Lustre V6

Synchronous Team
VERIMAG, Grenoble

Lustre Basics

Structuration

- Only nodes

Temporal operators

- memories : `pre`, `->`
- clocks : `when`, `current`

Data types/operators

- `bool`, `int`, `real` + all arithmethic and logic operators
- all the rest : abstract (imported) types, contants and functions

Lustre V4

- Aka Lustre/Pollux, F. Rocheteau 92
- Hardware oriented
- Static arrays serve both as data type and program structuration

Arrays and program structure

- Generic nodes (parameterized by array sizes)
- Static recursion (`with` = lazy `if`)

Arrays manipulation

- **main idea : index variable free**
- **homomorphic extension :**

  if `X` and `Y` are of type `int^n`

  then `X + Y` is of type `int^n`
- **slicing :**

  if `X` is of type `T^n`, then `X[i..j]` with $0 \leq$ `i` $\leq$ `j` $<$ `n`

  is of type `T^(j-i+1)`
- **recursive definition of arrays : `X = f(X)` correct**

  iff $\forall$`i` `X[i]` does not depend instantaneously on itself

  (i.e. the equivalent expanded code is correct)

LV4 examples

- **Or'ing a Boolean array :**

```
node BigOr(const n:int; X:bool^n)
returns (res :  bool);
var T : bool^n;
let
  T[0] = X[0];
  T[1..n-1] = T[0..n-2] or X[1..n-1];
  res = T[n-1];
tel
```

- **Remarks :**
  - ★ correct because `T[i]` depends only on `T[0]` ... `T[i-1]`
  - ★ dependencies hard to check in general (needs expansion)
  - ★ problem for generating good code (e.g. `for` loops)

- **Recursive version :**

```
node BigOr(const n:int; X:bool^n)
returns (res :  bool);
let
  res = with (n = 1) then X[0]
      else X[0] or BigOr(n-1, X[1,n-1]);
tel
```

- **Static recursion is really powerful !**

  e.g. binary decomposition leads to logarithmic circuits

```
node Max(const n:int; X:int^n)returns(mx:int);
var m1, m2 :  int;
let
  m1 = with (n=1) then X[0]
       else Max(n div 2, X[0..(n div 2)-1]);
  m2 = with (n=1) then X[0]
       else Max((n+1) div 2, X[n div 2..n-1]);
  mx = if m1 >= m2 then m1 else m2;
tel
```

Critics of LV4

- array mechanism not suitable for software (expansion)
- lack of program structures (name space)
- lack of structured data-type (records are heterogenous arrays)

Notes on LV5

- aka `lus2dc`
- better organized, but still “v4”-compliant

Arrays and iterators

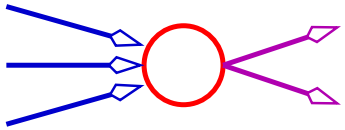
- **no longer lv4 compliant**
- **self reference forbidden**
- **homomorphic extension suppressed**
- **iterators for defining recursive arrays: `map`, `red`, `fill` and `mapred`**
- **usage: *iterator*<< *node* , *size*>> (*dynamic-params*)**

The “map” iterator

- **For any node N of sort**

  $\tau_1 \times \ldots \times \tau_k \rightarrow$

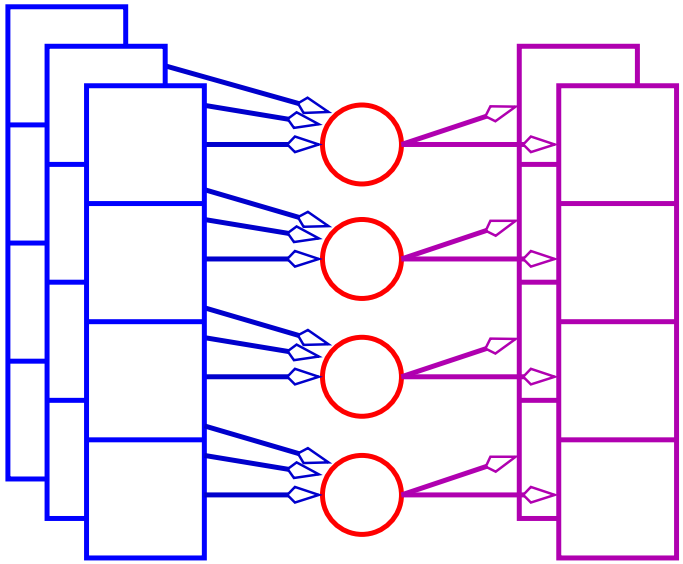
  $\theta_1 \times \ldots \times \theta_\ell$



- **`map<<N,n>>` is of sort**

  $\tau_1\hat{}n \times \ldots \times \tau_k\hat{}n \rightarrow$

  $\theta_1\hat{}n \times \ldots \times \theta_\ell\hat{}n$



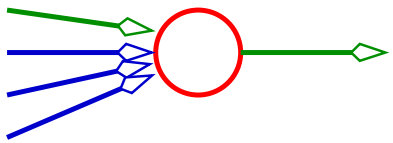
- **Example: `map<<+,3>>(X,Y)` means**

```
[ X[0]+Y[0], X[1]+Y[1], X[2]+Y[2] ]
```

The “red” iterator

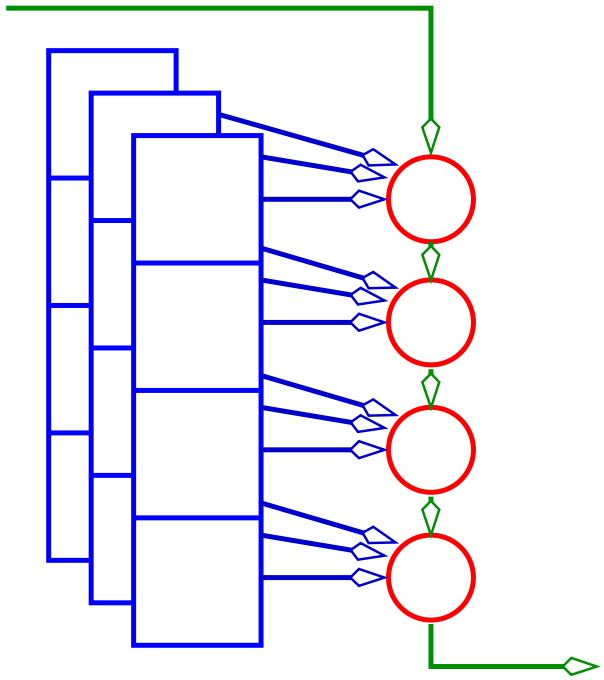
- For any node `N` of sort

  $\tau \times \tau_1 \times \ldots \times \tau_k \rightarrow \tau$



- `red<<N,n>>` is of sort

  $\tau \times \tau_1\hat{}n \times \ldots \times \tau_k\hat{}n \rightarrow \tau$



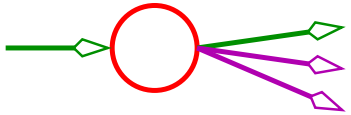
- Example: `red<<or, 4>>(false, X)` means

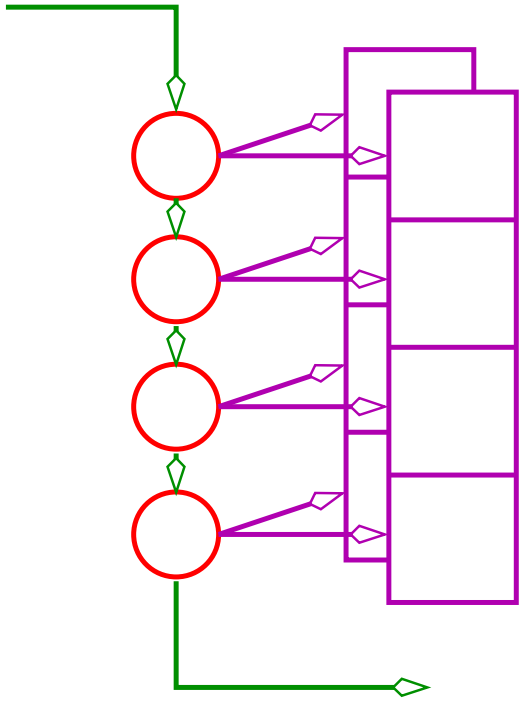
```
((((false or X[0]) or X[1]) or X[2]) or X[3])
```

The “fill” iterator

- For any node N of sort

  $\tau \longrightarrow \tau \times \theta_1 \times \ldots \times \theta_\ell$





- `fill<<N,n>>` is of sort

  $\tau \longrightarrow \tau \times \theta_1\hat{}n \times \ldots \times \theta_\ell\hat{}n$

- Example: given `node Incr(i:int)returns(j,k:int);`

  `let j=i+1; k=i; tel`

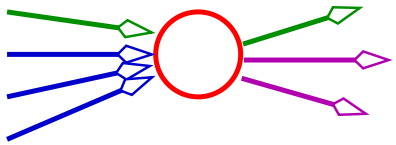
  `fill<<Incr, 4>>(0)` means `([0,1,2,3], 4)`

The “mapred” iterator

- For any node N of sort

  $\tau \times \tau_1 \times \ldots \times \tau_k \rightarrow$

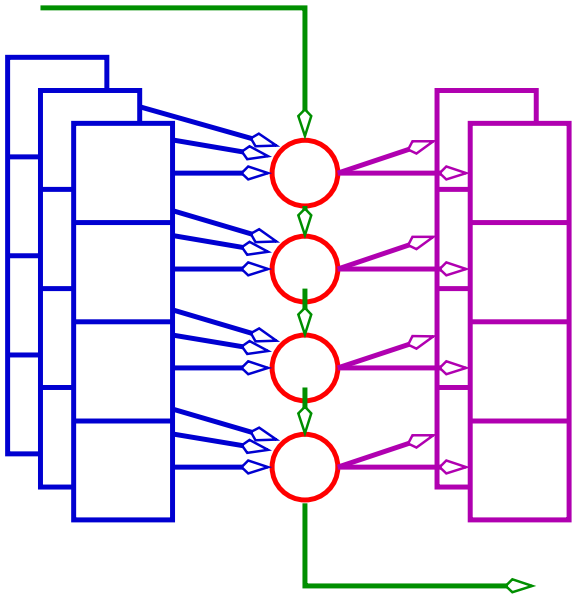
  $\tau \times \theta_1 \times \ldots \times \theta_\ell$



- red<<N,n>> is of sort

  $\tau \times \tau_1\hat{}n \times \ldots \times \tau_k\hat{}n \rightarrow$

  $\tau \times \theta_1\hat{}n \times \ldots \times \theta_\ell\hat{}n$



- Example: Given a 3bits adder

```
node fulladd( cin, x, y:bool)returns( cout, s:bool)
```

get an unsigned byte adder with:

```
(overflow,S) = mapred<<fulladd,8>>(false,X,Y);
```

Packages

- A name space for items,
- Items are types, constants, nodes,
- Items can be provided (exported) or hidden (private).
- Other packages may be used : if `Foo` is a package providing an item `bar`, then `Foo:bar` denotes the item.
- A program: a main node in a main package in a list of packages.

```
package Observer
uses StdCtrl
provides
  type safety;
  const TRUE: safety;
  const FALSE: safety;
  node EventOfBool(x: bool)
      returns(e: StdCtrl:event);
  node OnceFromTo(a,b,c: StdCtrl:event)
      returns(s: safety);
  node AlwaysFromTo(a,b,c: StdCtrl:event)
      returns(s: safety);
  node And(x,y: safety)returns(s: safety);
body
  (* lustre definitions *)
end
```

Models

- **Example**

```
model Binary
needs const SIZE: int;
provides type t;
           const ZERO: t;
           node chs(x: t)returns(s: t); ...
body
  type t = bool^SIZE;
  const ZERO = false^SIZE;
  node _ichs(ci,x:bool)returns(co,y:bool);
    let y= ci xor x; co= ci or x; tel
  node chs(x: t)returns(s: t);
    var c: bool;
    let (c,s)= mapred<<_icsh,SIZE>>(false, x); tel
end
```

- **Model = parameterized package.**
- **A package can be defined as an instance of model:**

```
package Bin8 is Binary(8);
package Bin16 is Binary(16);
```

- **Only model instances can be used:**

```
package Foo
uses Bin16   (* NOT: Binary(16) !!  *)
provides ...
body ...  end
```

Data types

Enumerated types

- `type color = enum {blue, white, red};`
- Once declared, an enum value behaves as *constant*:
  `blue` can no longer be used as a variable.

Structured types

- `type binres = {val: bool^16; over: bool};`
- creation: `{val: true^16, over: false}`
- reference: `var X: binres; ... X.val[15] ...`

Type equivalence

- **Structural equivalence:**
  - ⋆ $\tau\hat{}n \equiv \theta\hat{}m$ **iff** $\tau \equiv \theta$ **and** $n = m$
  - ⋆ $\{n_0:\tau_0;\cdots;n_i:\tau_i\} \equiv \{m_0:\theta_0;\cdots;m_j:\theta_j\}$
    iff $i = j$ **and** $\forall k \;\; n_k = m_k$ **and** $\forall k \;\; \tau_k \equiv \theta_k$
- **example:** `type binres = {val: bool^16; over: bool};`
  $\equiv$ `{val: Bin16.t; over: bool}`
  $\not\equiv$ `{x: bool^16; over: bool};`
  $\not\equiv$ `{over: bool; val: bool^16};`

Generalized activation condition

Basics: merge

- Not a flow operator, but rather a node operator
- `merge<<N1,N2>>(c, X, Y)`

  is equivalent to:

```
if c then current(N1(X when c))
     else current(N2(Y when not c))
```

- This binary merge is not provided: a more general n-ary merge is poposed.

Generalized merge

```
case c1 do   -- if c1 then
     N1(X1)  --  current(N1(X1 when c1)) else
case c2 do   -- else if c2 then
     N2(X2)  --  current(N2(X2 when (not c1 and c2))
....
default      -- else
     Nn(Xn)  --  current(Nn(Xn when not(c1 or c2 or ...)))
```

- all `Ni` have the same output type
- at least one `case` statement and the `default` statement

  `case c do N1(X) default N2(Y)` equiv.

  `merge<<N1,N2>>(c, X, Y)`

Implicit "*node-ification*"

- More convenient way to write case:

```
D = case (Z <> 0) do X / Z default 0;
```

- Each expression must be interpreted as a *online node definition:*

  `X/Z` stands for `N(X,Y)`, with

```
node N(a,b:real)returns(r:real);
let r = a / b; tel
```

- Warning: using `pre` and `->` in "node-ification" is error prone:

```
X = case incr do (0 -> pre X) + 1
    case decr do (0 -> pre X) - 1
    default (0 -> pre X);
```

  Each occurence of `(0 -> pre X)` denotes a different flow !!!

  $1, 2, 3...$ on "`incr`", $-1, -2, -3...$ on "`decr and not incr`",

  $0, 0, 0...$ on "`not(decr or incr)`"

- **The “right” one:**

```
PX = 0 -> pre X;  -- outside the case: same clock as X
X = case incr do PX + 1
    case decr do PX - 1
    default PX;
```

- **Similar example: Scade “condact”**

```
Y = condact<<N>>(c,X,V)
≡ Y=if c then current N(X when c)else(V->pre Y);
≡ PY=V->pre Y; Y=case c do N(X) default PY;
```