A Lustre Primer for Kind 2 Users

Rob Lorch and Daniel Larraz and Cesare Tinelli

May 2024

Contents

1 Basic Concepts

1.1 Lustre Nodes

Lustre is a synchronous data-flow language for modeling and implementing reactive systems. It can be seen indifferently as a declarative parallel programming language or as an executable specification language. The most basic unit of computation in a Lustre program, or model, is a **node**, which is just a stream transformer: it takes streams of input and produces streams of output. Operationally, a node reads its input and generates its output incrementally in discrete *timesteps*, or cycles, determined by an abstract global clock. At each cycle, all output values are assumed to be computed instantaneously from the current input and state values. By default, all nodes in a model compute synchronously and in parallel according to the global clock.

A **stream** is an infinite sequence of values, all of the same (given) type. Hence, a Lustre node can be viewed as modeling an infinite sequence of discrete timesteps, where at each timestep, each node variable takes its next value.

Below, the node Combine takes as input two integer streams x and y, and produces integer stream z as output. If we consider $x = (x_0, x_1, \ldots)$ and $y = (y_0, y_1, \ldots)$, then Combine produces output $z = (x_0 + 2 \cdot y_0, x_1 + 2 \cdot y_1, \ldots)$ (or more concisely, $z_n = x_n + 2 \cdot y_n$ at each timestep n).[1] Notice that line 3 is an equation between streams of integers. The operators =, + and * are stream operators obtained by lifting to streams the corresponding operators over integers. The same is true of concrete constants in Lustre, such as 2 in line 3 below, which are streams with the same value at each time step. Lustre respects typical rules of operator precedence, so x + 2*y will be parsed as x + (2*y) rather than (x + 2)*y.

Listing 1: Simple Lustre node

```
1 node Combine(x: int; y: int) returns (z: int);
2 let
3   z = x + 2*y;
4 tel
```

Line 1 of Combine is referred to as the **node interface**, where the node's inputs and outputs, and their types, are declared.

The code block surrounded by let and tel denotes the **node implementation** (or **node body**), where the node's outputs are defined in terms of the node's inputs. A node implementation is comprised of a set of equations of the form <var> = <expr>, where <var> is an output variable or a local variable (see below) and <expr> is an expression in terms of any of the variables that are in scope.

Nodes can have more than one output stream as exemplified by the node TwoOuts below.

Listing 2: Node with two outputs

```
node TwoOuts(x: int) returns (double: int; square: int);
let
  double = x + x;
  square = x * x;
tel
```

[1]Note that it is not possible to specify a stream pointwise in Lustre, so when we write $x = (1, 2, 3, \ldots)$, say, we are writing a mathematical statement about stream x, not an equation in Lustre.

Another optional component that can be added to a Lustre node are **local declarations**. The local variables and constants declared in this section can be used in the node implementation, but they are not exposed in the node interface.

Finally, **global constants** can be declared outside of the node body, and are visible within every node.

Below is another version of `Combine`, where the value 2 is stored in a global constant `C` and the local variable `l` is used to store an intermediate computation.

Listing 3: Node with global constant and local variable

```
const C: int = 2;
node Combine(x: int; y: int) returns (z: int);
var l: int;
let
  l = C*y;
  z = x + l;
tel
```

The order of the equations in the body of a node is immaterial. However, the definition of a variable provided by the equations cannot be *circular*, as we explain in Section 5.

In Lustre, identifiers (for constants, variables, types, and keywords) are delimited by whitespace characters, separators such parentheses and semicolon, and other symbols such as $+$, * and so on, as in most programming languages. Whitespace is, however, not semantically meaningful. For instance, indentation does not change the parsing of an expression.

1.2 Node analyses

Lustre was designed to be a programming language. Well-formed Lustre nodes are executable in the sense that they can be compiled to executable programs computing their output values incrementally from their input values and internal state.

Here, we are mostly interested in *analyzing* Lustre programs and their possible behavior with a tool like Kind 2.

A basic form of analysis that can be applied to a Lustre program is **node simulation**. During simulation, the user specifies a number n of timesteps to simulate, as well as the first n values of each input variable. Given this information, the first n values of each output variable are computed. For the `Combine` node above, if the user performed simulation with $n = 3$ and with given input stream prefixes $x = (1, 2, 3)$ and $y = (4, 5, 6)$, the output value $z = (9, 12, 15)$ would be computed.

Another form of analysis is **property checking**, where the user specifies a property in the node body (in the form of a Boolean expression) to be proven or disproven **invariant**, that is, true at every time step. For example, the conditional property `y > 0 => l > y` in the node below would be proven invariant. In contrast, the property `z > 0` would be disproven in the `Combine node`, as `z` is negative in timesteps where both `x` and `y` are negative.

Listing 4: Node with internal property checking

```
const C: int = 2;
(* Example with
   two properties
*)
node Combine(x: int; y: int) returns (z: int);
```

```
var l: int;
let
  l = C*y;
  z = x + l;

  check y > 0 => l > y; -- invariant
  check z > 0;          -- not invariant
tel
```

Property checking is performed by model checkers such as Kind 2, so further details are outside the scope of this document.

2 Comments

Listing 4 shows two ways to add comments in Lustre programs. Single line comments are introduced by the character sequence --. Multiline comments are delimited by the sequences (* and *). Nested multiline comments are not allowed.

3 Primitive Types

Lustre's primitive types are bool, int, and real. Informally, we say that bool is the type of Boolean values (true, false). Strictly speaking, bool is the type of *streams* of Boolean values. We identify the two for brevity since there is no possibility of confusions as all values in Lustre are streams.[2] The same is true for the other types.

In the **idealized** semantics of Lustre, int is the type of mathematical (infinite precision) integers, and real is the type of real numbers. Lustre compilers approximate that semantics by using machine integers for int and floating point numbers for real. In contrast, Kind 2 is faithful to the idealized semantics.

Lustre supports the Boolean operators not, and, or, xor, and => (implies), as well as the arithmetic operators +, - (both unary and binary), *, /, mod, and div (integer division), all with the expected arity and (pointwise) semantics. The arithmetic operators (+ and so on) are overloaded as they apply both to int and real terms. The binary operators, however, are applicable only to arguments of the same type (both int or both real). Numerals (0, 1, ...) have type int while decimals (e.g., 0.0, 31.97) have type real.

Additionally, Lustre supports if-then-else expressions with the syntax

```
if <expr_0> then <expr_1> else <expr_2>
```

where <expr_0> has type bool and <expr_1> and <expr_2> must have the same type.

4 Temporal Operators

Lustre contains two temporal operators: the binary operator -> (pronounced "arrow" and not to be confused with =>) and the unary operator pre.

[2]It is not possible to refer directly to the scalar values in a stream in Lustre. Even constants, such as true, 2, 3.6 denote streams of values, not individual values.

	0	1	2	...	n
`1`	1	1	1	...	1
`x`	x_0	x_1	x_2	...	x_n
`pre x`	?	x_0	x_1	...	x_{n-1}
`1 + pre x`	$1+?$	$1+x_0$	$1+x_1$	...	$1+x_{n-1}$
`1 -> (1 + pre x)`	1	$1+x_0$	$1+x_1$	...	$1+x_{n-1}$

Table 1: Stream computations for expression `1 -> (1 + pre x)` at times $0, \ldots, n$.

The arrow operator is an *initialization* operator, where the expression `a -> b` denotes the stream whose first value is equal to the first value of stream `a`, and whose nth value is equal to the nth value of stream `b` for every $n > 0$. For example, if `a` $= (-1, -1, -1, \ldots)$ and `b` $= (1, 2, 3, \ldots)$, then `a -> b` $= (-1, 2, 3, \ldots)$.

The `pre` operator can be viewed as referencing the previous value at every timestep—the expression `pre a` denotes the stream whose value at step n is equal to the value of stream `a` at step $n-1$. For example, if `b` $= (1, 2, 3, \ldots)$, then `pre b` $= (?, 1, 2, \ldots)$. Notice that with these semantics, `pre b` is undefined in the initial timestep (denoted by the question mark here).

Kind 2, treats undefined expressions as **underspecified**. That is, when simulating the stream `pre b`, it could take values `(-23, 1, 2, ...)`, `(79, 1, 2, ...)`, etc. In other words, Kind 2 assigns the first element of `pre b` an arbitrary integer. Consistently with that, a property of a node containing `pre`'s is considered invariant only if it holds at every step, regardless of the value assigned to the first element of any stream resulting from a `pre` application.

Because `pre` creates underspecified streams, we can combine it with `->` to obtain fully specified streams. For example, if `b` $= (1, 2, 3, \ldots)$, then `0 -> pre b` $= (0, 1, 2, 3, \ldots)$, where the arrow operator supplies the initial value 0 for the resulting stream. If an application of `pre` occurs without a corresponding application of `->`, the `pre` is **unguarded**. While unguarded `pre`s are allowed in Lustre, Kind 2 will produce warnings for nodes that contain them as this is usually an oversight by the user and may lead to unexpected results.

The `pre` operator has the same precedence as other unary operators such as `not`. For example, `pre x + y` is read as `(pre x) + y`, not as `pre (x + y)`. Note that `pre` distributes over all non-temporal operators. For instance, the expression `pre (x + y)` is equivalent to `pre x + pre y`.

To further reinforce how operators work over streams, the computation of the expression `1 -> (1 + pre x)` is illustrated in Table 1.

Using temporal operators, we can define a `Counter` node as follows.

Listing 5: Node with temporal operators

```
node Counter(init: int) returns (c: int);
let
  c = init -> pre c + 1;
tel
```

In `Counter`, the output stream `out` is initialized to the input initialization value `init`, and it is incremented at every timestep. Notice that `out` is recursively defined—the $n+1$st value of `out` is equal to the nth value of `out` plus 1, except in the *base case* of initialization.

	0	1	2	3	$\ldots$
`1`	1	1	1	1	$\ldots$
`2`	2	2	2	2	$\ldots$
`3`	3	3	3	3	$\ldots$
`1 -> 2`	1	2	2	2	$\ldots$
`2 -> 3`	2	3	3	3	$\ldots$
`pre (2 -> 3)`	$?$	2	3	3	$\ldots$
`1 -> (2 -> 3)`	1	3	3	3	$\ldots$
`(1 -> 2) -> 3`	1	3	3	3	$\ldots$
`1 -> pre (2 -> 3)`	1	2	3	3	$\ldots$

Table 2: Stream computations for expression `1 -> pre (2 -> 3)` at times $0, 1, \ldots$

The `pre` and `->` operators provide a declarative and mathematically elegant way to define **stateful** computations. An alternative, operational way to understand the functionality of node `Counter` is that `init` is an input variable and `c` is a *state* variable. Initially, the value of `c` is that of `init`. At each successive iteration, the new value of `c` is its old value (denoted as `pre c`) plus one.

A deceptively difficult example is defining in Lustre a stream with value $(1, 2, 3, 3, 3, \ldots)$, with infinite repetitions of 3 from the third step on. A first guess might be the term `1 -> (2 -> 3)` or perhaps the term `(1 -> 2) -> 3`. However, both of these streams will omit the value 2, as they take the initial value from the first argument of the outer arrow (which is `1` in both cases) and the non-initial values from the second argument of the outer arrow (which is a stream of `3`s in both cases). A key insight is that the `pre` operator can also be viewed as a *right-shift operator* on streams. From this, the correct answer is `1 -> pre (2 -> 3)`, which takes the initial value 1 and the remaining values from the stream $(?, 2, 3, 3, 3, \ldots)$.

Table 2 helps illustrate the difference between the various expressions above.

A node that generates the stream $(1, 2, 3, 3, 3, \ldots)$ from no inputs can then be defined as follows.

Listing 6: Tricky output stream example

```
node N() returns(y: int);
let
   -- defining output stream (1, 2, 3, 3, 3, ...)
   y = 1 -> pre (2 -> 3);
tel
```

Another deceptively difficult example is the following Lustre node which outputs the stream of all Fibonacci numbers in increasing order. Because `Fib` is defined in terms of the two previous Fibonacci values, the first *two* steps need to be initialized. The example is tricky and may require some thought for those new to Lustre.

Listing 7: Fibonacci numbers

```
node Fibonacci() returns(Fib: int);
let
  Fib = 1 -> pre (1 -> Fib + pre Fib);
tel
```

The example can be perhaps easier to see by introducing local names for the subexpressions on the equation's right-hand side.

Listing 8: Fibonacci numbers (alternative approach)

```
node Fibonacci() returns(Fib: int);
  var preFib: int;
  var prepreFib: int;
let
  preFib = 0 -> pre Fib;
  prepreFib = 1 -> pre preFib;
  Fib = preFib + prepreFib;
tel
```

5 Declarative Semantics

Lustre has a **declarative** semantics, meaning that the order of equations in node bodies does not matter. Because of this, node equations should not be viewed imperatively as assignments; instead, a node body is a set of stream constraints of the form `<var> = <expr>`.

To illustrate this concept, consider the following `Factorial` node which outputs a stream of factorial numbers (the nth value of the stream is $n!$). When defining output stream `F`, we can reference the helper stream `N` before it is defined.

Listing 9: Factorial node

```
node Factorial() returns (F: int);
var N: int;
let
  -- all the factorial numbers
  F = 1 ->  N * (pre F);
  -- all the natural numbers
  N = 0 -> (pre N) + 1;
tel
```

Even though Lustre has a declarative semantics and allows recursive definitions, circular definitions are rejected. For example, the following node is invalid Lustre because the nth value of `out1` is defined in terms of the nth value of `out2`, and the nth value of `out2` is defined in terms of the nth value of `out1`.

Listing 10: Node with circular dependencies

```
node Circular() returns (out1, out2: int);
let
  out1 = out2 + 1;
  out2 = out1 - 1;
tel
```

In fact, there are no values for the streams `out1` and `out2` that satisfy both equations. However, even if it is possible to satisfy all equations, as in the following example, any node with a circular dependence is conservatively rejected.

Listing 11: Another node with circular dependencies

```
node Circular() returns (out1, out2: int);
let
  out1 = out2;
  out2 = out1;
tel
```

Note that there is no circularity in the definition of output N of node Factorial since N is defined in terms of pre N, and not in terms of N itself.

6 Composite Types

In addition to the primitive types, Lustre supports records and arrays.

6.1 Records

Record types have the syntax

```
struct { <field_1> : <type_1>; ...; <field_n> : <type_n> }
```

They must be named and declared with a global **type declaration** of the form

```
type <ty_name> = <type>;
```

Record values can be constructed with the syntax

```
<ty_name> { <field_1> = <expr_1>; ... <field_n> = <expr_n> }
```

and destructed with the syntax

```
<record_term>.<field>
```

as seen in the next example.

Listing 12: Record construction and destruction

```
type sensorData = struct { speed: real; height: real; direction: int };

node AdjustSensorData(in: sensorData) returns (out: sensorData);
  var h: real;
let
  h = if in.height < 0.0 then 0.0 else in.height;
  out = sensorData { speed = in.speed;
                     height = h;
                     direction = in.direction };
```

6.2 Arrays

Array types have the syntax

```
<element_type>^<numeral>
```

Values of an array type can be constructed in two different ways. Lustre supports the **array literal** syntax of the form

```
[<element_1>, ..., <element_n>]
```

as well as the (constant) **array constructor** syntax of the form

```
<element>^<length>
```

Array elements can be accessed with the standard **array access** syntax <array_var>[<index>], with zero-based indexing.

Listing 13: Array construction

```
node ThreeArrays() returns (out1: bool^5; out2: int^4);
let
  out1 = [true, true, false, true, false];
  out2 = 1^4;   -- equivalent to out2 = [1, 1, 1, 1]
tel
```

Listing 14: Array access

```
node Fst(in: int^10, k: int) returns (out: int);
let
  out = if 0 < k < 10 then in[k] else in[0];
tel
```

7 Composition

A Lustre model can be hierarchically defined by defining nodes in terms of other nodes through the use of **node applications**. Revisiting the `Counter` node, we can use node applications to instantiate two distinct counter streams. In the following example, the output streams `ctr1` and `ctr2` of node `Top` are defined using expressions that contain node applications. More specifically, output variable `ctr1` is defined as the stream output by node `Counter` when applied to input `0`, and the output variable `ctr2` is defined as the stream output by node `Counter` when applied to input `5`. Output `P1` is a boolean stream representing the property that `ctr2` is greater than `ctr1`.

Note that nodes can have no inputs (as node `Top` below) or no outputs.

Listing 15: Lustre program with node applications

```
node Top() returns (ctr1, ctr2: int; P1: bool);
let
  ctr1 = Counter(0) + 3;
  ctr2 = Counter(5);
  P1 = (ctr2 > ctr1);
tel

node Counter(init: int) returns (out: int);
let
  out = init -> pre out + 1;
tel
```

Node applications must respect the expected type checking rules: each argument of the application of a node N, which can be any stream-denoting expressions, must have a type that matches the type of the corresponding input parameter in N's interface. Similarly, the return type of N must be a valid type for the expression that contains the node application. For example, the return type of `Counter` matches the expected type for the first argument of the + operator in the expression `Counter(0) + 3`.

Note that the definition of node `Top` includes an application of node `Counter`, even though `Top` is defined before `Counter`. Similarly to equations in a node body, the order of node definitions in Lustre model is immaterial. However, the application graph cannot contain cycles. In other words, a node cannot be defined, directly or indirectly (through subnodes), in terms of itself.

In general, an application of a node with a single output stream of some type T can occur anywhere an expression of type T can occur on the right-hand side of an equation in a node's body. In contrast, an application of a node with multiple outputs can occur only in an equation of the form

```
(<var_1>, ..., <var_n>) = <node_name>(<arg_1>, ..., <arg_m>);
```

or

```
<var_1>, ..., <var_n> = <node_name>(<arg_1>, ..., <arg_m>);
```

where <var_1>, ..., <var_n> are local or output variables of the node containing the application, with types matching the types of the outputs of the applied node <node_name>, in the same order as in that node's interface.

Listing 16: Applications of nodes with multiple outputs

```
node Top(x: int) returns (P1: bool);
  var positive: bool;
  var nonnegative: bool;
let
  (positive, nonnegative) = N(x);
  P1 = positive => nonnegative;
tel

node N(x: int) returns (positive, nonnegative: bool);
let
  positive = (x > 0);
  nonnegative = (x >= 0);
tel
```

8 Common Auxiliary Nodes

While the temporal operators -> and pre may not seem very powerful, they can be used to define auxiliary temporal operators, presented below.

Listing 17: Common auxiliary nodes

```
-- Y is true iff X has been true so far
node Sofar ( X : bool ) returns ( Y : bool ) ;
let
 Y = X -> (X and (pre Y)) ;
tel

-- Z is true iff X has been true at some point in the past,
-- and Y has been true since then.
node Since ( X, Y : bool ) returns ( Z : bool ) ;
let
  Z =  X or (Y and (false -> pre Z)) ;
tel

-- Y is true iff X was true in the initial timestep
node Initially(X: bool) returns (Y: bool)
let
  Y = X -> pre Y;
tel
```

```

-- Y is true iff X has been true at least once
node Once(X : bool) returns (Y : bool);
let
  Y = (false -> pre Y) or X;
tel
```

9 More Examples

For more examples, see the Kind 2 web application at: `https://kind.cs.uiowa.edu/app/`. Note that these examples contain some language features that are extensions to Lustre (for example, contracts) that are not covered in this document. For more information on Kind 2 and its extensions to Lustre, please check its documentation at `https://kind.cs.uiowa.edu`.