifier le bien-fondé des choix faits pour étendre LUSTRE mais, également, de se

faire une idée précise de ce qu'il était important ou utile de rajouter encore au langage, ainsi que des moyens à mettre en oeuvre, dans la prochaine version du compilateur, pour les traiter. Les points suivants du langage défini au chapitre précédent, qui se veut stable maintenant, ne sont pas traités par le compilateur existant.

- Les dépendances, pour permettre la compilation séparée de modules,
- Les paramètres statiques en sortie de noeud.
- Le polymorphisme des noeuds.
- Les assertions statiques
- Les expressions de sélection de tableau générales. Dans la version actuelle nous nous sommes limités uniquement à des expressions du type i .. j **step** k.
- L'accès aux structures par nom. Il est actuellement fait, comme les tableaux, par position. Il s'est révélé à l'usage que de telles structures n'étaient jamais utilisées.

9.2 Pré-compilation

Le pré-compilateur se décompose en une succession d'opérations,

- Les premières effectuent les vérifications propres aux extensions du langage et transformant un programme LUSTRE V4 en un programme LUSTRE V2/V3 équivalent.
- Les dernières sont, à peu de choses près, celles du pré-compilateur LUSTRE V2, elle terminent les vérifications et produisent le programme noyau équivalent.

Eclater complètement un programme peut sembler peu judicieux étant donné qu'avec les extensions proposées, un programme de taille importante peut être très facilement écrit. Mais, les objectifs principaux étant :

- En premier, la production d'un circuit synchrone, c'est à dire un ensemble de points mémoires et d'équations logiques.
- En second, un pré-processeur pour les outils existant qui ne connaissent que LUSTRE V3. Par effet de bord, cela fournit donc, pour LUSTRE V4, un générateur de code séquentiel optimisé [Ray91], un générateur de code oc [PS87], le format intermédiaire commun partagé par LUSTRE et ESTEREL ou un outil de vérification de propriétés [Rat92] .

Les principaux buts recherchés lors de la réalisation de cette première version de POLLUX ont été de définir le processus de compilation et valider les choix faits pour le langage.

L'orientation générale choisie a donc été de définir les algorithmes les plus simples possibles en essayant juste de garder la structure des programmes le plus longtemps possible afin de réduire les temps de compilation.

9.3 Programme correct

Avant de regarder plus avant la structure du compilateur, nous allons préciser la notion de programme LUSTRE correct. Dans la sémantique donnée au chapitre précédent, nous avons cherché avant tout à définir quel sens pouvaient avoir les programmes LUSTRE corrects, plutôt que dire lesquels l'étaient. Dans le soucis de rendre cette sémantique plus simple, nous avons donné un sens à n'importe quel programme dont il était possible de définir les environnements associés.

Dans la pratique, certains programmes, bien qu'ayant un comportement parfaitement défini, sont considérés comme incorrects. La raison principale est que, vouloir traiter le plus grand nombre possible de programmes alourdirait considérablement, et à tous les niveaux, la tâche du compilateur.

En adoptant ce principe, certaines vérifications ont ainsi pu être grandement simplifiées :

Déclaration : Tout objet utilisé dans un programme doit être déclaré. Le type d'un identificateur n'est jamais déduit de sa définition.

Double définition : Tout programme dont une des variables est définie plusieurs fois est rejeté, même si toutes les définitions sont équivalentes. Le programme comportant les deux équations suivantes est incorrect :

```
a = 3;
a = 3;
```

Absence de définition : Tout programme dont une des variables n'est pas complètement définie est rejeté. C'est, par exemple, une variable pour laquelle aucune équation n'a été donnée ou un tableau dont un élément n'est pas défini.

Boucle combinatoire : Aucun type, variable, constante ou statique ne peut être défini en fonction de lui même, même si l'équation admet une solution. L'équation booléenne `x = true or x` admet pour solution `x = true` mais le programme la contenant est rejeté.

9.4 Exemple

Comme nous l'avons dit plus haut, les algorithmes employés par le compilateur ne comportent pas de difficulté majeure, Les principaux problèmes qui ont du être résolus ont été les définitions de :

- La succession d'opérations à effectuer pour passer d'un programme LUSTRE V4 à un programme LUSTRE V3.
- L'ordre dans lequel ces opérations doivent être réalisées pour que chacune ait à sa disposition toutes les informations dont elle a besoin.

Décrire en détail tous les algorithmes ne présenterait donc pas un grand intérêt. Nous allons plutôt essayer de donner une idée globale du déroulement d'une compilation. Pour nous y aider nous utiliserons l'exemple décrit ci-dessous.

```
const size = 4;
type tni = bool^4;

node Count1 (bits: tni^2)
returns (sum :int^2);
let
   assert ((sum > 0^2) and (Select(bits,0^2) = 0^2));
   sum = RecCount(size^2, bits);
end;

node RecCount(const n:int; bits: bool^n)
returns (sum:int);
let
   sum = (if bits[0] then 1 else 0) +
            with n = 1 then 0 else  RecCount(n-1, bits[1..n-1]);
end;

node Select(array:tni; const index:int)
returns (bit:bool);
let
   bit = array[index];
end;
```

Le noeud `Count1` reçoit un tableau de vecteurs de bits et retourne un tableau d'entiers contenant le nombre de bits à un de chacun des vecteurs. Le calcul du nombre de un est réalisé récursivement par le noeud `RecCount`. Le noeud `Select` est utilisé pour accéder au premier bit de tous les vecteurs. En effet l'opérateur de sélection s'applique à un tableau et non à ses éléments, il permet uniquement, dans notre cas, de sélectionner un ou plusieurs éléments de types `tni`. L'extension homomorphe de `Select` nous permet d'appliquer la seconde partie de la sélection sur chaque élément de `bits`.

9.5 Construction de la structure de données

L'analyse syntaxique d'un programme est tout à fait classique [ASU86], elle produit :

- Son arbre abstrait qui, en fonction de la syntaxe donnée au langage, décrit la structure du programme, sous une forme plus aisément manipulable par le compilateur que le texte source.
- Trois tables des symboles donnant respectivement les noms des
  - opérateurs.
  - types.
  - variables, constantes et statiques

  déclarés ou utilisés dans le programme.

Un seul travail est fait sur l'arbre abstrait, il consiste à marquer, dans la table des opérateurs, ceux qui sont utiles, c'est à dire le noeud principal et ceux qu'il appelle directement ou indirectement. A partir de là, le compilateur ne considère plus que les déclarations de noeuds marquées. Ceci réduit sa tâche au strict nécessaire et ne pénalise pas les utilisateurs de bibliothèques de composants de base.

La représentation interne d'un programme LUSTRE, manipulée par le compilateur, consiste en un réseau d'opérateurs par noeud du programme et un pour les déclarations globales. Ces réseaux sont construits à partir de trois types d'éléments :

- Des boîtes, correspondants aux opérateurs prédéfinis ou aux appels de noeuds.
- Des fils permettant de connecter les opérateurs entre eux
- Des identificateurs, représentant les variables ou les constantes associés aux fils

La Figure 9.1 représente les réseaux d'opérateurs produits pour notre exemple. Ces schémas, comme la plupart de ceux des chapitres précédent sont produits par le compilateur à partir d'un programme LUSTRE puis arrangés manuellement sous un éditeur de dessin. Nous reviendrons plus en détail sur la façon dont ces schémas sont produits dans un des chapitres suivants.

La représentation des réseaux d'opérateurs est un peu simplifiée :

- Les fils ayant même nom sont identiques, ils ont juste été découpés pour rendre le dessin plus lisible.
- A chaque réseau est associé un ensemble de tables associant un identificateur à un objet.

La construction des réseaux est réalisée en deux étapes :

- Dans la première, les différentes tables d'identificateurs sont construites, seuls les déclarations et les en-têtes de noeuds sont considérés. C'est pendant cette opération que sont détectées les doubles définitions, lorsqu'un identificateur est ajouté à une table dans laquelle il se trouve déjà.
- Dans la seconde, les réseaux d'opérateurs et les expressions de types sont construits. C'est pendant cette opération que sont détectés les identificateurs non déclarés, chaque fois qu'un identificateur utilisé dans une expression ne se trouve pas dans la table adéquate (des variables ou des types selon la nature de l'expression).

9.6 Typage

L'opération faite en tout début de compilation consiste à typer les réseaux d'opérateurs :

- Cela permet de détecter dès le début un grand nombre de programmes incorrect.
- La plupart des autres phases de la compilation ont besoin de ces informations.

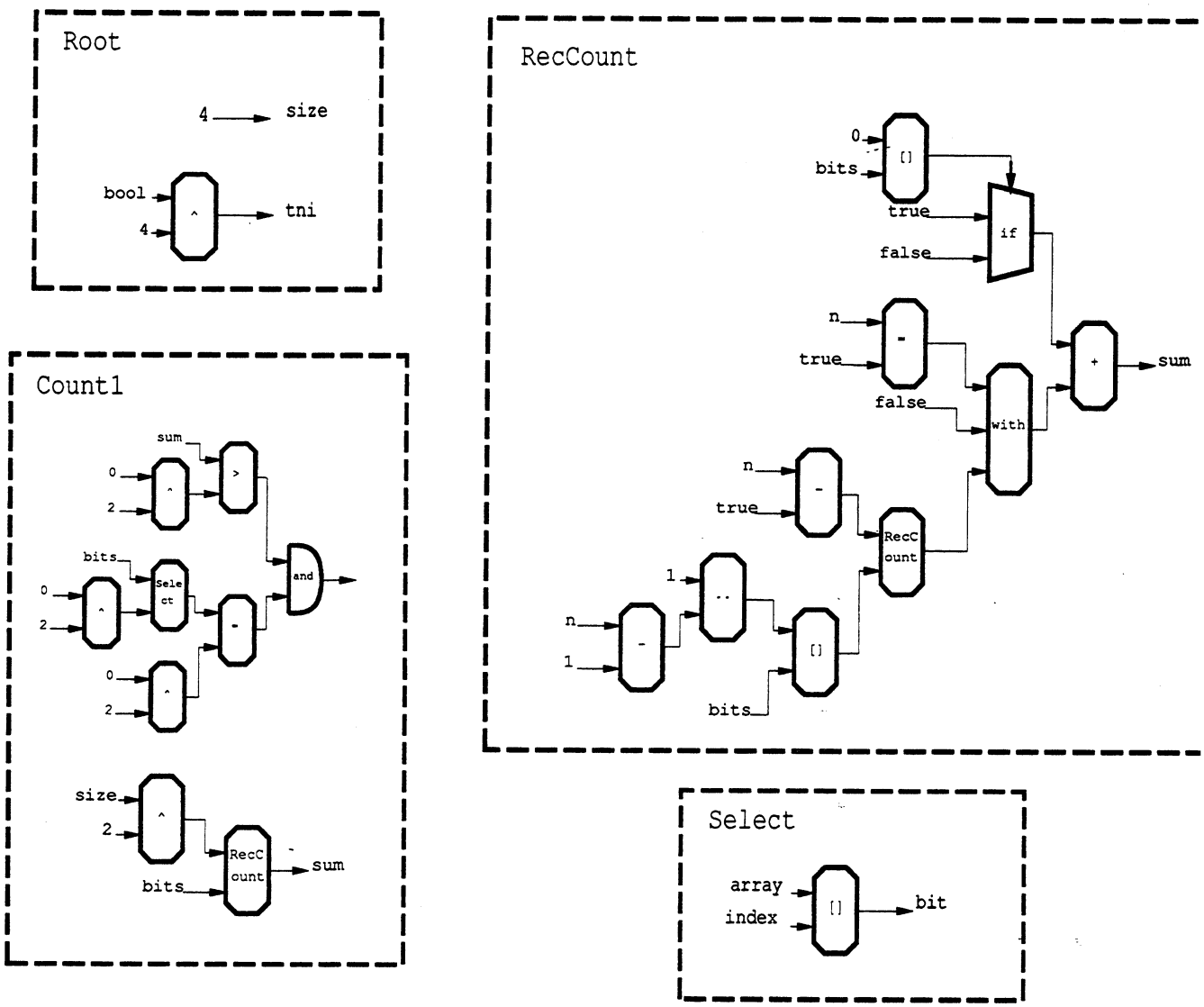


Figure 9.1 *Réseaux d'opérateurs produits pour le programme* `Count1`

Dans la sémantique du langage, nous avons donné les règles de calcul des types. Il reste maintenant à décrire comment sont organisés ces calculs.

Au préalable, afin de résoudre le problème posé par la surcharge de 0 et 1, pouvant désigner à la fois une constante booléenne et une constante entière, les types de toutes leurs instances sont déterminés en fonction des expressions dans lesquelles elles apparaissent. Celles dont le type est `bool` sont remplacées par la constante booléenne correspondante. A partir de ce moment 0 et 1 ne désignent plus que des constantes entières

Le noeud `Count1` de notre exemple devient donc après cette phase

```
node Count1 (bits: tni^2)
returns (sum :int^2);
let
   assert ((sum > 0^2) and (Select(bits,0^2) = false^2));
   sum = RecCount(size^2, bits);
end;
```

La dernière instance de 0 dans l'assertion est remplacée par `false`, l'autre entrée de l'opérateur qui en dépend est booléenne.

Décider du type exact de 0 ou de 1 n'est pas toujours possible, dans l'expression `0 = 0` par exemple. En fait ces cas pathologiques ne se produisent qu'avec les opérateurs polymorphes dont le type d'une entrée ne dépend que de celui d'une autre, il s'agit pour les opérateurs prédéfinis, uniquement de = et <>. La solution au problème consiste à choisir le même type pour les deux opérandes. Dans le cas des autres opérateurs polymorphes,`pre`, `->` ou `if`, le type est hérité de celui attendu pour le résultat.

Le principal problème du calcul de types vient du fait que le typage d'un réseau est fonction des valeurs de ses entrées statiques. Il est donc nécessaire de déterminer, dans le programme, les différentes valeurs des entrées statiques avec lequel un réseau est instancié puis d'effectuer le calcul des types pour chacune. Pour déterminer ces valeurs il faut également définir complètement l'arbre d'appel.

Le réseau associé aux déclarations globales est le premier à être typé. Il s'agit de faire un tri topologique des déclarations de types de façon à s'assurer qu'il n'y a aucune définition cyclique.

Le réseau du noeud principal est typé ensuite. Dans un programme correct celui-ci ne comporte aucune entrée statique. Son typage va consister à associer un type à tous les fils du réseau, connaissant :

- Ceux des identificateurs et par conséquent ceux des fils auxquels ils sont associés.
- Les fonctions, implémentations directes des règles de typage, calculant le type des fils de sortie d'un opérateur en fonction de ceux de ses entrées.
- La fonction qui calcule le type des sorties d'un appel de noeud a partir des types de ses entrées et des valeurs de ses entrées statiques.

Les erreurs de type sont détectées à deux endroits :

- Au niveau de chaque opérateur, lorsqu'une de ses entrées a un type incompatible avec sa définition.
- Au niveau d'un fil nommé, lorsque le type synthétisé par sa source et celui associé a l'identificateur ne sont pas équivalents.

Une fonction supplémentaire est nécessaire, celle calculant la valeur d'une expression statique, pour déterminer les tailles des dimensions de tableaux par exemple. Il s'agit sans doute de la fonction la plus délicate de tout le compilateur puisqu'elle doit parcourir un réseau, qui n'est pas vérifié ni même complètement typé, pour essayer de déterminer une valeur associée à un fil. Cette fonction doit donc faire différentes vérifications pour s'assurer que l'expression qu'elle tente d'évaluer est correcte.

Cette fonction, appliquée à un fil retourne soit la valeur qui lui est associée, soit un constat d'échec si l'expression ne peut être évaluée. Ce dernier cas se produit si l'expression n'est pas statique, si son type n'est pas évaluable ou s'il existe une boucle de causalité, une variable statique définie en fonction d'elle-même.

Le calcul des genres est réalisé de façon tout a fait similaire.

Une fois le typage du noeud principal fait, l'étape suivante consiste à évaluer les opérateurs `with` et les remplacer par l'opérande sélectionné. Ensuite, chaque appel de noeud est traité, ses entrées statiques sont évaluées. S'il n'existait pas déjà, un nouveau réseau est crée, copie de celui appelé, dans lequel les entrées statiques ont été remplacées par leur valeurs.

Le traitement qui vient d'être décrit est ensuite récursivement appliqué aux réseaux nouvellement créés. L'arbre d'appel pouvant être infini avec l'introduction de la récursivité, il est important de pouvoir s'arrêter. Un moyen pragmatique d'éviter au compilateur de boucler est utilisé, le traitement rigoureux de la terminaison d'une récursivité étant un problème indécidable. Ce moyen consiste à compter le nombre d'appels imbriqués et de s'arrêter dès que celui-ci passe un certain seuil, spécifié par l'utilisateur .

Après la phase de typage, notre exemple devient :

```
const size = 4;
type tni = bool^4;

node Count1 (bits: bool^4^2)
returns (sum :int^2);
let
   assert ((sum > 0^2) and (Select_0(bits) = false^2));
   sum = RecCount_4(bits);
end;

node RecCount_4(bits: bool^4)
returns (sum:int);
let
   sum = (if bits[0] then 1 else 0) + RecCount_3(bits[1..3]);
end;
```

```
node RecCount_3(bits: bool^3)
returns (sum:int);
let
   sum = (if bits[0] then 1 else 0) + RecCount_2(bits[1..2]);
end;

node RecCount_2(bits: bool^2)
returns (sum:int);
let
   sum = (if bits[0] then 1 else 0) +  RecCount_1(bits[1..1]);
end;

node RecCount_1(bits: bool^1)
returns (sum:int);
let
   sum = (if bits[0] then 1 else 0) + 0;
end;

node Select_0(array:bool^4)
returns (bit:bool);
let
   bit = array[0];
end;
```

Tous les noeuds paramétrés ont été instanciés, les opérateurs `with` supprimés et les noms de types, qui sont des alias remplacés par leur définition.

9.7 Autres vérifications

Deux vérifications supplémentaires sont réalisées pour s'assurer que les variables sont correctement définies et utilisées. Elles consistent à étudier les sources et les destinations de chaque fil associé à une variable.

Définition unique : Il s'agit de vérifier que toute variable, ou chacun de ses éléments dans le cas d'un tableau ou d'une structure, est définie une et une seule fois.

- Le fil associé à une variable doit avoir exactement une source.
- Un fil associé à un tableau ou à une structure possède plusieurs sources, définissant chacune un sous-ensemble de ses éléments. Chaque élément doit être défini par exactement une de ces sources.

Sélection Valide : Il s'agit de vérifier que toutes les sélections portent bien sur des éléments appartenant à la structure ou au tableau correspondant.

L'évaluation des horloges est également faite à ce point de la compilation. Une description du nouveau calcul d'horloge effectué est donnée au chapitre 1. Le résultat de cette opération associe

une horloge à chaque fil des réseaux. La détection d'une incohérence d'horloge est réalisée comme pour le calcul des types au niveau des opérateurs et des fils nommés.

9.7.1 Extension homomorphe

La seconde partie du processus de pré-compilation débute maintenant. La structure de données est maintenant construite et la plupart des vérifications faites. Le travail consiste maintenant à effectuer la traduction du programme LUSTRE V4 en son équivalent LUSTRE V3.

En premier, chaque instance d'un opérateur, utilisant son extension homomorphe, est expansée. Il suffit pour cela d'appliquer la définition de l'extension homomorphe d'un opérateur(§ 3.7).

Sur le réseau, l'opérateur homomorphe est remplacé par le réseau :

- destructurant chaque entrée en éléments ayant le même type que le paramètre formel correspondant de l'opérateur.
- appliquant chaque vecteur d'éléments à une instance "normale" de l'opérateur.
- restructurant les sorties.

Dans notre exemple, `RecCount_4` (c'est le cas également de `Select_0`) était appliqué à un tableau de deux paramètres. La suppression de l'homomorphisme a remplacé cet appel par deux, un pour chaque élément, et regroupé les résultats produits dans un autre tableau.

```
...

node Count1 (bits: bool^4^2)
returns (sum :int^2);
let
   assert [(sum[0] > 0) and (Select_0(bits[0]) = false),
           (sum[1] > 0) and (Select_0(bits[1]) = false)];
   sum = [RecCount_4(bits[0]), RecCount_4(bits[1])];
end;

...
```

9.8 Elimination de la structuration

La deuxième étape consiste à éclater les tableaux et structures, mais également à évaluer les opérateurs qui permettent de les manipuler.

En premier lieu, toute variable de type structuré, est remplacée par autant de variables qu'elle a d'éléments de type élémentaire. La convention utilisée pour nommer chacun d'eux consiste à suffixer le nom de la variable, par la succession d'indices et de labels permettant de le sélectionner.

De la même façon chaque fil structuré est remplacé, suivant son type, par une nappe de fils de type élémentaire.

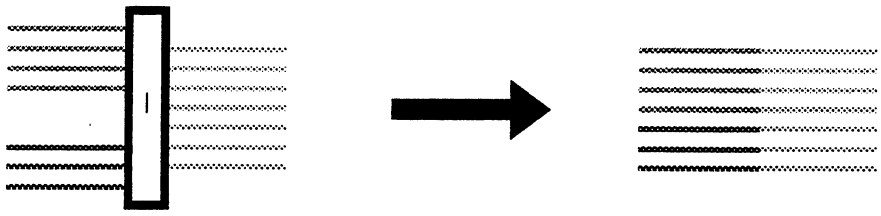


Figure 9.2 *Evaluation d'un opérateur de concaténation*

Lors de l'étape suivante les opérateurs de manipulation de tableaux et de structures sont évalués. Cela consiste à connecter, en fonction de leur nature, les nappes de fils de leurs entrées avec celles de leurs sorties.

L'exemple le plus simple est l'opérateur de concaténation, appliqué à deux tableaux monodimensionnels. Il reçoit deux nappes, respectivement de m et n éléments en entrée et en produit une de m+n éléments en sortie. Son évaluation effectue la connexion comme représenté Figure 9.2 du i^{ieme} de la première nappe d'entrée au i^{ieme} de celle de sortie et i^{ieme} de la seconde au m+i^{ieme} de celle de sortie.

Les nappes sont hiérarchisées. Si les éléments des tableaux concaténés sont eux-mêmes des tableaux, les éléments des nappes sont des nappes. La connexion de deux nappes consiste à connecter récursivement ses membres deux à deux.

La notion de nappe n'est qu'une information utilisée lors de l'évaluation des opérateurs de manipulation de tableaux et de structures, elle disparaît une fois cette tâche terminée, les fils d'une nappe devenant indépendants les uns des autres.

Voici le résultat de cette opération sur notre exemple :

```
...

node Count1 (bits_0_0: bool; bits_0_1: bool; bits_0_2:bool; bits_0_3: bool;
   bits_1_0: bool; bits_1_1: bool; bits_1_2: bool; bits_1_3: bool)
returns (sum_0: int; sum_1: int);
let
   assert (sum_0 > 0) and
      (Select_0(bits_0_0, bits_0_1, bits_0_2, bits_0_3) = false)
   assert (sum_1 > 0) and
      (Select_0(bits_1_0, bits_1_1, bits_1_2, bits_1_3) = false)
   sum_0 = RecCount_4(bits_0_0, bits_0_1, bits_0_2, bits_0_3);
   sum_1 = RecCount_4(bits_1_0, bits_1_1, bits_1_2, bits_1_3);
end;

node RecCount_4(bits_0: bool; bits_1: bool; bits_2:bool; bits_3: bool)
returns (sum:int);
let
```

```
   sum = (if bits_0 then 1 else 0) + RecCount_3(bits_1, bits_2, bits_3);
end;

...
```

9.9 Traduction en langage noyau

Nous arrivons maintenant à la seconde phase de traduction de LUSTRE V2/V3 en langage noyau.

9.10 Expansion

L'expansion consiste à remplacer chaque noeud par sa définition. Quelques opérations supplémentaires sont réalisées en plus de la simple copie du réseau.

L'horloge de base d'une instance de noeud est celle de son appel que nous noterons `h`. Lors de la copie du noeud, il s'agit de remplacer toute référence à sa propre horloge de base par `h`. Mais cela n'est pas encore suffisant, car l'horloge des constantes est implicitement `true`, il est donc nécessaire, lors de la copie du noeud, de remplacer toute occurrence d'une constante `c` par l'expression `c when h` de façon à ce qu'elle soit échantillonnée sur la bonne horloge. Pour la même raison, toute occurrence d'un noeud sans entrée `N()` doit être remplacée par l'expression `N() when h`).

Toutes les variables du noeud expansé deviennent des variables locales dans le noeud appelant. Mais, les variables d'entrées et de sortie d'un noeud servent avant tout à décrire l'interface d'un noeud, et il est peu judicieux de les conserver lorsque ce n'est pas nécessaire. Si l'on considère l'exemple suivant,

```
node invert (a:bool)
returns (b:bool);
let
   b = not a;
end;

...
x = y and invert(z);
...
```

le résultat de l'expansion du noeud `invert` désiré est

```
x = y and not z;
```

et non

```
x = y and b;
b = not a;
```

Ces deux programmes sont fonctionnellement équivalents mais en conservant les variables a et b, la structure de la définition de x est modifiée, elle est coupée en deux. Aussi lors de l'expansion, les variables d'entrée ou de sortie sont détruites chaque fois que c'est possible.

L'expansion de notre exemple donne finalement le résultat suivant :

```
node count1 (bits_0_0: bool; bits_0_1: bool; bits_0_2: bool; bits_0_3: bool;
   bits_1_0: bool; bits_1_1: bool; bits_1_2: bool; bits_1_3: bool)
returns (sum_0: int; sum_1: int);

let
  assert ((sum_0 > 0) and (bits_0_0 = false));
  assert ((sum_1 > 0) and (bits_1_0 = false));
  sum_0 = ((if bits_0_0 then 1 else 0) + ((if bits_0_1 then 1 else 0) + ((if
  bits_0_2 then 1 else 0) + ((if bits_0_3 then 1 else 0) + 0))));
  sum_1 = ((if bits_1_0 then 1 else 0) + ((if bits_1_1 then 1 else 0) + ((if
  bits_1_2 then 1 else 0) + ((if bits_1_3 then 1 else 0) + 0))));
end;
```

9.11 Interblocage

Les vérifications précédentes ont permis de s'assurer que le programme était correctement construit ; la dernière vérification permet de s'assurer qu'il est bien exécutable. Elle consiste à détecter si le calcul d'une variable ne dépend pas d'elle-même. Cette vérification ne s'intéresse qu'aux dépendances statiques, une variable dépend d'une autre si elle en a potentiellement besoin pour être calculée. Par conséquent, certains faux interblocages sont détectés. Dans l'exemple suivant, un interblocage est détecté car a a potentiellement besoin de b et vice-versa :

```
a = if x then b else c;
b = if not x then a else d;
```

Mais dans la réalité, à cause des valeurs opposées des conditions, il n'y a jamais dépendance mutuelle. Dans la plupart des cas, il existe un moyen de réécrire un tel programme pour supprimer le problème. Dans l'exemple ci-dessus, il suffit de substituer, dans les deux parties droites, a et b par leur définition puis de simplifier, pour obtenir un programme sans interblocage.

```
a = if x then d else c;
b = if not x then c else d;
```

A ce niveau de la compilation, cette vérification est très facile à faire, un simple tri topologique des variables permet de conclure.

9.12 Prise en compte des extensions

La description des différentes opérations que réalise le compilateur actuel pour transformer un programme LUSTRE V4 en un programme LUSTRE V3 est maintenant terminée. Les paragraphes qui suivent donnent un aperçu des traitements supplémentaires qui devront être effectués, dans la version future du compilateur, pour prendre en compte la totalité du langage.

9.12.1 Dépendances

Les informations fournies par les dépendances ne sont utilisées que lors de la recherche d'une variable définie en fonction d'elle-même; elles sont utilisées pour compléter le graphe de dépendances construit pour faire un tri topologique des variables. Au préalable, les différentes vérifications et manipulations du réseau sont également appliquées aux dépendances, de façon à obtenir des dépendances entre variables de type élémentaire.

9.12.2 Assertions statiques

L'intérêt principal des assertions statiques est de pouvoir décrire des invariants sur les noeuds, et spécifier quels sont les ensembles de valeurs que peuvent prendre leurs entrées statiques. Une application évidente est d'indiquer les conditions d'utilisation d'un noeud récursif. Par conséquent pour que ces assertions aient une utilité quelconque, elle doivent être évaluées très tôt dans la chaîne de compilation, juste après le calcul des types.

Dans notre exemple, l'assertion statique suivante permettrait d'indiquer que le programme est incorrect si le paramètre `n` du noeud `RecCount` n'est pas strictement positif.

```
static assert n > 0;
```

9.12.3 Sorties statiques

Avec cette extension, l'évaluation des expressions statiques devient plus complexe, elle ne peut plus se faire uniquement à l'intérieur d'un réseau. Pour limiter son coût, une restriction est faite. La valeur d'une sortie statique d'un noeud ne peut être calculée que si celles de toutes ses entrées statiques l'on été préalablement. Ainsi, le calcul des sorties statiques d'un noeud est grandement simplifié, toutes les données potentiellement nécessaires étant fournies. Le calcul se fait uniquement en descendant dans la hiérarchie.

9.12.4 Structures à champs nommés

Les structures à champ nommés ne posent pas de problème particulier si ce n'est de compliquer la gestion des identificateurs.

Chapitre 10

Description et synthèse de circuits

Nous allons maintenant aborder l'utilisation de LUSTRE qui est à l'origine de son extension : La description de programmes de grande taille et leur traduction en circuits synchrones.

10.1 La PAM

La principale machine cible pour les circuits décrits en LUSTRE est la PAM [BRV90][BRV92] (pour Programmable Active Memory, Mémoire Active Programmable) , un co-processeur matériel programmable développé au PARIS RESEARCH LABORATORY de DEC. Il y a deux raisons principales à cette réalisation :

- Il existe un grand nombre d'applications dont la plus grande partie du temps d'exécution est utilisé pour calculer un très grand nombre de fois quelques opérations de base. Une solution pour améliorer leurs performances consiste à utiliser un co-processeur spécialisé pour effectuer ces opérations. L'exemple classique de ce genre de solution est un co-processeur arithmétique, utilisé pour accélérer des applications réalisant de nombreux calculs arithmétiques.

  Cette solution permet d'importantes améliorations de performances mais elle est en fait relativement peu utilisée, la principale limitation venant du coût et le temps de développement de d'un co-processeur. Très peu sont donc disponibles sur le marché, il s'agit de co-processeur réalisant des opérations suffisamment générales pour avoir un grand nombre d'utilisations possibles. Or, beaucoup d'applications nécessitent des accélérateurs spécifiques.

  Une autre limitation vient du manque de flexibilité de la solution. Lorsqu'une application est modifiée, il n'y a que deux choix pour son accélérateur :

  - Le conserver même s'il ne convient plus parfaitement et ne fournit plus un aussi bon gain de performances .
  - En refaire un autre et donc recommencer tout la chaîne de conception .

- Pour apporter une réponse au problème du coût de conception d'un circuit, de nombreux nouveaux types de composants sont apparus. Il fournissent un certains nombre

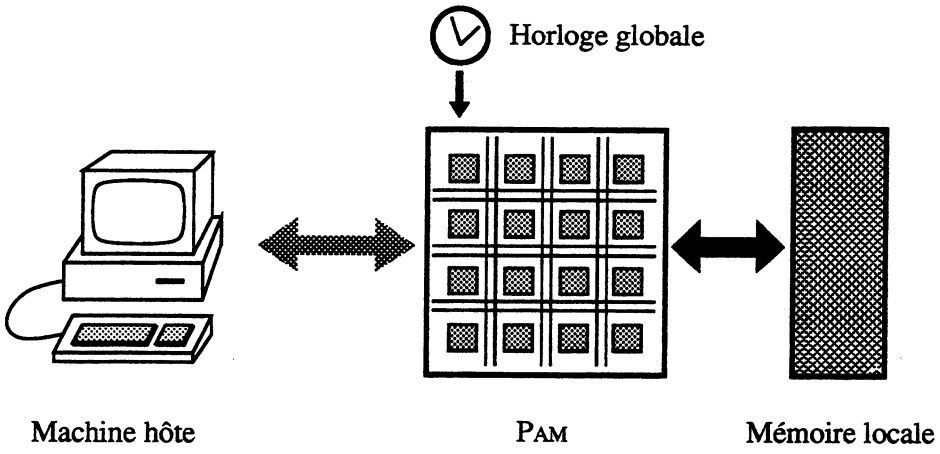


Figure 10.1 *Architecture générale d'une* PAM

de ressources prédéfinies qu'il ne reste plus qu'à configurer. Plus récemment encore sont apparus des circuits re-programmables, dont la configuration n'est plus fixée une fois pour toute, mais chargée à chaque initialisation.

La PAM a donc été développée, à partir de ces circuits re-programmables, pour simplifier et étendre l'utilisation d'accélérateurs programmables. Son architecture générale est représentée Figure 10.1 ; elle est constituée de trois principaux éléments :

- Un ensemble de cellules élémentaires, dont la structure exacte dépend de l'implémentation de la PAM. Elles sont cependant suffisamment générales pour être configurées en une fonction logique de quelques variables ou un point mémoire.
- Un ensemble de ressources d'interconnexions qui sont utilisées pour relier les cellules entre elles.
- Une horloge globale. Les applications implantées sur la PAM sont des circuits synchrones, tous les points mémoires mettent leurs valeurs à jour à chaque top de cette horloge.

La PAM est également dotée de mémoire locale pour stocker ses résultats intermédiaires. L'ensemble est connecté, au travers d'un bus, à une machine hôte qui a deux rôles :

- Il configure la PAM en lui envoyant une séquence de bits décrivant la configuration de chaque cellule (l'opération qu'elle doit réaliser), la structure du réseau d'interconnexions ainsi que la fréquence de l'horloge.
- Il contrôle l'exécution, il envoie des données et des commandes, demande des résultats. La PAM est un processeur esclave, il ne peut accéder au bus que s'il y est autorisé.

L'intérêt de la PAM est multiple :

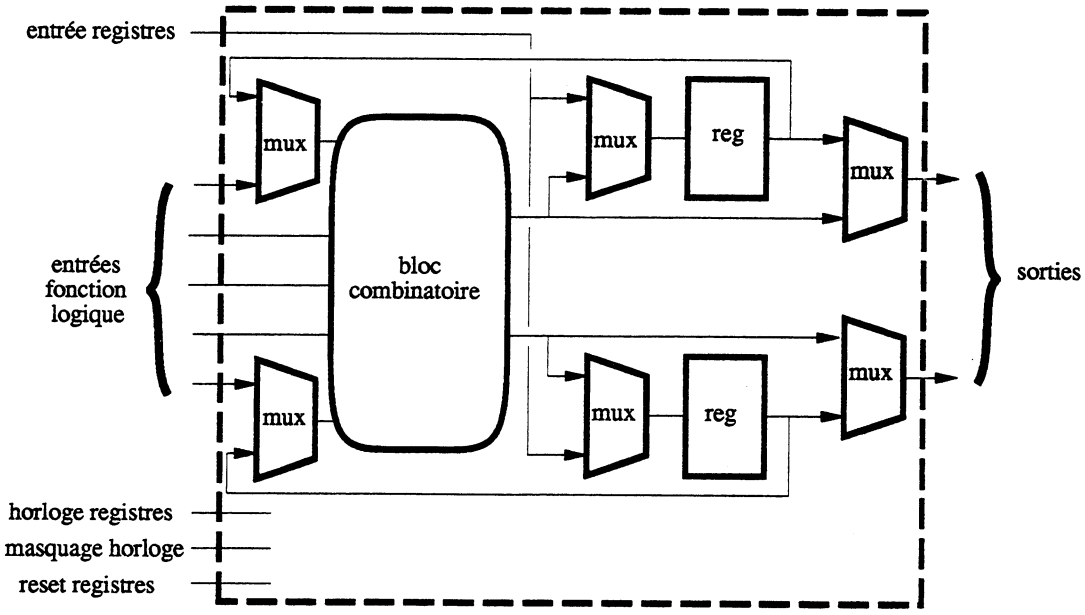


Figure 10.2 *Bloc logique configurable d'un* XILINX *série 3000*

- Elle réduit les temps et coût de conception de circuits. Elle peut d'ailleurs également être utilisée comme outil de prototypage.
- Une seule PAM est nécessaire pour réaliser n'importe quel nombre de circuits.
- Sa flexibilité permet de déterminer expérimentalement la configuration permettant la meilleure accélération d'une application exécutée sur l'hôte. Il suffit de re-configurer la PAM pour modifier son comportement.

Il y a actuellement deux versions de la PAM, PERLE0 et PERLE1, toutes deux construites à partir d'une matrices de circuits XILINX de la série 3000. La structure de la cellule est représentée Figure 10.2 , elle se compose principalement :

- D'un bloc combinatoire permettant de réaliser n'importe quelle fonction de cinq variables ou deux fonctions de quatre variables.
- De deux points mémoire d'un bit.
- De deux sorties à choisir parmi les quatre possibles.

PERLE0 est une PAM de 40x40 cellules, configurée en 500 ms par une séquence de 400 K bits ; sa mémoire locale est de 4 M octets. PERLE0 est connectée à un machine à base de MC68020 au travers d'un bus VME. PERLE1 est une PAM quatre fois plus grosse ayant pour hôte une machine à base de R3000. De la logique a été également ajoutée afin de simplifier les interfaces entre la PAM et sa mémoire locale ou l'hôte.

10.2 Informations Complémentaires

Le moyen initial de description d'une application pour la PAM consistait à utiliser un ensemble de fonctions LISP, chacune correspondant à un élément d'un circuit XILINX, pour donner une description de chacune de ses cellules. Des langages adaptés et de plus haut niveau ont été recherchés, les raisons principales étaient :

- Rendre la description d'applications aussi indépendante que possible de l'implémentation exacte de la PAM.
- Pouvoir facilement constituer des bibliothèques générales de composants de base.
- Avoir des descriptions simples à comprendre et à modifier

LUSTRE et ESTEREL, se sont révélés être deux bons candidats, une fois encore parfaitement complémentaires :

- Le premier pour décrire les chemins de données, c'est à dire les parties relativement régulières.
- L'autre pour décrire les contrôleurs, c'est à dire les parties chargées du séquencement de l'application, de taille en général beaucoup plus réduites mais plus difficiles à mettre au point du fait de leur absence de toute régularité.

Le principal défaut de la PAM que l'on pourrait avancer est, en raison de la technologie utilisée, sa relative lenteur par rapport à des circuits plus traditionnels. Ce défaut est en partie éliminé par la souplesse d'utilisation de cette approche qui facilite la recherche de chemins critiques et permet ainsi de réaliser une optimisation beaucoup plus poussée des circuits. La période d'horloge d'une application implantée sur la PAM est généralement comprise entre 30 et 100 ns, ce qui est obtenu uniquement avec des circuits ayant peu de niveaux de logique (le nombre maximum de portes combinatoires traversées) et des connexions courtes.

Les optimiseurs logiques basés sur des techniques de BDD [STB91] [CBM89] apportent une réponse à ce problème, mais étant donné le coût de ces méthodes potentiellement exponentiel, uniquement uniquement utilisable sur des circuits de taille modeste. C'est la solution utilisée par ESTEREL.

La taille des chemins de données interdit, au moins de façon globale, le recours à cette solution. La réduction des chemins critiques est, dans ce cas, réalisée en employant des techniques de retemporisation [LS81] [LS86], ou de duplication de logique. Ces opérations sont surtout réalisées manuellement, faisant souvent entrer en jeu des critères difficilement automatisables. Cela impose que les descriptions de circuits doivent être facilement manipulables, les différentes transformations de programmes aisées

Les performances désirées imposent également un placement et un routage optimal des chemins de données. Mais la taille de ces derniers et les contraintes imposées par l'architecture, les ressources de connexions limitées par exemple, ne permettent pas aux outils automatiques de placement et de routage d'obtenir dans tous les cas les résultat désirés. Il est donc primordial qu'un langage, utilisé pour décrire les chemins de données, donne au concepteur un contrôle

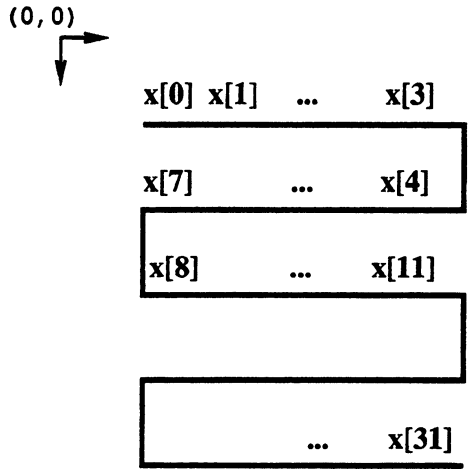


Figure 10.3 *Placement du tableau x*

complet sur l'implantation de ses circuits et puisse par exemple, donner des informations de placement.

Ceci est une des raisons du choix de LUSTRE, ses primitives sont suffisamment simples et proches de celles utilisées en conception de circuit pour fournir un bon contrôle sur l'implantation. Afin de le rendre total, de nouvelles extensions ont été ajoutées au langage.

10.2.1 Equations de placement

Les informations sur la topologie d'un circuit sont données, dans le programme source, sous la forme d'un système d'équations indépendant de celui définissant son comportement. Ce système permet d'attacher une position (sous la forme d'un couple d'entiers), dans une grille 2D infinie, à certaines variables du programme ou, s'il s'agit de tableaux ou de structures, à certains de leurs éléments.

Une équation de placement peut prendre deux formes :

- `x at (n1, n2)` place la variable x de façon absolue au point de coordonnées (n1, n2).
- `x at (n1, n2) from y` place la variable x relativement à la position de la variable y. Si les coordonnées de cette dernière sont (k1, k2), alors x est placé au point de coordonnées (n1+k1, n2+k2).

Considérons par exemple le tableau x, de type `bool^32` que l'on désire placer comme représenté sur la Figure 10.3 .

Ce placement est décrit par les équations :

```
x[0] at (2, 2);
```

```
x[1..3] at (2, 0) from x[0..2];
x[4..7] at (0, 1) from x[3..0];
x[8..31] at (0, 2) from x[0..23];
```

La signification des ces équations est donnée ci-dessous :

- L'élément d'indice 0 de `t` est placé au point de coordonnées (2, 2).
- Ceux d'indices 1 à 3 sont placés respectivement aux points de coordonnées (4, 2), (6, 2) et (8, 2) c'est à dire aux coordonnées de l'élément d'indice inférieur après un translation de vecteur (2, 0).
- Ceux d'indices 4 à 7 sont placés au dessous des quatre éléments précédents mais dans l'ordre inverse.
- Le motif qui vient d'être réalisé avec les huit premier éléments, est répété pour les vingt quatre restants mais translaté de deux vers le bas à chaque fois.

Les équations définissant le placement des variables sont traités exactement comme de celles donnant leur comportement. Les vérifications et manipulations décrites précédemment(§ 9) s'y appliquent.

10.2.2 Pragma opaque

Ce pragma permet d'interfacer LUSTRE avec des bibliothèques de composants. Il peut sembler bizarre d'ajouter au langage un mécanisme qui existe déjà, puisqu'un programme LUSTRE peut importer des opérateur externes. Mais, les éléments de bibliothèques ne sont définis que pour un unique post-processeur ; si l'on désire simuler un programme utilisant de tels opérateurs, il est nécessaire décrire pour chacun d'entre eux le morceau de code séquentiel simulant leur comportement. La même chose doit être réalisée pour n'importe quel autre post-processeur. Cette approche présente deux inconvénients :

- Ces circuits élémentaires se décrivent en général mieux en LUSTRE que dans certains autres langages.
- Il est nécessaire d'écrire autant de fonctions que de post-processeurs utilisés. Ceci est non seulement pénible mais également source d'erreurs.

Le but de ce pragma est de cacher à certains post-processeur la définition de noeuds qui sont alors considérés comme externes. Interfacer une bibliothèque de composants consiste alors à écrire chacun d'entre eux par un noeud LUSTRE et dont la définition est cachée au processeur utilisateur de la bibliothèque.

Un pragma `opaque` est attaché à un identificateur de noeud :

```
node foo {opaque: bar} ...
```

Ici, le noeud `foo` est considéré comme externe pour le processeur `bar`.

10.2.3 Attributs

La partie du langage ajoutée pour décrire la topologie d'un circuit s'est révélée, à l'usage, parfaitement adaptée. Aussi sera élle étendue dans la version future du compilateur l'introductiondu pragma attribute que nous allons maintenant décrire, et qui permettra de donner des informations plus variées.

Ce pragma est inspiré VHDL [SLM+85], il permet d'attacher des informations aux variables d'un programme ou, s'il s'agit de tableaux ou de structures, à leurs éléments. Ces informations étant spécifiques à la description de circuits voire à un outils particulier, elles sont données sous forme de pragmas, afin d'être ignorées par les outils n'en ayant pas l'usage.

Les différents attributs qui vont être attachés aux variables sont tout d'abord définis dans l'environnement global par une série de déclarations :

```
{attribute: a : t = e}
```

Elle définit un attribut nommé `a`, de type `t`, et dont la valeur par défaut est celle de l'expression statique `e`. Par exemple, si l'on désire pouvoir donner dans le programme des informations de placement, il suffit de faire la déclaration :

```
{attribute: place : int^2 = [nil,nil]}
```

L'information ajoutée est une paire d'entiers indiquant, pour chaque variable, sa position dans une grille 2D infinie. Par défaut, la valeur de l'attribut est indéfinie, la variable n'est pas placée.

La définition de la valeur de chaque attribut est définie par un système d'équations indépendant. Une équation est de la forme :

```
{attribute: x::a = e}
```

où `x` est une variable, `a` un nom d'attribut et `e` une expression statique. Cette équation définit la valeur de l'attribut `a` de la variable `x`. Par exemple `x::place` défini la position de la variable `x`

Les attributs associés à des tableaux (resp. des structures) sont des tableaux (resp. des structures) d'attributs.
Si `x` est une structure `[l1:x1, l2:x2, ... ln:xn]` de type `[l1:t1, l2:t2, ... ln:tn]`, l'attribut `a` de `x`, noté `x::a` ou encore `[l1:t1, l2:t2, ... ln:tn]::a`, est la structure formée avec les attributs `a` de ses éléments `[l1:x1::a, l2:x2::a, ... ln:xn::a]`.

Ainsi, l'attribut `a` de `x` défini par l'équation suivante :

```
{attribute: x::a = e}
```

peut l'être également par les n équations suivantes :

```
{attribute: x.l1::a = e.l1}
{attribute: x.l2::a = e.l2}
...
{attribute: x.ln::a = e.ln}
```

Le champ `li` de l'attribut `a` de `x` est la même chose que l'attribut `a` du champ `li` de `x`. Il existe une troisième manière de définir l'attribut `a` de `x` :

```
{attribute: x::a.l1 = e.l1}
{attribute: x::a.l2 = e.l2}
...
{attribute: x::a.ln = e.ln}
```

De même, si `x` est un tableau de k dimensions, de type `t^(n1, n2, ... nk)` l'attribut `a` de `x` est un tableau de k dimensions de tailles respectives `n1`, `n2`, ... et `nk`.

Avec cette nouvelle syntaxe, le précédent exemple de placement devient :

```
{attribute:
    x::place[0] = [2, 2];
    x::place[1..3] = [2, 0]^3 + x::place[0..2];
    x::place[4..7] = [0, 1]^4 + x::place[3..0 step -1];
    x::place[8..31] = [0, 2]^24 + x::place[0..23];
}
```

10.3 Implémentation matérielle de LUSTRE

La traduction d'un programme LUSTRE booléen en un circuit synchrone a été étudiée dans [Roc89]. Nous n'allons ici reprendre que les grandes lignes de ce travail et préciser certains détails d'implémentation.

Cette traduction se résume à transformer un programme, obtenu après la phase de précompilation en un autre uniquement composé d'opérateurs logiques et de `pre`. La traduction de l'opérateur `->` impose de rajouter une entrée au programme, `init`, valant `true` à l'instant initial et `false` le reste du temps. L'expression `e -> f`, par exemple, où `e` et `f` sont deux expressions sur l'horloge de base, se traduit par l'expression `if init then e else f`

Le passage du programme ainsi obtenu au circuit synchrone correspondant est alors directe :

- Chacun de ses opérateurs logiques correspond à une porte combinatoire.
- Chacun de ses `pre` à un registre.

La seule supposition faite, est que deux signaux globaux sont disponibles :

- L'horloge qui contrôle tous les points mémoires.
- Un signal `reset`, actif à 1, à partir duquel est `init` est produit. Ce dernier est pris en sorti d'un point mémoire ayant `reset` pour entrée et `true` pour valeur initiale. La valeur initiale de `init` à `true` est facultative, elle permet au programme de démarrer dès le premier instant. Dans la pratique elle est omise, le programme est, au départ, dans un état indéterminé et il le reste jusqu'à ce qu'un reset soit fait. L'instant initial, celui où le programme débute, est celui où le signal `reset` devient inactif.

  Deux remarques :

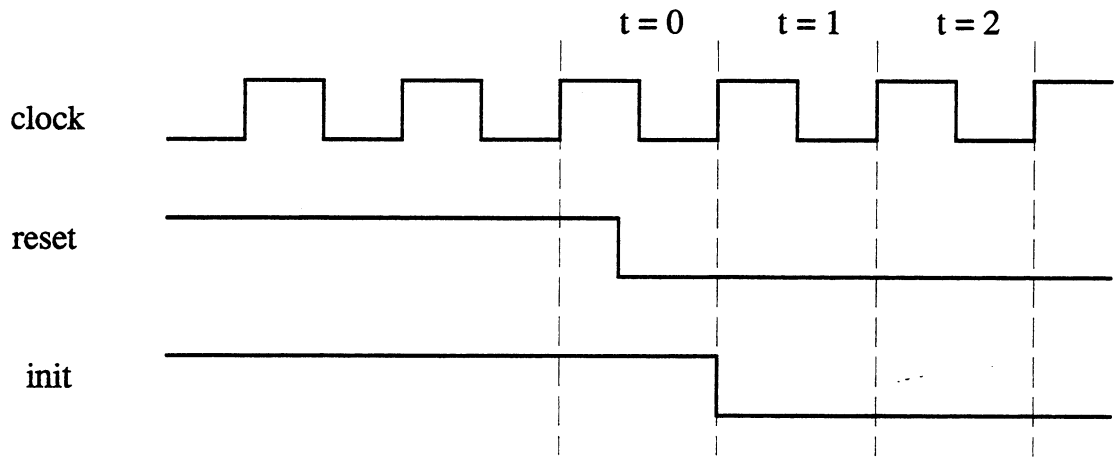


Figure 10.4 *Synchronisation du signal* **reset**

- **reset** et **clock** sont a priori deux signaux complètement asynchrones. C'est pour cette raison que le signal intermédiaire **init** est crée et fourni au programme à la place de **reset**. **init** n'est rien d'autre que le signal de reset synchronisé. (§ Figure 10.4)
- Un programme LUSTRE peut être redémarré, il suffit de mettre à **true** le signal **reset** pendant au moins un cycle. Le programme recommence dès que ce signal redevient inactif. (§ Figure 10.5)

Les points mémoire sur la PAM ont la particularité d'avoir 0 comme valeur initiale. Cette caractéristique est utilisée de deux manières lors de la génération d'un circuit pour la PAM.

- Toutes les expressions `0 -> pre x` sont traduites par un simple registre.
- Le signal **init** est crée sur la PAM à l'aide d'un point mémoire initialisé `init = not (0->pre(1))`.

  L'horloge (**clock**) ou la réinitialisation (**reset**) sont des signaux particuliers qui ont leurs propres réseaux de connexions, ce qui leurs permet d'être distribués partout sur la carte et utilisés par tous les points mémoires sans diminuer le façon notable les performances d'une application. Par contre, ce n'est pas le cas pour le signal **init** qui, à cause de cela n'est pas global sur la PAM, mais distribué. Un signal **init** local est créé pour chaque instance d'un opérateur $->$ (mis à part bien sûr celles décrites au point précédent).

Les circuits produits à partir de LUSTRE, sont décrits en sortie sous la forme d'un ensemble de cellules, éventuellement placées. Il y a deux types de cellules :

- Celles constituées uniquement d'un registre.
- Celles purement combinatoires.

Le découpage d'un programme en cellules se fait en fonction de la structure du système d'équations.

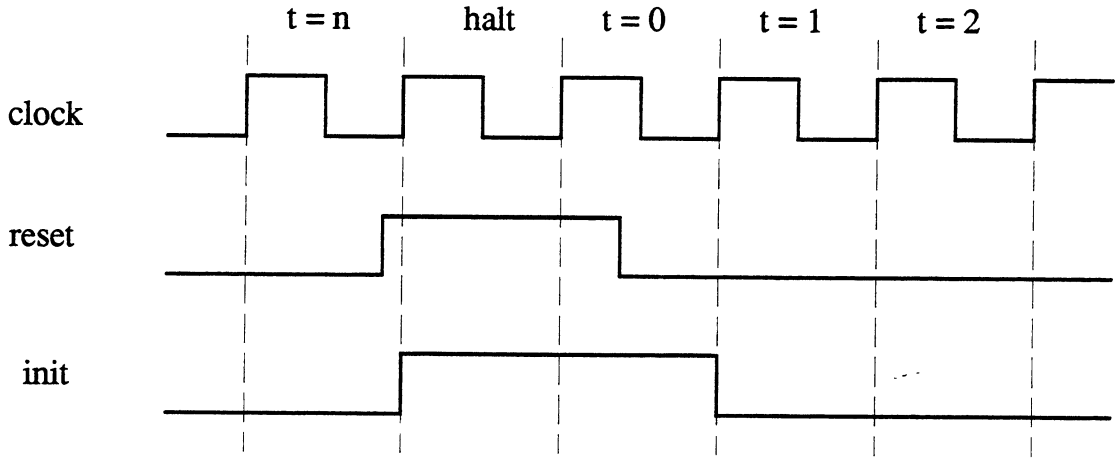


Figure 10.5 *Redémarrage d'un programme*

- Une équation dont la partie droite est uniquement composée d'opérateurs logiques constitue une cellule.
- Une équation dont la partie droite est uniquement composée d'un opérateur `pre` constitue également une cellule.
- Toute autre équation est découpée en le plus petit nombre possible d'équations des deux genres précédents.

Ces règles de construction de cellules, couplées avec la possibilité de définir leur placement donne un contrôle quasiment complet sur la structure du circuit produit. Seul le routage n'est pas complètement contrôlé, il est possible de définir les positions de la source et la destination d'une connexion mais pas de son tracé entre ces deux points. Si cela devenait nécessaire, un attribut spécifique pourrait être défini.

Considérons, comme exemple, le programme suivant qui décrit un compteur 3 bits, un opérateur qui a 000_2 pour valeur initiale et ajoute celle de sa retenue entrante `ci` à chaque instant. Nous désirons placer ce compteur dans trois cellules disposées côte à côte horizontalement à partir de (0,0). Ceci est défini en deux endroits :

- Le contenu d'une cellule est spécifié dans de noeud `Count1`. Les équations de placement indiquent que les fonctions calculant la somme `s` et la retenue `r`, ainsi que le point mémoire stockant l'ancienne valeur de `s` , `old`, sont placés au même endroit. Il est à noter que LUSTRE ne fait pas d'hypothèse sur la structure de la cellule de base, il ne vérifie pas que la cellule de la PAM est capable de contenir ces trois éléments. Les outils en aval du compilateur LUSTRE s'en chargent.
- La disposition relative des cellules est spécifiée dans le noeud principal `Count`. En plaçant les différents bits de la valeur du compteur respectivement en (0,0) (1,0) et (2,0).

Le résultat de la compilation de ce programme est représenté Figure 10.6

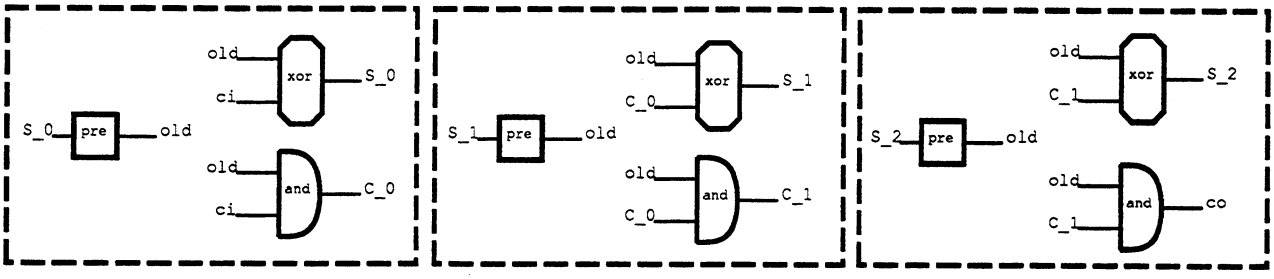


Figure 10.6 *Définition des cellules d'un compteur 3 bits*

```
node Count1 (c: bool) returns (s,r : bool);
var old:bool;
let
    r = (old and c);
    s =  old xor c ;
    old = false -> pre s;

    r at (0,0) from s;
    old at (0,0) from s;
tel;

node Count (ci:bool)
returns (S:bool^3; co:bool);
var
 C:bool^3;
let
   (S,C) = Count1([ci] | C[0..1]);
   co = C[2];

   S at (0, 0);
   S[1..2] at (1, 0) from S[0..1];
tel;
```

Le rôle du compilateur LUSTRE s'arrête là, la suite est réalisée par les outils de CAO de la PAM qui se chargent de terminer la décomposition en cellules et le placement, de faire le routage et enfin de produire les bits de configuration.

La traduction de LUSTRE en un circuit synchrone a eu deux autres applications :

- Un post-processeur permet de produire une description MODEL [GBR82] [GH85] du circuit produit à partir d'un programme LUSTRE. Cela fournit une autre moyen d'implanter les circuits produits, ce langage étant le point d'entrée d'un compilateur de silicium.

- Un autre post-processeur permet de sortir un description BLIF [Uni], un format qui est le point d'entrée d'outils de minimisation de logique et de vérification [BRAW87].

Chapitre 11

Génération de code séquentiel

Etant donné le domaine d'application initial de LUSTRE, une part importante du travail de compilation de LUSTRE a consisté à définir des méthodes de production de code séquentiel hautement optimisé [HPP88] [HRR91] [Ray91]. Les techniques développées, basées sur la génération d'automates, sont similaires à celles utilisée dans la compilation d'ESTEREL [Gon88]. Malheureusement, ces techniques ont un coût exponentiel et ne peuvent être utilisées que pour compiler des programmes de taille relativement réduite.

Dans le cadre de la génération d'une simulation de circuit, la problématique est différente. Les programmes peuvent être potentiellement de grande taille mais en revanche les contraintes sur la vitesse d'exécution sont moins importantes. La simulation n'est utilisée que pour tester qu'un circuit est fonctionnellement correct. Nous avons donc employé une méthode beaucoup plus classique mais également beaucoup moins coûteuse. Dans la pratique, la vitesse de la simulation n'a jamais été jusqu'à présent un problème.

11.1 Code en boucle simple

La méthode employée pour générer un code séquentiel est celle dite de "code en boucle simple". Il s'agit en fait d'un cas particulier de la génération d'un automate séquentiel, le résultat produit est un automate ne comportant qu'un seul état. Ce code est constitué d'une unique boucle, la i^{ieme} itération exécute le i^{ieme} cycle du programme :

- Elle lit les valeurs des entrées.
- Elle calcule la valeur courante de chaque variable.
- Elle affiche les valeurs des sorties.
- Elle calcule les valeurs des opérateurs `pre` (passage au cycle suivant).

Nous avons cherché à produire le code le plus simple possible : contrairement aux précédentes implémentations de ce générateur de code, nous n'avons absolument pas cherché à minimiser la place mémoire nécessaire pour exécuter ce code. Un emplacement mémoire est associé à chaque variable, à chaque opérateur `pre` et à chaque opérateur `current`.

Un programme LUSTRE expansé est fourni en entrée du compilateur. Deux tri topologiques sont effectués pour générer le code :

- Le premier permet de définir dans quel ordre évaluer les variables.
- Le second permet de définir dans quel ordre mettre à jour les valeurs des points mémoires.

11.2 Programmation modulaire

L'extension du langage rendant possible l'écriture de programmes de grande taille, pouvoir compiler séparément différentes parties d'un programme est devenu essentiel. Ce paragraphe donne un aperçu des techniques qui seront employées dans la future version du compilateur afin d'améliorer la compilation de LUSTRE sur machine séquentielle.

La génération de code telle qu'elle est réalisée actuellement a deux inconvénients :

- Il produit des programmes séquentiels énormes et mono-bloc, ce qui n'est pas très bien accepté par les compilateurs.
- La moindre modification dans un programme impose de le recompiler intégralement pour produire le code séquentiel correspondant.

Un exemple classique illustre le problème par la compilation séparée d'un noeud. Considérons l'équation `(X,Y) = Copy(A, not X)`, `Copy` étant le noeud défini par :

```
node Copy (a,b:bool) returns (x,y:bool);
let
   x = a;
   y = b;
end;
```

Si `Copy` est expansé, il n'y a aucun problème, le code généré pour l'équation est :

```
X := A;
Y := not X;
```

La question est quel code générer si `Copy` est compilé séparément? La difficulté vient du fait que `X` est à la fois produit par `Copy` et nécessaire à son calcul.

La solution proposée par [Ray88] pour résoudre ce problème, consiste à analyser le noeud à compiler séparément et le découper en blocs procéduraux, des noeuds que l'on sait compiler indépendamment de leur contexte d'appel. Chaque sortie d'un tel noeud dépend sans retard de toutes les entrées, ce qui interdit tout rebouclage, comme dans le cas du noeud `Copy`. Les valeurs de toutes les entrées peuvent donc toutes être calculées avant d'appeler le noeud. Dans notre exemple, le noeud se décompose ainsi :

```
node Copy (a,b:bool) returns (x,y:bool);
```

```
let
   x = Copy1(a);
   y = Copy1(a)
end;

node Copy1 (a:bool) returns (x:bool);
let
   x = a;
end;
```

Copy est expansé et Copy1 compilé séparément. Le code produit est donc :

```
X := Copy1(A);
Y := Copy1(not X);
```

Une procédure, dans un langage séquentiel classique, se contente de recevoir des données, faire des calculs et produire des résultats en retour. La mémoire dont elle peut avoir besoin pour effectuer ses calculs est rendue une fois ces derniers terminés. Un noeud LUSTRE en revanche est à la fois une unité de calcul et une unité de mémorisation. En effet, toute ses expressions peuvent être définies en fonction de n'importe quelle valeur passée ou présente de ses variables. Par conséquent, de la mémoire doit être associée à chaque instance de noeud pour conserver, d'un cycle sur l'autre, les calculs rémanents (typiquement les valeurs associées aux opérateurs **pre**), et permettre son évaluation. Un noeud compilé séparément possède donc un paramètre en plus de ceux décrits dans sa définition : la mémoire sur laquelle il doit faire ces calculs.

Un langage orienté objet, C++ [Str91] par exemple, peut être avantageusement utilisé à la place comme langage cible de la compilation de LUSTRE si la compilation séparée est utilisée. Il permet à la fois de :

- rendre implicite, grâce à la notion d'objet, la gestion de la mémoire associée à chaque instance d'un noeud compilé séparément.
- produire un code plus lisible.
- simplifier les interfaces avec les objet externes.

Un programme se décompose en un ensemble de modules, celui du programme principal, et ceux de blocs procéduraux. La pré-compilation d'un module produit un noeud expansé ne comportant plus que des appels à des noeuds ou des fonctions externes. Dans un second temps la génération de code séquentiel définit une classe d'objet pour ce noeud.

Une classe est également créée pour chaque type défini dans le programme. Seules les fonctions externes LUSTRE, qui sont des opérateurs sans mémoire, restent implémentées comme telles.

Un objet de la classe associée à un noeud se compose :

- Pour chaque variable, exception faite de celles de ses entrées et de ses sorties, d'un objet de la classe associée à son type.
- Pour chaque **pre**, également d'un objet de la classe associée à son type.

- Pour chaque appel de noeud, d'une référence à un objet de la classe associée au noeud appelé.

  Il s'agit d'une référence à un objet et non un objet proprement dit pour traiter le cas hypothétique d'un bloc procédural récursif. Par exemple, le noeud **pow** qui élève un entier **a** à la puissance **n** pourrait, dans une version future de la compilation séparée, être considéré comme un bloc procédural.

```
node pow (static n:int; a:int) returns (f:int);
let
f = with n = 0 then if a = 0 then 0 else 1
 else a* pow(n-1,a);
end;
```

- Trois fonctions :
  - de création. Les objets qui le composent ou ceux auxquels il fait référence sont également créés.
  - de calcul, en fonction de celle de ses entrées, des valeurs courantes des sorties de l'instance de noeud correspondante
  - de mise à jour, également en fonction des valeurs de ses entrées, des valeurs des mémoires associées aux **pre** de l'instance de noeud correspondante.

Un type externe est aussi implémenté par une classe :

- Les opérateurs de comparaison et d'affectation sont redéfinis
- Les fonctions de lecture et d'écriture des éléments de ce type sont également fournies.

Lorsque le noeud principal est construit, un programme principal est construit. Il définit les objets suivants :

- Un membre de la classe associée au noeud principal.
- Pour chaque variable d'entrée ou de sortie du noeud principal, un membre de la classe associée à son type.

Le code exécuté est très similaire à celui décrit dans la section précédente, il se compose toujours d'une boucle exécutant, à la i^{ieme} itération, le i^{ieme} cycle du programme. Un itération consiste à appeler successivement :

- la fonction de lecture de chaque objet associé à une entrée.
- la fonction de calcul des valeurs de l'objet associé au noeud principal. Elle invoque récursivement celles des objets associés aux différentes instances des noeud compilés séparément.
- la fonction d'écriture de chaque objet associé à une sortie.

- la fonction de mise des valeurs des `pre` de l'objet associé au noeud principal. Elle invoque également récursivement celles de objets associés aux différentes instances des noeud compilés séparément.

Remarque : Avec cette implémentation, un autre type de noeud peut être compilé séparément, il s'agit de ceux dont chacune des sorties ne dépend d'aucune entrée. Les valeurs des entrées ne servent dans ce cas qu'à mettre à jour les valeurs des mémoires associées aux opérateurs `pre`.

Chapitre 12

Interface Graphique

Un ensemble de schémas constitue, aussi bien en automatique qu'en conception de circuits, un moyen très employé pour décrire des applications. Les avantages de cette représentation, au moins pour de petits circuits, sont incontestables par rapport à une quelconque représentation textuelle. Un schéma est beaucoup plus explicite et universel que n'importe quelle syntaxe textuelle, il est possible de se faire beaucoup plus rapidement une idée globale du comportement d'un circuit. C'est le moyen que l'on utilise le plus naturellement, bien avant d'écrire des équations.

Les inconvénients sont également incontestables, le dessin d'un schéma est infiniment plus lent que l'écriture d'un programme, il est également moins compact. En fait l'intérêt d'un schéma, devient limité dès qu'il ne peut plus tenir sur une page ou un écran.

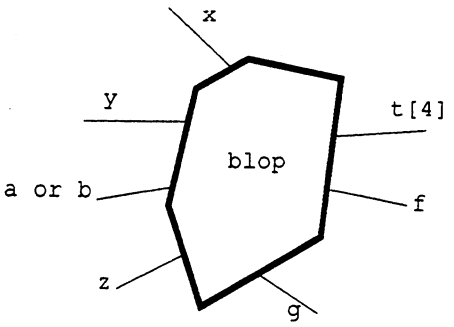
L'unique conclusion que l'on peut tirer est que, suivant les cas l'une ou l'autre représentation est plus adaptée. C'est la raison pour laquelle il a semblé intéressant en LUSTRE de pouvoir décrire n'importe quel noeud ou programme indifféremment sous forme graphique ou textuelle et de pouvoir aisément passer de l'un à l'autre. Ce travail a été grandement simplifié par le fait que tout programme puisse être vu comme un réseau d'opérateurs.

12.1 Interprétation de schémas

La passerelle dont nous allons parler en premier est celle qui transforme un ensemble de schémas en un programme textuel correspondant. Les schémas, produits grâce à un éditeur de dessins courant, sont fournis en entrée sous la forme d'un ensemble de lignes, de polygones, de textes et autres motifs élémentaires.

LUSTRE a été doté d'une syntaxe graphique, pour permettre l'analyse du programme et permettre d'en déduire le réseau d'opérateur.

- La définition d'un noeud est constituée d'un ensemble de motifs placés à l'intérieur d'une boite dessinée en pointillé. Le nom du noeud est un texte, se trouvant également à l'intérieur, commençant par le mot clé `node`. Un programme est défini par un ensemble de ces boites.
- Un fil définit une connexion. Deux fils dont l'extrémité de l'un touche l'autre désigne le même connexion.




```
(t[4], f, g) = blop(x, y, a or b, z)
```

Figure 12.1 *Représentation d'un opérateur*

- Un identificateur est un texte proche d'un fil. Deux fils associés au même nom sont identiques. Une expression peut également être attachée à un fil.

- Un contour fermé quelconque, dessiné en traits pleins comme celui représenté Figure 12.1 , désigne un opérateur dont le nom est le texte se trouvant à l'intérieur. Il peut s'agir aussi bien d'un opérateur prédéfini que d'un appel à un noeud, défini textuellement ou par un autre schéma. Les entrées/sorties d'un opérateur sont les fils dont une extrémité touche le contour. La distinction entre les entrées et les sorties est faite suivant le coté de l'opérateur ; le flux de donnée va de la gauche vers la droite. L'entrée d'un opérateur est donc un fil connecté à sa gauche, une sortie un fil connecté à sa droite.

  L'ordre des entrées (resp des sorties) est défini en fonction des coordonnées du point de contact entre le fil de chacune et le contour. Le classement s'effectue en fonction de leur ordonnée puis de leur abscisse. Une entrée (resp. sortie) est placée avant une autre dans la liste des paramètres si son point de contact avec le contour est plus haut ou alors à la même hauteur et plus à droite. Cet ordonnancement a été défini pour représenter le flot de données se propageant principalement horizontalement et de la gauche vers la droite.

  Le nom d'une entrée est un identificateur attaché à un fil rentrant par la gauche de la boite du noeud et une sortie, un identificateur attaché à un fil en sortant par la droite. Les entrées (resp des sorties) d'un noeud sont ordonnancées de la même façon que celles d'un opérateur.

  La Figure 12.2 décrit deux moyens de représenter un rebouclage.

  La syntaxe qui vient d'être donnée, ne permet pas en revanche de représenter un rebouclage comme représenté Figure 12.3 , en raison du sens imposé par les données, alors que celui-ci est relativement naturel.

  Il semble judicieux dans la future version du compilateur de modifier légèrement la syntaxe, afin d'assouplir les règles de dessin, en supprimant les contraintes sur le sens des flots de données. Les entrées/sorties d'un opérateur sont les fils dont une extrémité touche le contour. Il s'agit d'un entrée si l'extrémité est fléchée, d'une sortie dans le cas contraire. Le rebouclage Figure 12.3 est dessiné en utilisant cette nouvelle syntaxe.

  Cette modification des règles de dessins concerne également les paramètres du noeud. Le

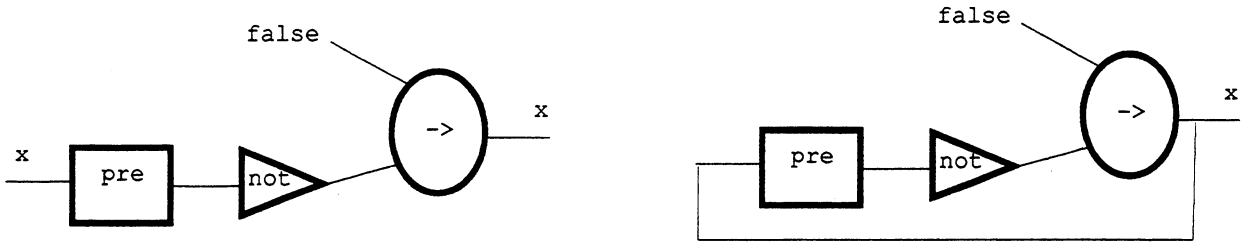
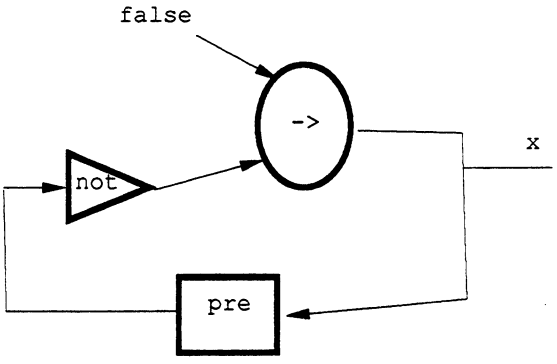


Figure 12.2 *Rebouclage d'un opérateur* **pre**




```
x = false -> not pre x
```

Figure 12.3 *Rebouclage d'un opérateur* **pre** *avec la bouvelle syntaxe*

nom d'une entrée est un identificateur attaché à une flèche rentrant dans la boite du noeud et une sortie, un identificateur attaché à une flèche en sortant.

- Un texte qui n'est attaché à aucun fil ou boite est considéré comme une partie, décrite sous forme textuelle, du programme
- Le type d'une variable suit au moins une des instance de l'identificateur. Dans la version actuelle, l'en-tête du noeud est ecrit.

Les schémas Figure 12.4 une fois interprétés produisent le programme suivant décrivant un compteur trois bit avec reset.

```
node ResetCounter1 (reset, count: bool) returns (sum, carry: bool);
var old:bool;
let
    sum =   if reset then false else old xor count;
    carry =   if reset then false else old and count;
    old = false->pre sum;
tel;

node ResetCounter3 (reset, count: bool) returns (sum:bool^3; carry: bool);
var old:bool^3; C:bool^3;
let
    (sum,C) = ResetCounter1 (reset^3, count^3)
    carry = C[2];
tel;
```

12.2 Production de schémas

La traduction d'un programme LUSTRE textuel en un ensemble de schémas a été avant tout développée pour accélérer le dessin de noeuds. Le but est de produire les schémas dans un format compréhensible par un éditeur et ne laisser au concepteur comme travail que la dernière phase de mise en forme. L'autre pré-requis était de produire des schémas compatibles avec la syntaxe graphique qui vient d'être décrite, de façon a pouvoir les interpréter après modification.

Réaliser un processeur produisant une représentation graphique de programmes LUSTRE pose, dans l'absolu des problèmes similaires à ceux rencontrés, en CAO, avec les outils de dessin de circuits. Il s'agit :

- du placement des opérateurs
- du tracé et du croisement des connexions.

Heureusement les buts fixés ont permis de faire des choix qui ont grandement réduit les problèmes.

La lisibilité d'un schéma se réduit très rapidement dès que des fils se croisent. Pour éviter cela nous utilisons la ressource de couper un fil nommé que nous donne la syntaxe graphique de

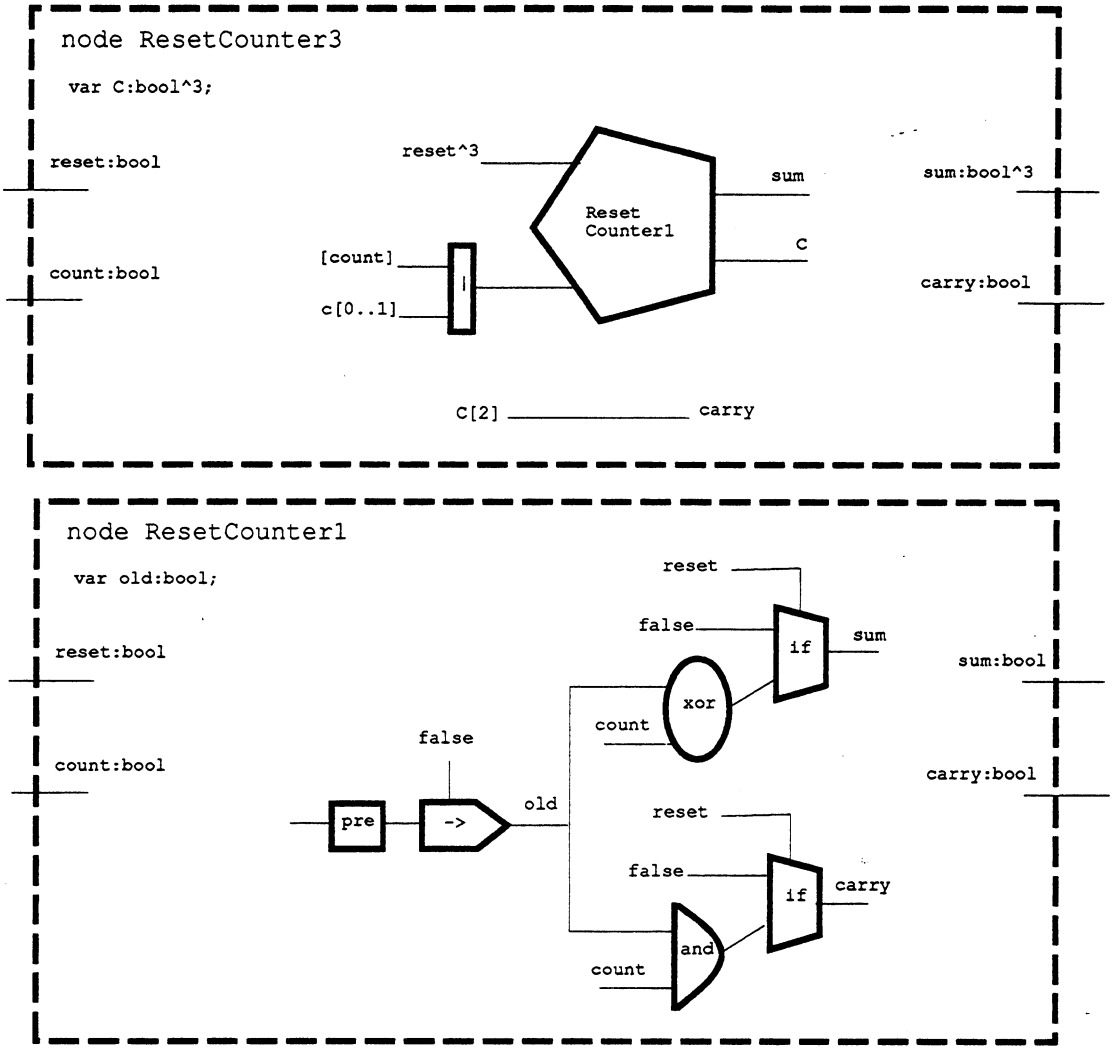


Figure 12.4 *Schéma d'un compteur de trois bits avec reset*

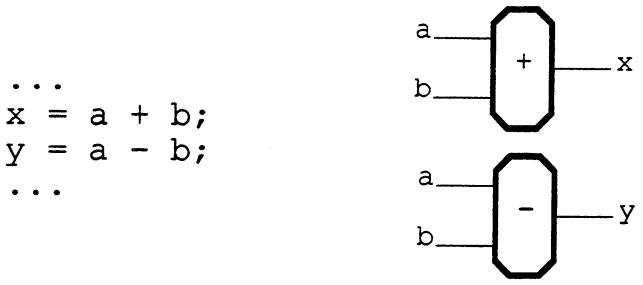


Figure 12.5 *Dessin du réseau*

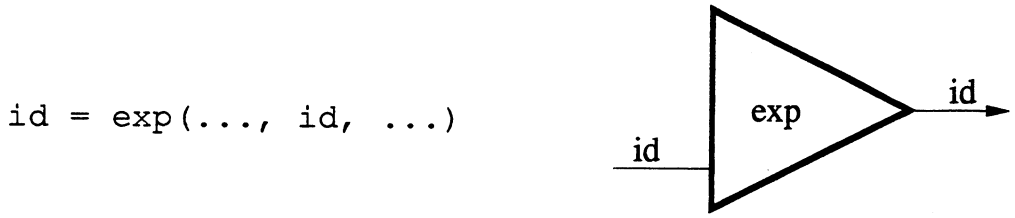


Figure 12.6 *suppression des rebouclages*

LUSTRE. Aucune heuristique n'est utilisée pour essayer de minimiser le nombre de ces coupures, elles sont faites systématiquement. Par conséquent, la représentation graphique d'un noeud LUSTRE sera un ensemble de réseaux disjoints, un par équation, comme représenté Figure 12.5 .

Ces réseaux sont en fait des arbres. En effet, un programme LUSTRE vérifie, par construction, la propriété suivante : dans tout circuit du réseau d'opérateurs, il existe au moins un fil nommé. il n'y a donc pas de rebouclage, un cycle étant coupé au niveau du fil nommé comme représenté sur la Figure 12.6 .

Egalement par construction, un fil ayant plusieurs consommateurs est nommé. Tous les fils dessinés auront donc au plus une source et une destination.

Puisqu'il n'y a plus de rebouclage, toutes les données circulent dans le même sens. Par convention, nous décidons que ce sens est horizontalement et de la gauche vers la droite.

Afin d'augmenter encore la lisibilité, une partie du programme source est laissée sous forme textuelle. Il s'agit des sélections et des expressions définissant leurs bornes.

Le problème des connexions étant maintenant résolu, il reste encore à faire le placement. A chaque composant d'un programme LUSTRE, nous associons une boite dans laquelle il va être dessiné. L'organisation du schéma est définie en fonction de ces boites.

- Les boites des équations sont placées les unes sur les autres.
- La boite d'une expression `e = op(e1, e2, ... , en)` contient (§ Figure 12.7) les boites

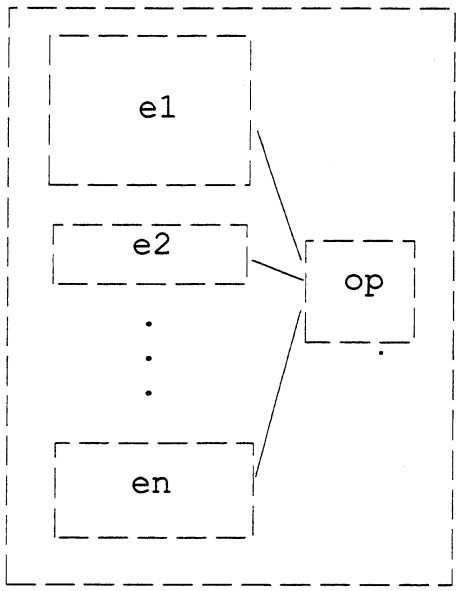


Figure 12.7 *Boite d'une expression*

superposées de ses sous expressions `e1`, `e2`, ... , `en` et, à coté au centre, la boite de `op`.

Si l'opérateur possède plusieurs sorties, la boite de l'expression `e` n'est pas incluse dans l'expression englobante mais placée à cotée comme représenté Figure 12.8 . Les boites des expressions consommatrices d'une sortie de `e` sont placées les unes sur les autres et centré verticalement avec celle de `e`.

Il est à remarquer que ce placement pourrait être encore utilisé si certains fils ayant plusieurs destinations n'étaient pas découpés, le placement d'un boite ayant plusieurs sorties étant sensiblement identique ùne boite ayant plusieurs un sortie mais plusieurs consommateurs. Toute la difficulté serait de trouver un bon critère (simple également pour ne pas retomber dans les problèmes que nous avons cherchés à éviter) qui permette de déterminer quels fils il est possible conserver intact sans compliquer le dessin d'un schéma.

- La boite d'un identificateur est la boite englobante du texte correspondant

Le placement est en réalité un peu plus complexe car il est nécessaire de laisser des canaux pour laisser passer les fils.

- Dans la boite de `e`, celle de `op`, est espacée de celles des autres d'un nombre d'entrées égal à la moitié de son nombre d'entrées. (les entrées i et n -i utilisent le même canal. Puisque `op` est centré il n'y a pas de recouvrement de fils) Les fils reliant la sortie d'une sous-expression `ei` à `op` sont composés de trois segments :
  - le premier est horizontal et relie la sortie de `ei` au canal qui lui est affecté
  - Le second est vertical.

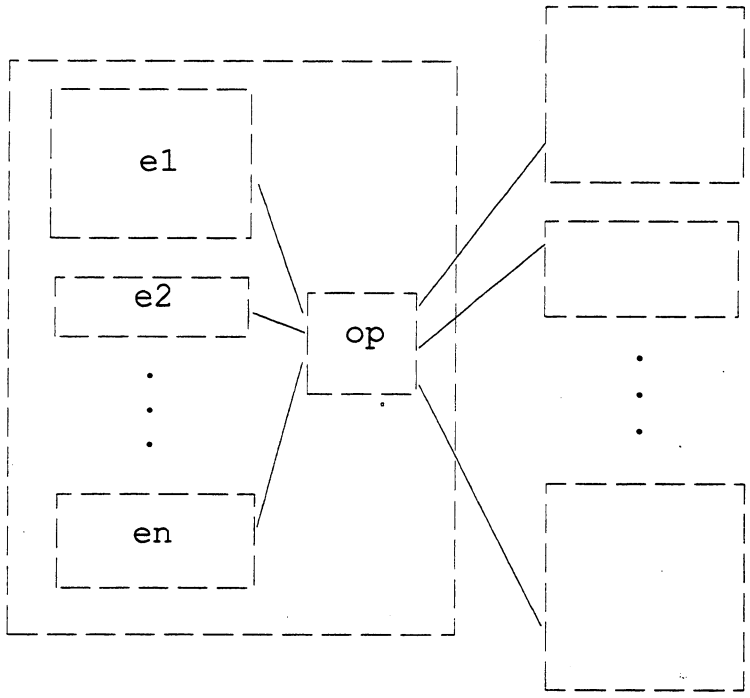


Figure 12.8 *Boite d'une expression ayant plusieurs sorties*

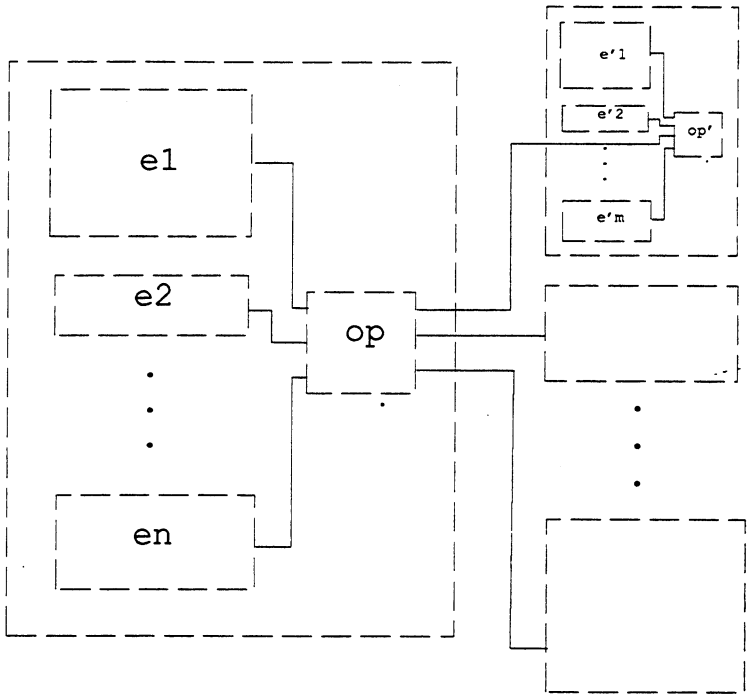


Figure 12.9 *Tracé des connexions*

- Le troisième est horizontal il relie le canal à l'entrée `i` de `op`.

Le même type d'espacement est réalisé en sortie de `e`.

- si `e` a plusieurs sorties, la hauteur de la boite de chaque expression consommatrice est calculée de façon à laisser un canal d'accès au fil comme représenté Figure 12.9 . Un fil de sortie de `op` est composé de cinq segments au lieu de trois, il passe par trois canaux (deux verticaux, un horizontal) au lieu de un.

Voici le schéma du compteur trois bits dont le programme textuel est donné dans la section précédente.

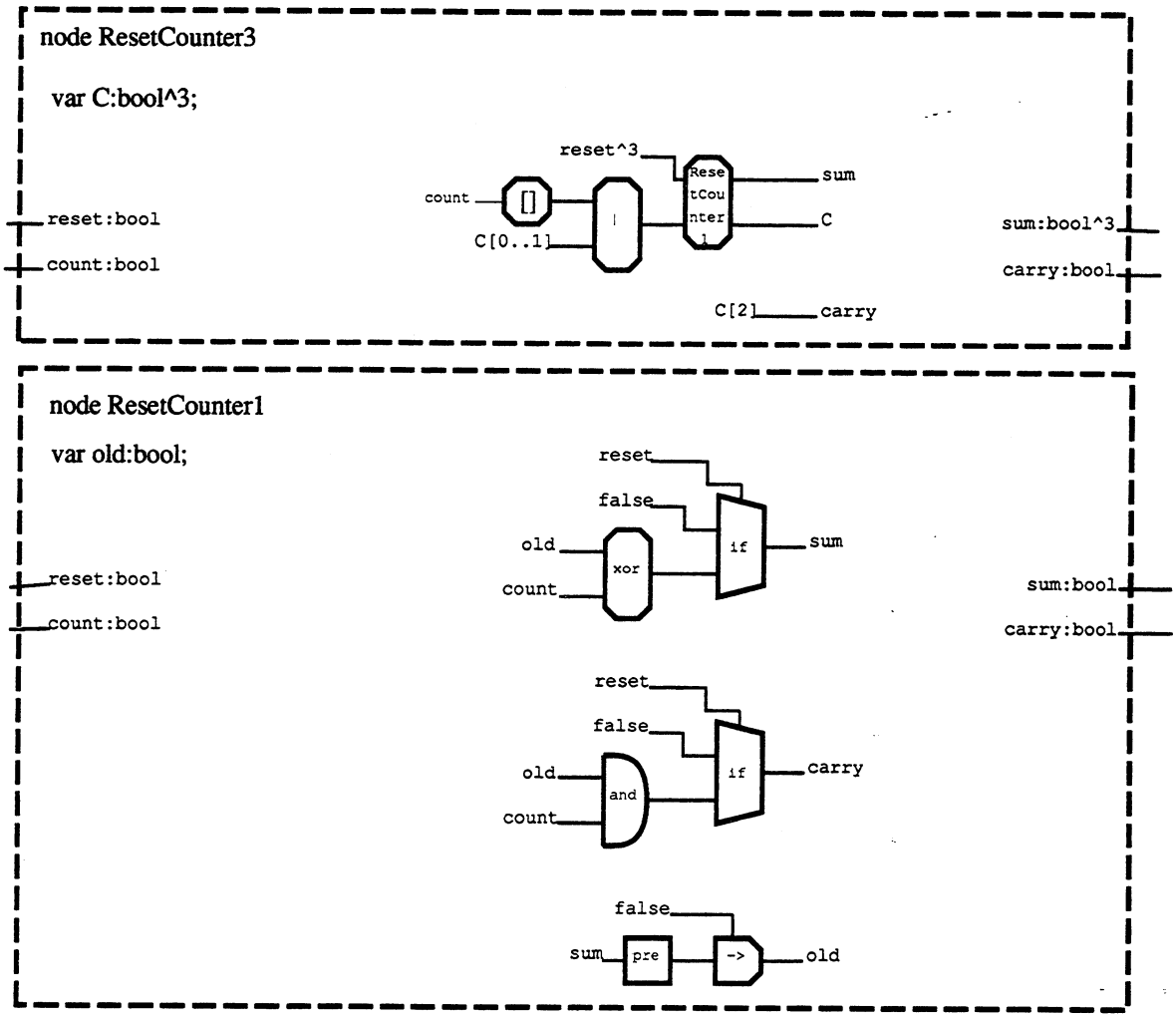


Figure 12.10 *Schémas produits du compteur de trois bits*

Conclusion

La version actuelle de POLLUX a été écrite en C++ et compilée sur SUN 3, SUN-Sparc, VAX et Dec-Mips. Il a permis de tester et valider les extensions apportées au langage LUSTRE. A ce jour, il a été utilisé à prl pour réaliser six applications hautes performances opérationnelles sur la PAM. POLLUX est également utilisé en entrée des autres outils LUSTRE.

Différentes suites à ce travail sont envisageables. L'extension du langage tout d'abord entraîné de nouvelles études sur les techniques à mettre en oeuvres pour compiler efficacement un programme sur machine séquentielle :

- L'amélioration de la génération de code séquentiel est actuellement étudiée par [Pro92]. Son objectif est limiter au strict minimum l'éclatement des tableaux lors de la compilation. Les principaux résultats attendus sont une compilation accélérée et un code produit beaucoup plus compact. Il s'agit par exemple pour les deux équations suivantes :

  ```
  X[0] = 0;
  X[1 .. n] = X[0 .. n-1] + 1;
  ```

  de produire, plutôt que n+1 affectations distinctes le code séquentiel :

  ```
  X[0] = 0;
  pour i = 0 -> n
           X[i] = X[i-1] +1;
  finpour
  ```

- Avec la taille les programmes maintenant décrits, la compilation modulaire de programme LUSTRE devient indispensable pour éviter de devoir recompiler l'ensemble d'un application chaque fois que la moindre modification locale est réalisée. Le problème est qu'il est impossible de générer un code séquentiel pour un noeud LUSTRE pris indépendamment de son contexte d'appel. La compilation séparée, évoquée § 11.2 qui est actuellement implantée par [Dub92] est une première réponse à ce problème.

Un autre genre d'extension consiste à réaliser le travail inverse de celui décrit dans ce document, Il s'agit d'utiliser la PAM comme machine cible sur laquelle sont exécutés les programmes LUSTRE. PAM pour exécuter n'importe quel programmes LUSTRE

Une implantation directe, comme celle employée pour programmer la PAM, permet d'obtenir des performances optimales, le parallélisme intrinsèque d'un programme étant totalement conservé. Mais, si cette solution est intéressante pour des programmes purement booléens, elle n'est pas

utilisable pour de gros programmes comportant des calculs entiers ou réels, car elle est trop coûteuse en surface. En effet un circuit comprendra par exemple autant d'additionneurs et de multiplieurs que le programme source contient d'additions et de multiplications.

Une autre solution extrême peut consister à développer sur la PAM un processeur spécialisé pour exécuter le code séquentiel produit par la compilation de LUSTRE. Mais le gain de vitesse par rapport à un processeur classique est alors limité. Une solution souhaitable serait donc un compromis entre celle purement parallèle et celle purement séquentielle, de façon à employer au mieux la PAM. Il s'agirait, lors de la compilation d'un programme, de générer à la fois un code séquentiel de simulation et la configuration programmant sur la PAM le processeur permettant d'exécuter efficacement ce code.

Une dernière solution enfin consisterait à utiliser la PAM uniquement comme co-processeur. La compilation d'un programme produirait un code séquentiel et l'accélérateur matériel associé.

Bibliographie

[Ash90] P. J. Ashenden. The VHDL cookbook, 1990. 1^{nd} Edition.

[ASU86] A. Aho, R. Sethi et J. Ullman. *Compilers : Principles, Techniques and Tools.* Addison-Wesley, 1986.

[AW85] E.A. Ashcroft et W.W. Wadge. *Lucid, the data-flow programming language.* Academic Press, 1985.

[BCP88] B. Buggiani, P. Caspi et D. Pilaud. Programming distributed automatic control systems: a language and compiler solution. Rapport Technique SPECTRE L4, LGI/IMAG, Grenoble, France, Juillet 1988.

[Ber86] J.L. Bergerand. *Lustre, un langage déclaratif pour le temps réel.* Thèse, Institut National Polytechnique, Grenoble, France, Janvier 1986.

[Ber91] G. Berry. A hardware implementation of pure ESTEREL. In *ACM Workshop on Formal Methods in VLSI Design*, Janvier 1991.

[BRAW87] R. K. Brayton, R. Rudell, Sangiovanni-Vincentelli A et A. Wang. Mis: Multiple level logic optimization system. In *IEEE Transactions on Computer-Aided Design of Integrated Circuits*, pages 1062–1081, Novembre 1987.

[BRV90] P. Bertin, D. Roncin et J. Vuillemin. Introduction to programmable active memories. In J. McCanny, J. McWhirter et E. Swartzlander, editors, *Systolic Array Processors*, pages 301–309. Prentice-Hall, 1990. Aussi disponible en tant que Rapport de Recherche DEC PRL 3.

[BRV92] P. Bertin, D. Roncin et J. Vuillemin. Programmable active memory : Performances assesments. In *FPGA Conference*, Février 1992.

[BS91] F. Boussinot et R. de Simone. The ESTEREL language. *Proceedings of the IEEE*, 79(9):1293–1304, Septembre 1991.

[Buo88] C Buors. Sémantique opérationnelle du langage Lustre. Rapport de DEA, Institut National Polytechnique, Grenoble, France, Juin 1988.

[CBM89] O. Coudert, C. Berthet et J. C. Madre. Verification of synchronous sequential machines based on symbolic execution. In *International Workshop on Automatic Verification Methods for Finite State Systems, LNCS 407.* Springer Verlag, 1989.

[Che65] N.V. Chernikova. Algorithm for finding a general formula for the non-negative solution of a system of linear inequalities. *USSR Computational Mathematics and Mathematical Physics*, 5(2):228–233, 1965.

[Che86] M.C. Chen. Transformations of parallel programs in crystal. In H.J. Kugler, editor, *IFIP*, pages 455–462, 1986.

[Che88] M.C. Chen. The generation of a class of multipliers: Synthesizing highly parallel algorithms in vlsi. *IEEE Transactions on Computers*, 37(3):329–338, 1988.

[Dub92] C. Dubois. Un pré-processeur de compilation séparée pour le langage Lustre. Mémoire cnam, LGI/IMAG, Grenoble, France, Juillet 1992.

[DVPY91] C. Dezan, H. Le Verge, P.Quinton et Y.Saouter. *The Alpha du Centaur Experiment*, pages 325–334. Elsevier, 1991.

[FC89] H. De Man F. Catthoor. Target architectures in the Cathedral synthesis systems: objectives and design experience. *IEEE International symposium on Circuits and Systems*, pages 1907–1910, 1989.

[FP88] F.Fernandez et P.Quinton. Extension of chernikova's algorithm for solving general mixed linear programming problems. Rapport Technique PI321, IRISA, Rennes, France, 1988.

[GBR82] J. P. Gray, I. Buchanan et P. S. Robertson. Designing gate arrays using a silicon compiler. In *19^{th} Design Automation Conference*, pages 377–383, 1982.

[GBR83] J. P. Gray, I. Buchanan et P. S. Robertson. Controlling vlsi complexity using a high-level language for design description. In *International Conference on Computer Design*, 1983.

[GH85] J. P. Gray et J. Hunter. Portability in silicon cae. In *22^{th} Design Automation Conference*, pages 597–601, 1985.

[GL85] S. M. German et K. J. Lieberherr. Zeus: A language for expressing algorithms in hardware. *IEEE computer*, pages 55–65, Février 85.

[Gon88] G. Gonthier. *Sémantiques et modèles d'exécution des langages réactifs synchrones; application à Esterel.* Thèse, Université d'Orsay, Paris, France, 1988.

[HCRP91] N. Halbwachs, P. Caspi, P. Raymond et D. Pilaud. The synchronous dataflow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, Septembre 1991.

[Hil87] P.N. Hilfinger. *Silage Reference manual.* IMEC, 1987.

[HL91] N. Halbwachs et F. Lagnier. Sémantique statique du langage Lustre. Rapport Technique SPECTRE L15, LGI/IMAG, Grenoble, France, Février 1991.

[HP86] N. Halbwachs et D. Pilaud. Use of a real time declarative language for systolic array design and simulation. In *International Workshop on Systolic Arrays*, Juillet 1986.

[HPP88] N. Halbwachs, D. Pilaud et J.A. Plaice. Generating efficient code from data-flow programs. Rapport Technique SPECTRE L8, LGI/IMAG, Grenoble, France, Juillet 1988.

[HRG+90] P. Hilfinger, J. Rabaey, D. Genin, C. Scheers et H. De Man. Dsp specification using the Silage language. *IEE International Conference on Acoustics, Speech and Signal Processing*, 1990.

[HRR91] N. Halbwachs, P. Raymond et C. Ratel. Generating efficient code from data-flow programs. In *Third International Symposium on Programming Language Implementation and Logic Programming*, Passau,, Août 1991.

[J.B78] J.Backus. Can programming be liberated from the von neumann style? a functionnal style and its algebra of programs. *Communications of the ACM*, 21(8):613–641, Août 1978.

[J.L88] J.L. *81/2 un modèle MSIMD pour la simulation massivemement parallèle.* Thèse, Université d'Orsay, Paris, France, 1988.

[KL80] H.T. Kung et C.E. Leiserson. Algorithms for vlsi processors arrays. In Addison-Wesley, editor, *Introduction to VLSI Systems*, chapter 8.3. C.A. Mead and L.A. Conway, Reading, Mass, USA, 1980.

[Kun82] H.T. Kung. Why systolic architectures? *Computer*, pages 37–46, Janvier 1982.

[Leg79] B. Legrand. *Apprendre et Appliquer le Langage APL.* Masson, 1979.

[LGLL91] P. Le Guernic, T. Gautier, M. Le Borgne et C. Le Maire. Programming real-time applications with signal. *Proceedings of the IEEE*, (9), Septembre 1991.

[LS81] C.E. Leiserson et J.B. Saxe. Optimizing synchronous circuitry. *Foundations of Computer Science*, pages 23–36, Octobre 1981.

[LS86] C.E. Leiserson et J.B. Saxe. Retiming synchronous circuitry. Rapport technique, DEC SRC, 1986.

[Mar90] F. Maraninchi. *Argos, un langage graphique pour la conception, la description et la validation des systèmes réactifs.* Thèse, Université Joseph Fourrier, Grenoble, France, Janvier 1990.

[Mau89] C. Mauras. *Alpha : un langage équationnel pour la conception d'architectures parallèles synchrones.* Thèse, Université de Rennes 1, France, Décembre 1989.

[Mon85] C. Mongenet. *Une méthode de conception d'algorithmes systoliques, résultats théoriques et réalisation.* Thèse, Institut National Polytechnique, Lorraine, France, Mai 1985.

[Nik85] R.S. Nikhil. Practical polymorphism. In *Funtional Programming Languages and Computer Architecture*, pages 319–333, Septembre 1985.

[Pa85] T. Piloty et al. The Conlan project, concepts, implementations and applications. *IEEE computer*, pages 81–92, Février 85.

[Pay91] E. Payan. *Etude d'une architecture cellulaire programmable : définition fonctionnelle et méthodologie de programmation.* Thèse, Institut National Polytechnique, Grenoble, France, 1991.

[Pla88] J.A. Plaice. *Sémantique et compilation de Lustre, un langage à flots de données temps réel.* Thèse, Institut National Polytechnique, Grenoble, France, Mai 1988.

[Plo81] G. D. Plotkin. A structural approach to operational semantics. Lecture notes, Aarhus University, 1981.

[P.Q84] P.Quinton. Automatic synthesis of systolic arrays from uniform reccurent equations. In *11^{th} Annual Symposium on Computer Architecture*, pages 208–214, 1984.

[Pro92] Y.E. Proy. Compilation des tableasux Lustre. Rapport de DEA, Institut National Polytechnique, Grenoble, France, Juin 1992.

[PS87] J. A. Plaice et J-B. Saint. The Lustre-Esterel portable format. Manuel de Référence, 1987.

[PSE85] D. Patel, M. Schlag et M. Ercegovac. ν-FP, an environment for multi-level specification, analysis and synthesis of hardware algorithms. In P. Jouannaud, editor, *Functional Programming and its applications*, pages 238–255, 1985.

[Rat92] C. Ratel. *Définition et reálisation d'un outil de vérification formelle de programmes Lustre : Le système Lesar.* Thèse, Université Joseph Fourrier, Grenoble, France, Juillet 1992.

[Ray88] P. Raymond. Compilation séparée d'un langage déclaratif synchrone. Rapport Technique SPECTRE L5, LGI/IMAG, Grenoble, France, Juin 1988.

[Ray91] P. Raymond. *Compilation efficace d'un langage déclaratif synchrone : le générateur de code Lustre-V3.* Thèse, Institut National Polytechnique, Grenoble, France, Novembre 1991.

[Roc89] F. Rocheteau. Programmation d'un circuit massivement parallèle à l'aide d'un langage déclaratif synchrone. Rapport Technique SPECTRE L10, LGI/IMAG, Grenoble, France, Juin 1989.

[She84] M. Sheeran. Designing regular array architectures using higher order functions. In *ACM Symposium on Lisp and Functional Programming*, pages 104–112, 1984.

[She85] M. Sheeran. μ-FP, a language for VLSI design. In P. Jouannaud, editor, *Functional Programming and its applications*, pages 220–235, 1985.

[SLM+85] M. Shadad, R. Lipsett, E. Marschner, K. Sheenan, H. Cohen, R. Waxman et D. Ackley. VHSIC hardware description language. *Computer*, pages 94–103, Février 1985.

[STB91] H. Savoj, H. Touati et R. K. Brayton. The use of image computation techniques in exctracting local don't cares and network optimization. In *International Conference on Computer Aided Design (ICCAD)*, Novembre 1991.

[Str91] B. Stroustrup. *The C++ Programming Language.* Addison-Wesley, 1991. 2^{nd} Edition.

[Uni] Université de Californie. *Berkeley Logic Interchange Format (Blif).*

Annexe A

Manuel de référence du langage Lustre V4

A.1 Considérations lexicales

A.1.1 Jeu de caractères

Un programme LUSTRE est fourni en entrée du compilateur POLLUX sous la forme d'une séquence de caractères. Celle-ci est, lors de la phase initiale d'analyse lexicale, est décomposée en une succession de groupes de caractères, chacun associé à un symbole terminal de la syntaxe, aussi appelé lexème.

Par exemple, l'addition des deux entiers 12 et 23 est décrite dans un programme LUSTRE par la séquence de caractères `12 + 23`. Elle se décompose en trois groupes `12`, `+` et `23` associés respectivement aux lexèmes *constante entière*, *opérateur addition* et *constante entière*.

Un programme LUSTRE est principalement décrit à l'aide trois catégories de caractères

- alphanumériques `0 .. 9`, `A .. Z`, `a .. z`
- `" , #, (, ), *, +, −, . /, :, ;, <, =, >, [, ], ^, _, {, |, }`
- espace, retour de chariot, tabulation

D'autres caractères peuvent être utilisés mais uniquement:

- à l'intérieur d'un commentaire ou d'un pragma A.1.5
- dans le nom du fichier d'une directive d'inclusion textuelle A.1.6

Les groupes de caractères associés aux lexèmes sont formés à l'aide de ceux des deux premières catégories. Les caractères spéciaux espace, retour de chariot et tabulation quant à eux, sont considérés comme des séparateurs. Il peut y avoir entre deux groupes de caractères un nombre quelconque de séparateurs, éventuellement zéro si aucune ambiguïté n'est possible dans la détermination des lexèmes.

Ainsi, si l'on reprend l'exemple précédent, l'addition de 12 et 13 peut aussi s'écrire, entre autres façons: `12 + 23`, `12+23` ou `12 +`
`23`

A.1.2 Les identificateurs

Les identificateurs sont formés d'une lettre ou d'un caractère souligné suivi éventuellement de n'importe quel nombre de caractères alphanumériques et de caractères soulignés.

- `blurg`, `_Blurg`, `_12`, `bl_urg` ou `BLURG12` sont des identificateurs
- `12blurg` ou `bl#urg` n'en sont pas

Les majuscules et les minuscules sont des caractères différents. Ainsi

- `blurg`, `Blurg` et `BLURG` sont trois identificateurs différents

A.1.3 Mots-clés

Les mots clés suivants sont réservés et ne peuvent pas être utilisés comme identificateurs:

`and`	`assert`	`const`	`current`	`dep`
`div`	`else`	`end`	`function`	`if`
`include`	`let`	`length`	`mod`	`needs`
`node`	`not`	`or`	`pre`	`returns`
`static`	`step`	`then`	`type`	`var`
`when`	`with`	`xor`		

Pour rester compatible, avec les versions antérieures de LUSTRE `tel` est aussi un mot clé.

Tout comme pour les identificateurs, les majuscules et les minuscules sont des caractères différents. Ainsi si `node` est un mot clé alors que `Node` ou `NODE` sont des identificateurs.

Certains identificateurs sont également prédéfinis :

- les types :

`bool` `int` `real` char

- les constantes :

`false` `true` `nil`

- les fonctions :

`int` `real`

A.1.4 Constantes

Trois types de constantes sont prédéfinis :

- Les constantes booléennes *vrai* et *faux* sont représentées respectivement par les chaînes de caractères:
  - ◇ `true` ou `1`
  - ◇ `false` ou `0`
- Une constante entière est une suite de chiffres. Par exemple, 12 et 0012 sont deux représentations de la constante entière 12. La valeur d'une constante entière est positive ou nulle. Ainsi, -12 est interprété comme une expression formée de la constante 12 précédée de l'opérateur moins unaire.
- Une constante réelle est un entier suivi au moins d'une partie décimale ou d'un exposant. La partie décimale est un point suivi d'un entier, l'exposant est un caractère 'e' ou 'E' suivi éventuellement d'un signe puis d'un entier. Par exemple,
  - ◇ 0.12, 12.12, 12e12 ou 12.12E+12 sont des constantes réelles
  - ◇ 12 ou .12 n'en sont pas

A.1.5 Commentaires et Pragmas

Les commentaires se présentent sous deux formes:

- ceux qui commencent par la séquence de caractères '−−' et qui se terminent à la fin de la ligne.
- ceux qui commencent par la séquence de caractères '`(*`' et se terminent par la séquence '`*)`'.

Les pragmas sont des chaînes de caractères délimitées par '{' en début et '}' en fin. Le début de chaque chaîne est un identificateur terminé par deux points, il indique le type du pragma, en fonction duquel chaque outil choisi de l'interpréter ou de l'ignorer.

Les commentaires, tout comme les pragmas, peuvent ˆetre placés à n'importe quel endroit du programme, entre deux lexèmes quelconques.

A.1.6 Décomposition d'un programme en plusieurs fichiers

Il est possible d'écrire un programme LUSTRE dans plusieurs fichiers grâce à un mécanisme d'inclusion textuelle. La directive: `include "[`*path/*`]`*file*`"` indique au compilateur qu'il doit aussi lire les données contenues dans le fichier *file* et les inclure à l'endroit où est placée la directive.

Cette directive peut se trouver dans le fichier principal mais aussi dans les fichiers inclus eux même.

L'argument optionnel *path* permet d'indiquer le chemin, absolu ou relatif, pour accéder au fichier *file* lorsqu'il se trouve pas dans le répertoire courant (celui du fichier contenant la directive). Enfin, si plusieurs directives concernent un même fichier, seule la première est prise en compte.

A.2 Méta-syntaxe

La syntaxe de LUSTRE est donnée en employant un ensemble de règles de la forme *left::= right* où *left* est un non terminal et *right* sa définition. L'écriture des règles respecte les conventions suivantes:

- Un choix est représenté par une barre verticale: ‖
- Une partie de règle optionnelle est placée entre crochets: [ ]
- Une partie de règle ayant un nombre d'occurrences positif ou nulle est placée entre accolades: { }
- les non terminaux sont écrits en italique: *ident_decl*
- les terminaux sont écrits en gras: **bool**

Voici deux exemples de règles:

A.2.0.0.a liste d'identificateurs :

ident_list ::= **ident** { , **ident** }

A.2.0.0.b liste de déclarations de variables :

ident_decl_list ::= *ident_decl* { ; *ident_decl* }
ident_decl ::= [**const** ‖ **static**] *ident_list* : *type*

A.3 Structure d'un programme

Un programme est une liste de déclarations séparées par des points virgules, dont l'ordre est sans importance.

program ::= ε ‖ *declaration* { *declaration* }

A.4 Déclarations

Les déclarations permettent d'associer des identificateurs à des constantes, des types, des noeuds ou des fonctions. Il est interdit d'utiliser un même identificateur dans deux déclarations différentes.

declaration ::= *constant_declaration* ‖ *static_declaration* ‖ *type_declaration* ‖
node_declaration ‖ *function_declaration*

A.4.1 Déclaration de constantes

Une liste de déclarations de constantes, chacune terminée par un point virgule, est précédée du mot réservé `const`.

constant_declaration ::= `const` *constant_decl_list*
constant_decl_list ::= *constant_decl* ; {*constant_decl* ; }

Peuvent être déclarées:

- des constantes locales dont la valeur et le type se déduisent de son expression en partie droite.

  constant_decl ::= *ident_list* = *expression*

- des constantes externes dont on indique juste le type.

  constant_decl ::= *ident_list* : *type*

Les constantes booléennes de LUSTRE `true` et `false` ne peuvent pas être redéfinies.

A.4.2 Déclaration de constantes statiques

Une liste de déclarations de constantes statiques, chacune terminée par un point virgule, est précédée du mot réservé `static`.

static_declaration ::= `static` *static_decl_list*
static_decl_list ::= *static_decl* ; {*static_decl* ; }

Ne peuvent être déclarées que des constantes locales dont la valeur et le type se déduisent de son expression en partie droite.

static_decl ::= *ident_list* = *expression*

A.4.3 Déclaration de types

Une liste de déclarations de types, chacune terminée par un point virgule, est précédée du mot réservé `type`.

type_declaration ::= `type` *type_decl_list*
type_decl_list ::= *type_decl* ; {*type_decl* ; }
type_decl ::= *ident_list* | *ident* =[`const` | `static`]*type*
type ::= *any* | `ident` | *type* ^ *expression_list* | [*labelled_type_list*]
labelled_type_list ::= `ident` : *type* { , `ident` : *type* }
any ::= * | * *any*

Peuvent être déclarés:

- des types externes dont on fournit juste le nom.
- des types locaux, dérivés des types prédéfinis ou déjà déclarés à partir des deux constructeurs de structures et de tableaux.

Les types prédéfinis `bool`, `int` et `real` ne peuvent pas être redéfinis.

A.4.4 Déclaration de noeud

La déclaration d'un noeud débute par le mot clé `node` et se termine par le mot clé `end`. A l'intérieur sont déclarés :

- Son interface.
- D'éventuelles déclarations de variables locales.
- Un système d'équation.

node_declaration ::= `node` *header* ; *local_decl_list* [*body*]`end;`
header ::= `ident` (*param_list*) `returns` (*param_list*)
[`dep` *dependancy_list*]

L'entête d'un noeud se compose :

- D'un identificateur qui est le nom associé au noeud.
- De la liste, entre parenthèses, de ses entrées. Chaque déclaration, séparée de la suivante par un point virgule, spécifie le type, et éventuellement l'horloge (c'est à dire une entrée booléenne du noeud précédemment déclarée) d'une partie des entrées. Si l'horloge est omise, les entrées concernées sont supposées être sur l'horloge de base.
  Le fait qu'une entrée soit une constante (resp. statique) est indiqué en précédant sa déclaration du mot réservé `const` (resp. `static`).

param_list ::= ε | `()` `when ident` | *param* { ; *param* } ;
param ::= *ident_decl_list* |
(*ident_decl_list*) `when` *ident*

- De la liste entre parenthèse de ses sorties après le mot réservé `returns`. Les déclarations de sorties ont la même syntaxe que celles des entrées.
- De la listes de dépendances instantanées entre entrées et sorties ou sorties et entrées

dependancy_list ::= *dependancy* {; *dependancy* };
dependancy ::= *left* `needs` *left*

liste de déclarations de variables locales, chacune terminée par un point virgule, est précédée du mot réservé `var`. Ces déclarations sont identiques aux déclarations des paramètres de sortie.

local_decl_list ::= `var` {| *ident_decl_list* ; |}

A.4.5 Déclaration de fonction

Une déclaration de fonction est constituée uniquement d'un entête, similaire à celui d'un noeud, à quelques exceptions près:

- Il débute par le mot réservé `function`
- Tous les paramètres d'entrées et de sorties ne peuvent avoir d'autre horloge que `true`, l'horloge de base de la fonction, qui définit à quels instants elle est appelée.

function_declaration ::= `function ident (` *param_list* `)`
`returns (` *param_list* `) ;`

A.5 Equations

Un système d'équations, chacune terminée par un point virgule, est précédée par le mot clé `let`.

body ::= `let` *equation_list*
equation_list ::= *equation* ; {| *equation* ; |}
equation ::= *behavioral_eq* ‖ *assertion*

A.5.1 Equation

Le comportement au cours du temps de chaque variable locale ou de sortie est défini par une équation.

behavioral_eq ::= *left* = *expression*

L'expression en partie gauche d'une équation est limitée à un identificateur, a une expression de sélection ou une liste de telle expression.

left ::= ( ) ‖ *structured_ident* ‖ (*structured_ident_list*)
structured_ident ::= `ident` ‖ `ident [` *select_list* `]` ‖ `ident.ident` ‖
`[` *structured_ident_list* `]`
structured_ident_list ::= *structured_ident* {| , *structured_ident_list* |}

A.5.2 Assertion

Les assertions permettent de compléter la définition d'un programme donnée par ses équations en indiquant certaines relations entre les variables qui doivent toujours être vérifiées.

assertion ::= `assert` *expression*

A.6 Expressions

expression ::= `constant` | `ident` |
unary_op expression | *expression binary_op expression* |
ternary_op | *prefixed_op* |
`(` *expression_list* `)` | *structuration_op*

expression_list ::= *expression* { `,` *expression* }

A.6.1 Constante

Toutes constantes entières, booléennes, réelles ou déclarées peuvent être utilisées dans les expressions

A.6.2 Identificateur

Toutes les variables utilisées dans une expression doivent être déclarées dans le noeud où elle apparaît.

A.6.3 Opérateurs unaires

unary_op ::= `not` | `-` | `pre` | `current` |

A.6.4 Opérateurs binaires

binary_op ::= `->` | `when` | `xor` | `or` | `and` | `=` | `>` | `>=` | `<` | `<=` |
`+` | `-` | `*` | `/` | `%` | `**` |

A.6.5 Opérateurs ternaires

ternary_op ::= `if` *expression* `then` *expression* `else` *expression* |
`with` *expression* `then` *expression* `else` *expression*

A.6.6 Opérateurs préfixes

Ce sont l'opérateur #, les appels de noeuds ou de fonctions et les fonction de conversion de types.

prefixed_op ::= **ident** (*expression_list*) ‖ **#** (*expression_list*) ‖
int (*expression*) ‖ **real** (*expression*) ‖

A.6.7 Structure et tableau

structure_op ::= [*expression_list*] ‖ *expression* ^ *expression* ‖
expression | *expression* ‖ *expression* [*expression_list*] ‖
expression ‖ *expression* .. *expression* [**step** *expression*]

A.6.8 Signature des opérateurs

A.6.8.1 Les opérateurs sur les valeurs

A.6.8.1.a Opérateur conditionnel :
les expressions `if e1 then e2 else e3 with e1 then e2 else e3` sont de type `t` si les expressions `e1`, `e2`, `e3` ont pour types respectifs `bool`, `t`, `t` où `t` est un type atomique quelconque.

A.6.8.1.b Opérateurs logiques :

- **and** et
- **or** ou inclusif
- **xor** ou exclusif
- **not** négation

les expressions `e1 and e2`, `e1 or e2` et `e1 xor e2` est de type `bool` si les expressions `e1` et `e2` sont de type `bool` le résultat de l'expression `not e1` est de type `bool` si l'expression `e1` est de type `bool`.

A.6.8.1.c Opérateurs arithmétiques :

- + somme
- - différence (binaire)
- - opposé (unaire)
- * multiplication

- `/` division
- `%` modulo
- `**` puissance

les expressions `e1 + e2`, `e1 - e2`, `e1 * e2`, `e1 / e2` et `e1 div e2` est de type `int` si les expression `e1` et `e2` sont de type `int`, de type `real` si les expressions `e1` et `e2` sont de type `real`.

l'expression `- e1` est de type `int` si l'expression `e1` est de type `int`, de type `real` si l'expression `e1` est de type `real`.

les expressions `e1 mod e2` et `e1 % e2` sont de type `int` si les expressions `e1` et `e2` sont de type `int`.

l'expression `e1 ** e2` est de type `int` si les expression `e1` et `e2` sont de type `int`, de type `real` si les expressions `e1` et `e2` sont respectivement de type `real` et `int`.

A.6.8.1.d Opérateurs de comparaison :

- `=` égalité
- `<>` différence
- `>` supériorité stricte
- `>=` supériorité
- `<` infériorité stricte
- `<=` infériorité

les expressions `e1 > e2`, `e1 >= e2`, `e1 < e2` et `e1 <= e2` sont de type `bool` si les expressions `e1` et `e2` sont de type `t` oú `t` est `int` ou `real`.
les expressions e_1 `=` e_2 et e_1 `<>` e_2 sont de type `t` si les expressions e_1 et e_2 sont de type `t` où `t` est un type atomique quelconque.

A.6.8.1.e Opérateurs de conversion :

- `int` conversion d'un réel en entier
- `real` conversion d'un entier en réel

L'expression `int (e1)` est de type `int` si l'expression `e1` est de type `real`.
L'expression `real (e1)` est de type `real` si l'expression `e1` est de type `int`.

A.6.8.2 Opérateurs sur les suites

A.6.8.2.a Opérateur de retard :

L'expression `pre e1` est de type `t` si l'expression `e1` est de type `t` où `t` est un type atomique quelconque.

A.6.8.2.b Opérateur d'initialisation :

L'expression `e1 -> e2` est de type `t` si les expressions `e1` et `e2` ont pour type `t` où `t` est un type atomique quelconque.

A.6.8.2.c Opérateur d'échantillonnage :

L'expression `e1 when e2` est de type `t` si les expressions `e1` et `e2` ont pour type respectifs `t` et bool où `t` est un type atomique quelconque.

A.6.8.2.d Opérateur de projection :

L'expression `current e1` est de type `t` si l'expression `e1` est de type `t` où `t` est un type atomique quelconque.

A.6.8.3 Les opérateurs de manipulation de tableaux

A.6.8.3.a Opérateur de sélection :

- L'expression `e[e1, e2, ... , en]` sélectionne un sous tableau d'un tableau `e` à n dimensions. `e1`, `e2`, ... , `en` sont des entiers ou des tableaux d'entiers statiques.
- L'expression `e.l` sélectionne un élément de label `l` d'une structure.

A.6.8.3.b Opérateur de concaténation :

L'expression `e1 | e2` est de type `t^(n1 + n2)` si les expressions `e1` et `e2` sont respectivement de type `t^n1` et `t^n2`.

A.6.8.3.c Opérateurs de construction de tableau :

- L'expression `e1 .. e2 step e3` construit un tableau d'entiers à partir de trois expressions entières statiques `e1`, son premier élément, `e2`, son dernier élément, `e3` le pas entre deux éléments consécutifs.

- L'expression `e1 ^ e2` construit un tableau dont chaque élément est une copie de `e1` et dont la taille est fournie par l'expression entière `e2`
- L'expression `[e1, e2, ... , en]` construit un tableau mono-dimensionnel de n éléments si `e1`, `e2`, ... `en` sont de même type.
- L'expression `[l1:e1,l2:e2, ... , ln:en]` construit une structure de n éléments l1, l2, ..., ln étant n label différents.

A.6.9 Priorités

Les priorités des opérateurs sont par ordre croissant:

1. `if_then_else`, `with_then_else`
2. `->`
3. `or`, `xor`
4. `and`
5. `not`
6. `..`, `step`
7. <, <=, >, >=, =, <>
8. +, − (binaire)
9. *, /, %
10. ^
11. **
12. `when`
13. − (unaire), `pre`, `current`, `int`, `real`
14. [] (selection), |

A.7 Souplesse de la syntaxe

Quelaues points de la syntaxe ont été assouplis pour conserver la compatibilité avec les syntaxes précédentes de LUSTRE.

- Le mot clé `tel` peut remplacer le mot clé `end` en fin de déclaration de noeud.
- Le point virgule terminant la déclaration d'un noeud peut ^etre omis ou remplacé par un point
- Un point virgule peut ^etre ajouté après le dernier param^etre d'entrée ou de sortie d'un noeud ou d'une fonction.

Annexe B

Manuel utilisateur du compilateur Pollux

B.1 utilisation de la commande POLLUX

La commande pollux peut prendre deux formes :

- pollux *filename nodename* [-e] [-p] [-h] [-s] [-l] [-b] [-0] [-9] [-d] [-t] [-v] [-g *filename*] [-y] [-D *directory*] [-v] [-I] [-L] [-z] [-x] [-o *filename*] [-O] [-E] [-C] [-m] [-r] [-j *filename*] [-K] [-A] [-R *name*] [-1 *chipname clockname*] [-c] [-S] [-u]
- pollux [-i *filename*] [-n *nodename*] [-e] [-p] [-h] [-s] [-l] [-b] [-0] [-9] [-d] [-t] [-v] [-g *filename*] [-y] [-D *directory*] [-v] [-I] [-L] [-z] [-x] [-o *filename*] [-O] [-E] [-C] [-m] [-r] [-N] [-j *filename*] [-K] [-A] [-R *name*] [-1 *chipname clockname*] [-c] [-S] [-u]

filename est le nom d'un fichier LUSTRE textuel (ASCII, suffixe .lux ou .lus).
gfilename est le nom d'un fichier LUSTRE graphique (POSTSCRIPT-IDRAW, suffixe .ps).
nodename est le nom du noeud principal du programme à compiler.
directory est le nom d'un répertoire.
name est un identificateur.
chipname est le nom d'un circuit XILINX.
clockname est 0 ou 1.

B.2 Options

Les deux premières options indiquent le fichier source à utiliser et sa nature (texte ou graphiques). Ces deux options sont exclusives entre elles et l'une des deux doit toujours être présente.

- -i *filename* : indique le nom du fichier textuel (ASCII) contenant le programme à compiler. Cette option provoque l'analyse syntaxique du programme et la construction de la structure de données interne.

- -g *filename* : indique le nom du fichier graphique contenant le programme à compiler. Ce fichier doit avoir été réalisé avec l'éditeur de dessins IDRAW. Cette option provoque l'interprétation des schémas et la construction de la structure de données interne.

Une partie d'un programme peut également être prise dans un second fichier texte.

- -j *filename* : indique de prendre les équations de placement dans le fichier *filename*. Ce fichier est composé d'un ensemble de déclarations :

```
node N
let

équations

tel;
```

ou *équations* est un ensemble d'équations de placement.

- -n *nodename* : indique le nom du noeud principal du programme à compiler. Si cette option n'est pas spécifiée, rien n'est compilé.

Les options suivantes spécifient les opérations à réaliser sur la structure de données produite.

- -e : réalise l'expansion du programme. Le polymorphisme du programme est enlevé, les tableaux sont éclatés, les horloges sont calculées, les appels sont remplacés par une copie du noeud.
- -h : réalise la transformation du programme en un circuit synchrone. Les opérateurs temporels sont supprimés. Doit être utilisée avec l'option e.
- -p : réalise la transformation du programme en un circuit synchrone destiné à être implanté sur la PAM. Les opérateurs temporels sont supprimés, certaines configurations d'opérateurs sont reconnues et optimisées. Doit être utilisée avec l'option -e.
- -v : réalise un certain nombre de vérifications sémantique pour s'assurer que le programme source est bien correct.
- -S : appelle le module de compilation séparée. (Réalisé par Christian Dubois)
- -D *directory* : indique dans quel repertoire le compilateur peut trouver les fichiers de données dont il a besoin (simul.c, Prolog.ps, Init.ps, Conc.ps, norprog.H). Si cette option n'est pas spécifiée, le programme prend le nom du répertoire donné par la variable d'environnement TOPDIR. Si enfin cette variable d'environnement n'est pas définie le répertoire par défaut est *./DATA*.
- -P *dir* : indique dans quel répertoire trouver les fichier à inclure. Il est possible de spécifier plusieurs répertoires à l'aide de plusieurs options P.
- -I : indique qu'aucune équation identité ne doit être supprimée lors de la compilation.

- -L : indique que toutes les variables locales deviennent des variables de sorties (utile pour la simulation, pour avoir les chronogrammes des variables locales)
- -z : indique que les entiers 0 et 1 ne peuvent pas être utilisés pour représenter les constantes **false** et **true**. Cette option est automatiquement ajoutée si l'option v est spécifiée
- -x : les paramètres des fonctions ne sont pas éclatés. Cette option est automatiquement ajoutée si l'option s est spécifiée
- -a : génère une table d'alias qui donne à chaque variable un nom permettant de la retrouver dans le programme source. Le nom d'une variable est préfixé du chemin à parcourir dans l'arbre des appels depuis le noeud principal pour parvenir à celui où elle est déclarée.
- -O : supprime le calcul d'horloge
- -C : utilise le vieil algorithme de calcul d'horloge.
- -r : évalue toutes les expressions statiques.
- -u : force la destruction des variables locale inutiles.
- -R *identifier* : préfixe avec *identifier* le nom de toutes les variables du programme.

Les options suivantes sont destinées à produire les résultats dans un format donné

- -l : produit un programme LUSTRE. Elle peut être utilisée avec l'option -e pour produire le fichier LUSTRE après expansion.
- -E : produit un programme LUSTRE noyau avec une syntaxe plus ancienne et restrictives définies dans le manuel utilisateur de LUSTRE V2 :
  - Les déclarations de types doivent être faite en premier. Chacune est terminée par une virgule sauf la dernière terminée par un point.
  - Les déclarations de constantes doivent être faite en second. Chacune est terminée par une point-virgule sauf la dernière terminée par un point.
  - Les déclarations de fonctions doivent être faite en troisième. Chacune est terminée par un point.
  - Les déclarations de noeud sont faites en dernier elles se terminent par un point virgule sauf la dernière terminée par un point.

  Il s'agit d'un programme ec si elle est utilisée avec l'option -e. A utiliser pour interfacer POLLUX avec de vieux outils LUSTRE.
- -s : produit un programme C. Il est composé :
  - D'une fonction qui calcule un cycle du programme source à chaque appel
  - De fonctions d'interfaces permettant de lire les entrées et produire les sorties.
  - D'un programme principal qui réalise l'itère le calcul.

  . Doit être obligatoirement avec l'option -e.

- -c : produit une fonction C++ qui calcule un cycle du programme source à chaque appel. Doit être utilisée obligatoirement avec l'option -e.
- -0 : produit une fonction LISP au format Perle0. les variables non placées dans le programme source sont pre-placées en (x, y). Doit être utilisée avec l'option -e ainsi que l'option -h ou l'option -p.
- -1 *device clock*: produit un fichier LISP complet au format Perle0. Produit la même fonction qu'avec l'option 0 mais l'encapsule dans une autre qui se charge de faire les différentes initialisations nécessaires pour entrer dans les outils de CAO de la PAM.
  - *device* définit le type du circuit xilinx sur lequel le circuit va être implanté.
  - it clock spécifie quel réseau d'horloge, 0 ou 1, va être employé.

  Doit être utilisée avec l'option -R pour spécifier le nom associé au circuit produit. Doit être également utilisée avec l'option -e ainsi que l'option -h ou l'option -p.
- -9 : produit un fichier LISP au format Perle0. Les variables non placées le sont au même endroit que la variable placée la plus proche si elle existe, pre -placées en (x, y) sinon . Doit être utilisée avec l'option -e ainsi que l'option -h ou l'option -p.
- -d : produit un fichier POSTSCRIPT contenant une représentation graphique du programme source avant ou après expansion selon la présence de l'option -e. Ce fichier POSTSCRIPTpeut être lu par l'éditeur de dessins IDRAW et par POLLUX avec l'option -g.
- -t : produit une trace de la structure de données interne. Réservé aux utilisateurs avertis.
- -b : produit un fichier BLIF. Doit être utilisée avec l'option -e ainsi que l'option -h ou l'option -p.
- -m : produit un programme MODEL. Doit être utilisée avec les options -e et -h.
- -y : affiche une trace des arbres abstraits produits par la phase d'analyse syntaxique
- -A : utilise le nom complet des variables dans les descriptions.
- -K : supprime l'impression de `when true` dans les déclarations de variables lors de l'affichage d'un programme LUSTRE.

La dernière option sert à rediriger les sorties.

- -o *filename* : Met les résultats produits par POLLUX dans le fichier *filename* par défaut, ces résultats sont affichés à l'écran.

B.2.1 Utilisations usuelles

B.2.1.1 Simulation

Si *file* est le nom du fichier exécutable de simulation, obtenu après àprès compilation d'un source LUSTRE puis compilation du fichier C résultat, son lancement est réalisé par une ligne de commande ayant le format suivant :

file [-c] [-i *ifile*] [-o *ofile*]

ifile et *ofile* sont deux fichiers de données respectivement d'entrée et de sortie.

L'option -g permet de spécifier que le format des données utilisé est le même que celui utilisé par l'outil de visualisation de chronogrammes CHRONO.
La première partie des données est constituée d'un ensemble de déclarations de signaux, ayant la syntaxe suivante :

signal_declaration	::=	*signal_decl* { *signal_decl* }
signal_decl	::=	*ident integer*

ident représente le nom du signal et *integer* sa taille en nombre de bits.
La seconde partie est composées d'une suite de définitions des valeurs des signaux pour un cycle donné. La syntaxe

simulation_data	::=	*one_cycle_data* { *one_cycle_data* }
one_cycle_data	::=	# *integer* { *integer* }

Le premier entier indique le numéro du cycle, les autres les valeurs des signaux, données dans le même ordre que leurs déclarations.

Le simulateur reçoit, dans ce format, les données de simulations des entrées et et produit en sortie ces mêmes données augmentées de celles des sorties.

Lorsque qu'aucune de ces option n'est spécifié, le mode de simulation par défaut est le même que celui des autres simulateurs PLUS ou LUCS. Les données sont une simple suite de valeurs des entrées, dans le même ordre que leur déclaration dans le programme, au premier instant puis au second ...

Les options suivantes permettent de rediriger les flots 'entrées sorties

- i *ifile* : les données d'entrée sont lues dans le fichier *ifile* et non au clavier.
- o *ofile* : les résultats sont placés dans le fichier *ofile* et affichées à l'écran.

Lorsque les entrées son lues au clavier plutôt que dans un fichier, le simulateur se charge de demander les données dont il a besoin. L'utilisateur n'a qu'à rentrer les valeurs (dans le cas des booléens 0 pour **false** et 1 pour **true**) et

- dans le mode par défaut le caractère '.'à la place d'une valeur permet l'arrêt de la simulation.
- dans le mode chronogramme (options c) la fin d'une séquence de valeurs est indiquée par le caractère retour de chariot.

pollux -i *file.lux* -n *nodename* -e -s -o *file.c*
cc file.c -o file
file -g -i data.in -o data.out
chrono data.out

B.2.1.2 Execution sur la PAM

pollux -i *file.lux* -n *nodename* -e -p [-0] [-9] -o *file.ll*
lelisp
...

B.2.1.3 Production du fichier ec

pollux -i *file.lux* -n *nodename* -e -l -o *file.ec*

B.2.1.4 Production de schémas

pollux -i *file.lux* -n *nodename* [-e] -d > *file.ps*
idraw file.ps

B.2.1.5 Interprétation de schémas

à vérifier

pollux -g -l *file.ps* -o *file.lux*

AUTORISATION de SOUTENANCE

VU les dispositions de l'Arrêté du 23 novembre 1988 relatif aux Etudes doctorales

VU les rapports de présentation de

- Monsieur BERRY Gérard
- Monsieur QUINTON Patrice

Monsieur ROCHETEAU Frédéric

est autorisé(e) à présenter une thèse en soutenance en vue de l'obtention du diplôme de DOCTEUR de L'INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE, spécialité " Informatique "

Fait à Grenoble, le 17 juin 1992

Pour le Président de l'INPG
et par délégation
le Directeur de l'Ecole Doctorale
J.L. LACOUME

Résumé : L'objectif de ce travail, est l'étude des applications possibles du langage déclaratif synchrone LUSTRE à la description de circuits matériels, et plus particulièrement, étant donné ses caractéristiques, son aspect flot de données notamment, aux chemins de données.

Par rapport aux systèmes temps réels, dont la programmation et la vérification constituent le domaine d'application privilégié de ce langage, les circuits synchrones sont généralement beaucoup plus réguliers mais également de taille plus importante.

Afin de permettre une description aisée de ces applications, une extension de LUSTRE a été développée. Elle ajoute au langage, tout en conservant sa sémantique rigoureuse : des données structurées et les opérateurs permettant leur manipulation, une notion de sous programmes génériques, la récursivité bornée,

Le système POLLUX a ensuite été développé autour de ce langage, il se décompose en :

- un pre-compilateur chargé de l'analyse sémantique d'un programme. afin de s'assurer de sa consistance.
- Divers post-processeurs dont le principal est un générateur de circuits synchrones implantés ensuite sur la PAM, une machine à base de circuits prédiffusés programmables. Les autres principaux post-processeurs sont un générateur de code séquentiel classique, une interface graphique permettant le passage d'un programme textuel à un ensemble de schémas et vice-versa.

Mots-clés : flot de données, langage synchrone, conception de circuit, chemin de données, compilation, outils de CAO, circuits prédiffusés programmables