Arrays in Lustre

P. Caspi, N. Halbwachs, F. Maraninchi, L. Morel, P. Raymond

Verimag/CNRS
Grenoble
1. Arrays in Lustre-V4

2. Arrays in Lustre-V6
Arrays in Lustre-V4 [Rocheteau 92]

Introduced for hardware description (register, regular circuits), and verification (scalable examples).

Expanded by the front-end

Constraints:

- Introduce an array mechanism that fits the general data-flow philosophy of the language
- Don’t introduce the possibility of unpredictable runtime errors (index out of bounds)
Arrays in Lustre-V4 (cont.)

T : int^{42} ;
const SIZE = 16;
type register = bool^{SIZE} ;
R : register ;

(SIZE is an integer constant known at compile time)

Equations (as usual)
R = Exp ;

(Exp is an expression of type register)
Access to array elements
Let A be an array of size $n$. Its elements are

$A[0], A[1], \ldots A[n-1]$

$A[i]$ is legal if $i$ is an integer constant known at compile time, with $0 \leq i \leq n - 1$
Arrays in Lustre-V4 (cont.)

Constructors: \([0, 3, 2]\) \(\text{true}^3 = [\text{true, true, true}]\)

Slices:

\[
\]

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

(i and j are static constants)
Arrays in Lustre-V4 (cont.)

Concatenation:

\[ A \| B = [A[0], A[1], \ldots, A[n-1], B[0], B[1], \ldots, B[m-1]] \]

Lustre **polymorphic** operators

if...then...else..., pre, \( \rightarrow \)

apply to arrays

Ex.: \( A = \text{true}^4 \rightarrow \text{if } c \text{ then } B[4..7] \text{ else } \text{pre}(A) \)
Homomorphic extension

Each operator or node of type

\[ \tau_1 \times \tau_2 \times \ldots \times \tau_k \rightarrow \theta_1 \times \ldots \times \theta_\ell \]

has also the type

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

Ex.: \( A \) or \( B = [A[0] \) or \( B[0], A[1] \) or \( B[1], \ldots, A[n-1] \) or \( B[n-1] ] \)
Examples

Define an array \( A \) of size 10, containing successive integers from 1 to 10.

1st solution

\[
A = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
\]
2nd solution

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

3rd solution

\[ A = [1] \mid (A[0..8] + 1^9); \]
Define a node building an array $A$ of size $n$, containing successive integers from 1 to $n$.

```
node N(const n: int) returns (A: int^n);
let
    A = [1] | (A[0..(n-2)] + 1^(n-1));
tel
```
Cumulated “or”

\[ OR(s,A) = A[0] \text{ or } A[1] \text{ or } \ldots \text{ or } A[s-1] \]
node OR(const s: int; A: bool^s) returns (OR: bool);
var X : bool^(s+1);
let
   OR = X[s];
   X = [false] \ (X[0..s-1] or A);
tel
Generalized delay

Takes a non negative integer constant $d$ and a boolean $x$ and returns the $d$th last value of $x$ (false during the first $d$ cycles).

\[
\begin{align*}
x_n & \quad x_{n-1} & \quad x_{n-d} \\
x & \quad dx
\end{align*}
\]
node DELAY(const d: int; x: bool) returns (dx: bool);
var D: bool^(d+1);
let
  dx = D[d];
  D = [x] | (false^d -> pre(D[0..d-1]));
tel
A general combinational adder

Use the one bit adder FULL ADD

node FULL_ADD(ci,a,b: bool) returns (co, s: bool);
let
  s = a xor (b xor ci);
  c0 = (a and b) or (b and ci) or (a and ci);
tel

to build an $n$-bits adder.
node ADD(const n: int; A, B: bool^n)
  returns (S: bool^n; overflow: bool);
var CARRY: bool^n;
let
  (CARRY, S) =
    FULL_ADD([false] | CARRY[0..n-2], A, B);
  overflow = CARRY[n-1];
  tel
Arrays in Lustre-V4

Restrictive, but quite satisfactory and elegant for description.

The compiler V4 first expands arrays into single variables.

- Ok for hardware and verification
- Unsatisfactory for software
  Arrays and loops in the object code needed.
Good sequential code generation

Problems:

- Code optimization: avoid useless intermediate arrays
- Computation order: difficult cases can occur.
Code optimization: avoid useless intermediate arrays

node OR
    (const s: int; A: bool^s)
    returns (OR: bool);
var X : bool^(s+1);
let
    OR = X[s];
    X = [false] | (X[0..s-1] or A);
tel

x=0;
for(i=0;i<n;i++){
    x = x | A[i];
}
OR = x;

N. Halbwachs (Verimag/CNRS)
Computation order: difficult cases can occur

\[ Y = f(X); \]
\[ X[0] = a; \]
\[ X[1..2] = g(Y[4..5]); \]
\[ X[3..5] = h(Y[0..3]); \]
Arrays in Lustre V6

New proposal integrated into SCADE6

- Restrict the use of arrays to identified operators of general-usage.
- Define efficient compilation scheme for these operators.
- Key: forbid arbitrary dependences within a single array.
Arrays in Lustre V6

- **Array types:** unchanged
- **Equations:** unchanged, but *self-dependence forbidden*
- **Access to elements:** unchanged, but *no more slices*
- **Constructors:** unchanged
- **Application of polymorphic operators** unchanged
- **Homomorphic extension:** suppressed (replaced)
Iterators

Introduce a few higher order operators, implementing the most useful patterns *(classical in functional programming)*

Iterators take as arguments a node or an operator, and a size, and define a new operator.
The “map” iterator

For any node or operator N, of sort

\[ \tau_1 \times \ldots \times \tau_k \rightarrow \theta_1 \times \ldots \times \theta_\ell \]

map\(<N,n>\) is of sort

\[ \tau_1 ^ n \times \ldots \times \tau_k ^ n \rightarrow \theta_1 ^ n \times \ldots \times \theta_\ell ^ n \]
The “map” iterator

Ex.: $A = \text{map}_{+,n}(B,C)$

means $\forall i = 0 \ldots n-1, \ A[i] = B[i] + C[i]$
The “red” iterator

For any node or operator $N$, of sort

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

$\text{red}<N,n>$ is of sort

$$\tau \times \tau_1^{\hat{n}} \times \ldots \times \tau_k^{\hat{n}} \rightarrow \tau$$
The “red” iterator

Ex.: $b = \text{red<or,n>}(\text{false, A})$

means $b = \text{false or A[0] or \ldots or A[n-1]}$
The “fill” iterator

For any node or operator N, of sort

\[ \tau \rightarrow \tau \times \theta_1 \times \ldots \times \theta_\ell \]

fill\langle N, n \rangle is of sort

\[ \tau \rightarrow \tau \times \theta_1^{\hat{n}} \times \ldots \times \theta_\ell^{\hat{n}} \]
The “fill” iterator

Ex.: Given the node

```
node MyInc(i: int) returns (j,k: int);
let j = i+1; k = i; tel
```

```
(A,x) = fill<MyInc,n>(0);
```

means $\forall i = 0 \ldots n-1, \ A[i] = i$, and $x = n$
The “mapred” iterator

For any node or operator $N$, of sort

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

$\text{map\_red}\langle N, n \rangle$ is of sort

$$\tau \times \tau_1 \hat{n} \times \ldots \times \tau_k \hat{n} \rightarrow \tau \times \theta_1 \hat{n} \times \ldots \times \theta_\ell \hat{n}$$
The “mapred” iterator

Ex.:

\[(\text{overflow}, S) = \text{map\_red}<\text{FULL\_ADD}, n>(\text{false}, A, B)\];

means

\[\forall i = 0 \ldots n-1,\]

\[(c_{i+1}, S[i]) = \text{FULL\_ADD}(c_i, A[i], B[i])\]

\[c_0 = \text{false}\]

\[\text{overflow} = c_n\]
Conclusions

Still quite powerful
Easy to produce good code
Further optimizations [Morel01 ...]

\[ \text{e.g. } \text{map}<N,n>(\text{map}<M,n>(X)) = \text{map}<N \circ M, n>(X) \]
Conclusions

A step towards a higher order language
   (Synchronous Lucid [Pouzet], Lava [Sheeran])
Tremendous reduction of code size in real applications
   (Airbus)
Implementation in Lustre V6, and (more or less) in Scade6