Programming with arrays in a data-flow languages

The Lustre experience(s)

Pascal Raymond

Verimag-CNRS

MOSIG - Embedded Systems

Introduction

Dedicated languages

- Synchronous languages (like Lustre) are not general purpose
  - ↪ They are dedicated to reactive kernel
  - ↪ Complex data-types and algorithms are imported from a host language
- But arrays are not “only” data:
  - ↪ They are program structure (arrays of concurrent programs)
  - ↪ Required for scalable, generic programs

    e.g. elevator control system for n floors ?

Challenge: how to design a “safe” notion of array ?

- Forbid memory overflow
- Forbid run-time errors (index out of bound)
- Solution:
  - ↪ Statically sized arrays to guarantee bounded memory
  - ↪ Statically computed indexes: index out of bounds rejected at *compile time*

N.B. Strong restriction, strong guarantees !

- problem of “style”: how to manipulate arrays in a declarative language ?
- i.e. how to deal with array without loops and index variables ?
  - ↪ Lustre V4: (original) slice reasoning
  - ↪ Lustre V6: more classical “higher order” iterator (cf. Functional programming)

Arrays in Lustre-V4

Hardware oriented approach

- Usefull: registers, busses, regular and scalable circuits
- Constraints:
  - ↪ Must fit the general data-flow approach
  - ↪ Don’t introduce unpredictable errors (index out of bound)
- General approach:
  - ↪ No control statement
  - ↪ Sizes and indices are computed at *compile-time*

Array types

- type declarations

```
const sz=8;
type tab = int^sz;
type bigtab = int^(sz*sz);
```

- Important notion: Compile Time Integer expression (CTI)

  ↪ array sizes must be integer expression taht can be evaluated statically

- variable declarations (as usual)

```
A: tab;
B: bool^4;
```

- variable definition (as usual)

```
A = exp;
```

  ↪ where **exp** is of type **tab** = **int^8**

Accessing cells

- If **A** is an array of type **T^n**, its elements are

  A[0], A[1], ..., A[n-1]

- **A[i]** is a correct expression of type **T** iff:

  ↪ **i** is a CTI evaluated to i such that $0 \leq i \leq n$

- can be used as right hand side

  A[i] = exp ;

  is a correct definition iff:

  ↪ **i** is a CTI evaluated to i

  ↪ such that $0 \leq i \leq n$

  ↪ **exp** is correct expression of type **T**

Array expressions

- Immediate arrays

```
[3,10,22,-4]   -- type int^4
true^3         -- i.e [true,true,true]
```

- Concatenation: let **A:T^n** and **B:T^m** be two array exp.

  A | B

  is an array exp. of type **T^(n+m)**

Operating on arrays: Homomorphic Extension

- Each operator or node of type

  $\tau_1 \times \tau_2 \times \ldots \times \tau_k \rightarrow \theta_1 \times \ldots \times \theta_\ell$

  is also, implicitly an array operator

  $\tau_1\hat{}n \times \tau_2\hat{}n \times \ldots \times \tau_k\hat{}n \rightarrow \theta_1\hat{}n \times \ldots \times \theta_\ell\hat{}n$

- Example, **A, B: bool^3**

  A or B $\Leftrightarrow$ **[A[0] or B[0], A[1] or B[1], A[2] or B[2]]**

What about more general operations ?

- i.e. how to program complex algo. without loops and index variable ?
- $\Rightarrow$ the *def by slice* principle

Slices

- Examples:

  A[2..5] $\Leftrightarrow$ **[A[2], A[3], A[4], A[5]]A[5..2]** $\Leftrightarrow$ **[A[5], A[4], A[3], A[2]]**

- more generally:

$$\texttt{A[i..j]} = \begin{cases} \texttt{[A[i], A[i+1], ..., A[j]]} & \text{if } i \leq j \\ \texttt{[A[i], A[i-1], ..., A[j]]} & \text{if } j < i \end{cases}$$

- General form:

  A[i..j step k]

- N.B. Slices may also appear as right hand side of an equation

Valid array definitions

- A variable array (output, local) `A: T^N` can be declared
  - ↪ globaly, with a valid array expression `A = exp;`
  - ↪ or with any combination of slice/cell equations, as far as:
    - ∗ *each cell is defined exactly once*
    - ∗ *no cell depends instantaneously on itself* (no combinational loop)
- Example, for `A:int^6` :

```
A[0] = 0;
A[1..3] = 2^3;
A[4..5] = [7, 9];
```

- Example, for `B:int^4` :

```
B[0..4 step 2] = 0^3;
B[1..5 step 2] = 1^3;
```


Example: filling an array with consecutive int.

- First (verbose) solution

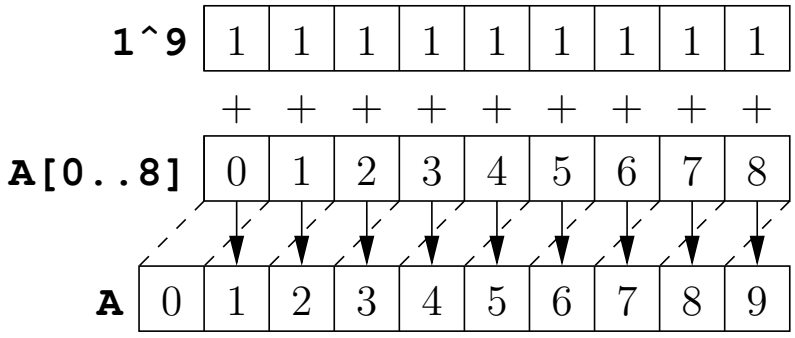
```
A = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9];
```

- Second (slice-based) solution

```
A[0] = 0;
A[1..0] = A[0..8] + 1^9;
```

or equivalently:

```
A = [0] | (A[0..8] + 1^9);
```



Generic nodes

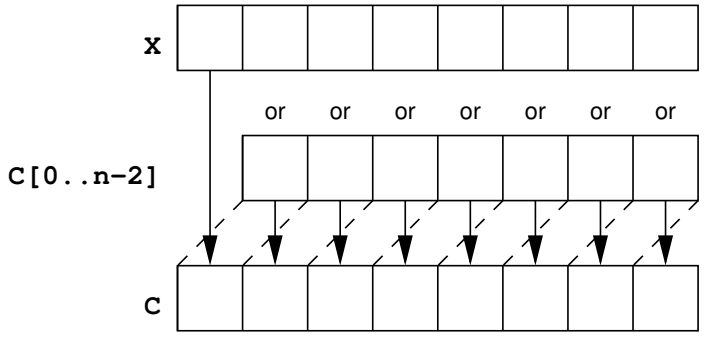
- Parameterized by a static constant (typically an array size)
- Example: fill an integer array with $x, \cdots x + n - 1$

```
node fill_int(const n:int; x:int) returns (T:int^n);
let
  T = [x] | (T[0..n-2] + 1^(n-1));
tel
```

- Usage: `A = fill_int(10,0);`

Generic node example: cumulated "or"

```
node Cor(const n:int; X:bool^n) returns (o:bool);
var C:bool^n;
let
  C = [X[0]] | (C[0..n-2] or X[1..n-1]);
  o = C[n-1];
tel
```



Homomorphic extension of user-defined node

- Use a full-adder (aka 3 bits adder)

```
node fadd(cin,x,y:bool) returns (cout,s:bool);
let
  s = cin xor x xor y;
  cout = if cin then (x or y) else (x and y);
tel
```

- ... to define a n bits parallel adder:

```
node ADD(const n:int; X,Y:bool^n) returns
  (S:bool^n; overflow:bool);
var C: bool^(n+1);
let
  C[0] = false;
  C[1..n], S = fadd(C[0..n-1],X,Y);
  overflow = (C[n] <> C[n-1]);
tel
```


Static recursion

- Based on *static conditional*:

  with cond then exp1 else exp2

  where **cond** is a *compile-time Boolean*

- Example: recursive cumulated ”or”

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

- Alternative way to program generic array nodes without the need of ”buffers”

Static recursion (cntd)

- Advanced example: dichotomy

```
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
```

- Interest versus ”linear” Max ?
  - ↪ for software, none ($n-1$ comparisons each)
  - ↪ for hardware, yes: log(n) versus n depth !

Summary on Lustre V4 arrays

- Expressiveness
  - ↪ Some limitation, restrictions (no dynamic access)
  - ↪ Satisfactory for a wide range of problems
- Execution, compilation
  - ↪ Static recursion and (complex) slice manipulation require expension
  - ↪ “Natural” for circuit design (hardware)
  - ↪ Unadapted for compilation (software)

Note on array compilation

- Good (efficient, compact) sequential code generation
  - ↪ save code: loop statements instead of replicating similar code
  - ↪ save memory: avoid useless intermediate arrays
- Example of (expected) good code:

```
-- Lustre source
node Cor(
  const n:int;
  X:bool^n
) returns (o:bool);
var C:bool^n;
let
  C = [X[0]] |
    (C[0..n-2] or X[1..n-1]);
  o = C[n-1];
tel
```

```
//Generated C code
bool Cor(
  const int n,
  bool X[]
){
  bool c=X[0];
  int i;
  for(i=1;i<n;i++){
    c |= X[i];
  }
  return c;
}
```


Note on array compilation (cntd)

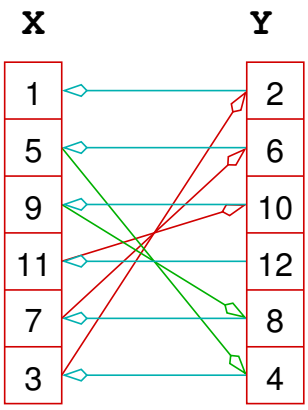
- Generating efficient loop statements from arbitrary slice manipulation is hopeless

```
Y = f(X);

X[0] = a;

X[1..2] = g(Y[5..4);

X[5..3] = h(Y[0..3]);
```



- Correct: a computation order exists, but impossible to use loop...
- Possible solution:
  - ↪ recognize and optimize some “loop patterns”
  - ↪ expand all the rest...
- Or change the language !

Arrays in Lustre-V4

Software oriented approach

- Key idea: forbid arbitrary dependencies within an array
- Inherits from V4 the basic array features
  (declaration, access, slice, constructor etc.
- But not fully compatible:
  - ↪ Array slice overlap forbidden in definition
  - ↪ Homomorphic extension suppressed
- This complex features are replaced by a set of pre-defined iterators

Iterators

- Classical solution in *functional languages*,
  where they replace imperative loop statements
  e.g. “map”, “fold” in ML
- Problem:
  - ↪ Iterators are *higher-order operators*...
  - ↪ .. don’t want to make Lustre an higher-order Language !
- Solution:
  - ↪ Iterators cannot be *programmed* in Lustre
  - ↪ A (small) set of built-in iterators are provided

Iterators (cntd)

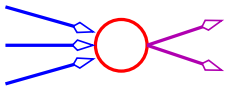
- General syntax: **`iterator<<operator,size>>`**
  - ↪ **`operator`** denotes a node or function
  - ↪ **`size`** is a compile-time integer constant, denoting the number of required iteration,
  - ↪ The whole construction denotes a *node*, whose profile depends on:
    - its particular semantics,
    - the profile of the iterated **`operator`**
    - the **`size`**
- Let see in details the 4 built-in iterators...

The **`map`** iterator

- Explicit homomorphic extension
- For any node **`P`** of sort

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

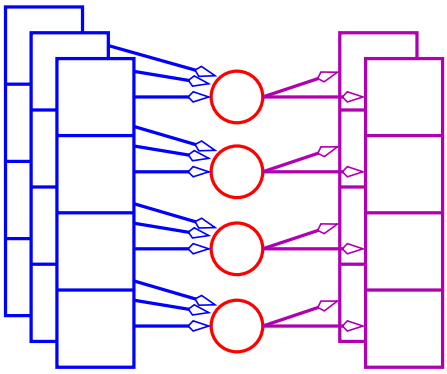
  $\theta_1 \times \ldots \times \theta_\ell$



- **`map<<P,n>>`** is a node 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, let **`X, Y: int^3`**

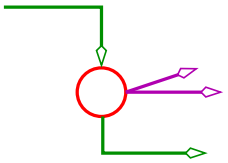
  `map<<+,3>>`

  equiv. **`[X[0]+Y[0], X[1]+Y[1], X[2]+Y[2] ]`**

The `fill` iterator

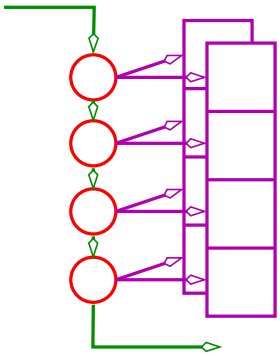
- For any node `P` of sort

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



- `fill<<P,n>>` is a node of sort

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



- Example, let

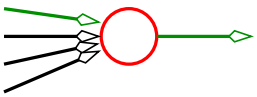
```
node incr(i:int)returns(j,k:int); let j=i+1; k=i; tel
fill<<incr,4>> (0)
```

  equiv. `(4, [0,1,2,3])`

The `red` iterator

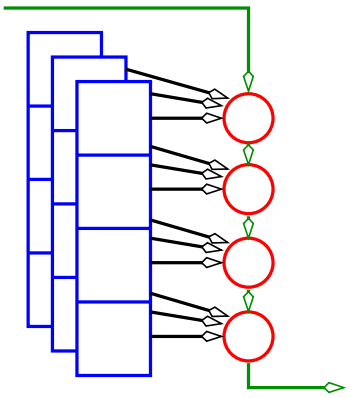
- For any node `P` of sort

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



- `red<<P,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)`

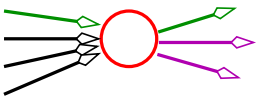
  equiv. `false or X[0] or X[1] or X[2]`

The **mapred** iterator

- For any node **P** of sort

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

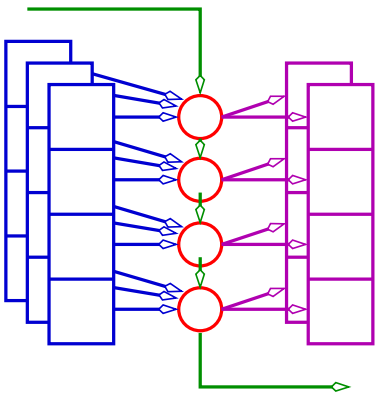
  $\tau \times \theta_1 \times \ldots \times \theta_\ell$



- **mapred<<P,n>>** is of sort

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

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



- Example: Given the full-adder (cf. slide **??**)

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

builds a signed int adder:

```
overflow,S = mapred<<incr,32>>(false,X,Y)
```


Other useful V6 features

- Limited notion of *node expression*
- Avoid the tedious writing of node profile and def
- e.g.

```
node add32 = mapred<<fadd,32>>
```

instead of:

```
node add32(
  cin:bool;X,Y:bool^32
)returns(
  cout:bool;S:bool^32
);
let
  cout, S = mapred<<fadd,32>>(cin,X,Y);
tel
```

Conclusion on Lustre-V6

- Reasonable tradeoff between expressivity, (built-in) safety and code efficiency
  - ↪ allows to “mimic” most of imperative array manipulation
  - ↪ guaranties bounded memory and forbids (array related) run-time errors
  - ↪ straightforward compilation into classical “for” loops
- Similar solution adopted in Scade (industrial version)
  N.B. iterators were a strong demand form Airbus, in order to drastically reduce the size of (generated) code