Inlining in a Verified Synchronous Language Compiler

Balthazar Patiachvili

March — August 2025

Supervised by Timothy Bourke in the Parkas team at INRIA Paris

General context

Vélus [BBP19; Bou+17; Bou+21; BPP23] is a verified compiler from a synchronous dataflow programming language based on Lustre [Hal+91] and SCADE, to Clight, a large subset of the C99 programming language. It specifies the dynamic semantics and type systems of the input language, and relies on CompCert [Ler09], a verified compiler, for the specification of the output language and the compilation from Clight to assembly code. It is implemented in the Rocq Prover [ICc24], a proof assistant in which the compiler and the proof of its correctness are written. Each function is proved correct with regards to the formal models of the input language, of Clight, and of all the intermediate languages.

This internship follows the initial work done by Timothy Bourke, Lélio Brun, Pierre-Évariste Dagand, Xavier Leroy, Marc Pouzet, and Lionel Rieg [Bou+17], and the theses of Lélio Brun [Bru20], which set up the basis of Vélus with modular reset, Basile Pesin [Pes23], which extends the input language with new operators and state machines, and Paul Jeanmaire [Jea24], which adds a denotational semantics to the compiler.

Research problem

This internship studies node inlining, an optimization similar to function inlining in imperative languages. The idea is to replace a call to another node with its body to avoid function calls and data manipulation that may increase the execution time of the generated imperative code. This would add a phase to the compilation flow of Vélus, which will require to implement an algorithm of node inlining in Rocq, and more importantely to prove that this phase preserves all the properties (synctactic, typing, semantic, ...) required to maintain the complete proof of correctness of the compiler.

Adding inlining to a verified compiler has already been done several times, notably in CompCert[1] and in PureCake [KKM24]. However, this problem is new for synchronous dataflow languages, in particular due to the synchronous semantics of the language which is not a from step-based imperative semantics, and the modular reset that must be distributed to inlined equations. The standard compilation scheme [Bie+08] implemented in the Vélus compiler transforms each node instance into a single function call, and thus inlining is required for certain feedback patterns (node instances where some outputs only depends on some inputs).

Another question is the inlining heuristic, i.e., which nodes should be inlined by the compiler. This would be more beneficial than using exclusively the one of CompCert because a typical Lustre/SCADE program contains many small nodes.

Your contribution

During my internship, I implemented node inlining in Rocq and included it in the compilation flow. I proved that syntactic and typing properties are preserved by this new compilation phase. No real difficulty was encountered during the proofs, but they gave me a better understanding of the full compiler.

[1] https://compcert.org/doc/html/compcert.backend.Inlining.html

However, existing proof schemes were not sufficient to directly ensure the required properties: node inlining is the only compilation phase where the compilation of a node depends on the definition of other nodes, and not just their input/output interfaces.

The proof of correctness, ensuring the preservation of the semantics, provided the major problem of the internship. It took several weeks to understand the issue in detail, to clearly define the central property and prove it. The result is a complete proof of an intermediate lemma which states an interesting and crucial semantics property for node inlining. The full proof of correctness was not finished because of a lack of time.

Arguments supporting its validity

All the proofs have been written in Rocq, ensuring the logical consistency of the stated properties. Some powerful tools of logic had to be used, in particular the axiom of choice, but all of them were already in use in the compiler so no new axiom or tool had to be introduced in the project. Moreover, as the compilation phase that has been introduced is located in the middle of the full compilation flow, the only assumption is that the result of the previous phase verifies the required properties.

Summary and future work

During this internship, I implemented node inlining in Vélus, and proved many of the properties required to ensure compiler correctness. Unexpected and interesting properties about the modular reset have been proved, which are valuable independently of our focus on node inlining. Future work on Vélus should complete the proof of correctness of this new compilation pass, and improve the dependency analysis to be able to compile valid programs that are currently rejected.

Furthermore, the compiler is increasingly difficult to work with because of its size. Simplifying the backend is necessary to be able to add more features to Vélus in the future. In particular, functional arrays are the next big addition planned in Vélus, and will be the focus of a future PhD. This would need changes in every part of the Vélus architecture, so it may be beneficial to make backend changes at the same time.

The official repository of Vélus is available on the GitLab instance of the Inria at https://gitlab.inria.fr/velus/velus, and on its GitHub mirror at https://github.com/INRIA/velus. The work done during my internship is currently only available on the branch `balthazar-intership` on GitLab.

To simplify the references to the source files of Vélus, only the name of the file will be given: the full path from the repository source is available in Appendix A.

Contents

```lustre
node nand(a, b: bool)
returns (z: bool);
var c: bool;
let c = a and b;
    z = not c;
tel
```

a	F	F	T	T	...
b	F	T	F	T	...
c	F	F	F	T	...
z	T	T	T	F	...

Figure 1: Example of a Lustre node (left) and its chronogram (right)

1 — Context

1.1 — Lustre/SCADE

Lustre [Hal+91] is a formally defined synchronous dataflow language for programming reactive systems. Introduced in the 1980s, it progressed to pratical and industrial use through SCADE 3, an industrial environment developed by Ansys. The development continued with SCADE 6 [CPP17], another version of the environment with new features inspired by Lucid Synchrone [CP99]. It is now used for critical control software in aircraft, helicopters and nuclear power plants thanks to its DO-178B certification.

A Lustre program is a series of node, type and external function declarations. A type in Lustre can be either an `int`, `float`, or C-style enumeration. For example, `bool` is defined in Lustre as:

```lustre
type bool = true | false
```

Moreover, external function are built-in functions, like sinus and cosinus. Finally, nodes are sets of equations, that will be compiled into imperative functions. Figure 1 shows an example of a node, where `nand` is the name of the node, `a` and `b` are input variables, `z` is an output variable, and `c` a local variable. The lines between `let` and `tel` are the *equations*, and their order does not matter: swapping them does not change the result.

Execution traces of synchronous dataflow programs can be represented by infinite streams in tables called *chronograms*. Indeed, all variables are updated simulaneously, with respect to the list of equations of the node, it thus makes sense to represent them in a grid where each column is an *instant*.

This representation as streams is at the core of Lustre's semantics, and facilitates the definition of some operations that will be defined in Section 1.3.2. Formally, streams are defined coinductively as $\langle v \rangle \cdot vs$ for the concatenation of the value v with the stream vs.

1.2 — Vélus

Vélus is a verified compiler written in Rocq, from a synchronous dataflow language, inspired by Lustre and SCADE, to Clight, a large subset of C99. The input language is also called Lustre even though it is not exactly the one described formally in [JRH20]: it is a superset of Lustre and a subset of SCADE 6. This compiler is implemented as a composition of functions that rewrites the program into successive intermediate languages, each with its own semantic model. Each function is then proved correct with respect to the semantics of those languages.

The (almost) complete architecture of Vélus can be seen in Figure 2. The general layout was described in [BBP19], after an adaptation of the compilation scheme presented in [Bie+08]. The "node inlining" pass will be described later in this document, and has been fully implemented during this internship.

Lustre contains many operators and constructions, but only the ones used in this internship will be described here. You can find the full syntax of Lustre in Appendix B.

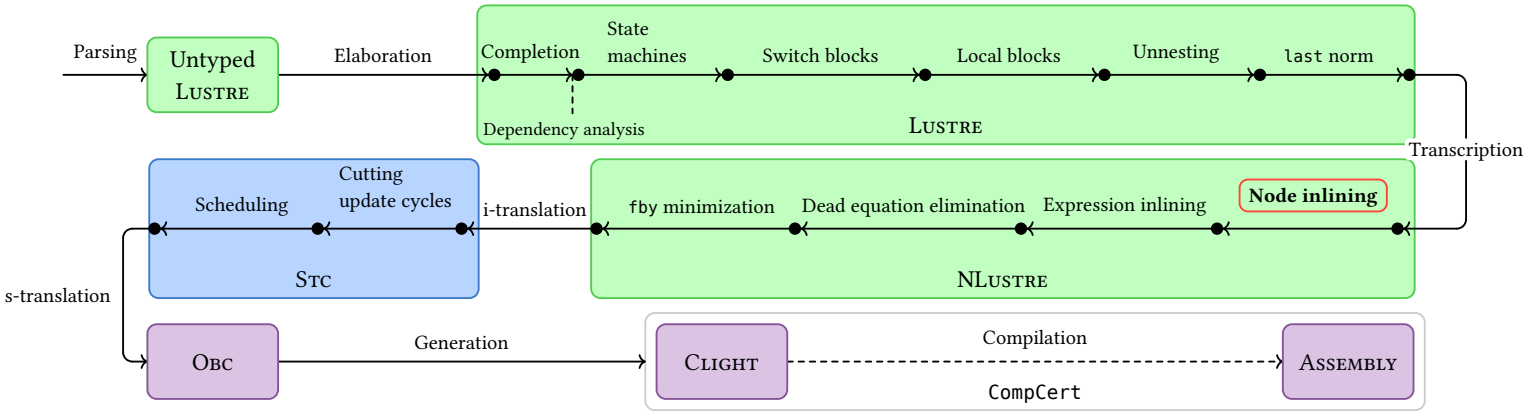

Figure 2: Architecture of Vélus

1.3 — NLustre

NLustre is a intermediate language used in Vélus. It means "Normalised Lustre", and is transcripted from Lustre after some rewriting passes (see Figure 2). Its syntax is a restricted subset of Lustre and the node inlining is a compilation pass in this language, which is why it will be described in detail in the following paragraphs.

1.3.1 — Syntax

Here is the complete syntax of NLustre expressions [Pes23]:

Simple expressions

$e ::= c$ (constants)
$\mid C$ (enumeration value)
$\mid x$ (variable)
$\mid$ `last` x (last operator)
$\mid \diamond\, e$ (unary operator)
$\mid e \oplus e$ (binary operator)
$\mid e$ `when` $C(x)$ (sub-sampling)

Control expressions

$ce ::= e$ (simple expression)
$\mid$ `merge` $x\ (C \Rightarrow ce)^+$ (over-sampling)
$\mid$ `case` e `of` $(C \Rightarrow ce)^+$ (pattern-matching)

Right hand-side

$rhs ::= ce$ (control expression)
$\mid f(\text{args}^+)$ (extern function call)

The constants, enumeration values, variables, pattern matching, unary and binary operators are the usual ones that can also be found in C. The `last` operator will be explained in detail in Section 1.3.3. Finally, `when` and `merge`, respectively operators of sub-sampling and over-sampling, allow to modifiy the "rhythm" of an expression, formally defined as a *clock* (see Section 1.3.2).

1.3.2 — Clocks

Vélus streams are sampled, meaning that some may be produced at a slower rate than others. Moreover, they are synchronous: a stream with a slower rate will balance by adding a missing value, represented by an absence denoted $\langle\rangle$. The clock of a stream xs is then a boolean stream defined as follows:

$$\textbf{clock-of}(\langle\rangle \cdot vs) := \mathrm{F} \cdot \textbf{clock-of}(vs)$$
$$\textbf{clock-of}(\langle v\rangle \cdot vs) := \mathrm{T} \cdot \textbf{clock-of}(vs)$$

The *base* clock of a node is defined as the clock whose value is T if and only if at least one of the inputs is present. In particular, the base clock of the main node is the clock whose value is T at each instant.

In Figure 3, the variable `y` has clock "• `on true(b)`" and `z` has clock "• `on false(b)`" due to the subsampling induced by the `when` operator. The operator `merge` then recompose the two streams to define the variable `t` with clock •. More generally, sub-sampling a stream with respect to another clock

```lustre
node foo(x: int, b: bool)
returns (t: int);
var y: int :: . on true(b);
    z: int :: . on false(b);
let
  y = x when true(b);
  z = (x * 2) when false(b);
  t = merge b
    (true => y) (false => x);
tel
```

b	T	T	F	F	T	F	T	F	F	...
x	1	2	3	4	5	6	7	8	9	...
y	1	2			5		7			...
z			6	8		12		16	18	...
t	1	2	6	8	5	12	7	16	18	...

Figure 3: Example trace of `when` and `merge`

makes a new stream based on the first one where a value is replaced by an absence when the other stream is F. Merging two streams is the reverse operation: two streams where the present values do not overlap can be merged regarding to a clock (take the value in the first stream if T, and the second otherwise). A clock represents an abstraction of the presence/absence status of a stream, and can then be defined inductively:

$ck ::= \bullet$ (Cbase) $\mid ck$ on $C(x)$ (Con), with x a variable and C an enumeration value

1.3.3 — Equations

Here is the syntax for equations in NLustre:

Equations

$eq ::= x = rhs$ `(Def)` $\mid x =$ `reset` c `fby` e `every` x^* `(Fby)`

$\mid$ `last` $x = c$ `every` x^* `(Last)` $\mid x^+ = (\text{reset } f \text{ every } x^*)(e^+)$ `(App)`

`Def` equations are the most basic equations, and they are stateless. Fby, Last and App are stateful and can be reset (see Section 1.3.4).

`Fby` equations use the `fby` binary operator, which can be defined as the concatenation of the first value of the first stream with that of the second stream. Figure 4 shows an accumulator built with this operator.

`Last` equations initialize *shared variables*. A shared variable is a variable from which it is possible to retrieve the current value or its previous value through the `last` operator. To ensure that the result is always defined, the "initial last" value must be given, through `Last` equations. In practice, all `Last` equations could be expressed as `Fby` equations, and reciprocally, but it has been decided to keep both to better optimize the generated code.

`App` equations are calls to another node, and are the most important type of equation for node inlining. Figure 6 an example of such a call.

```lustre
node acc(inc: int)
returns (prev: int; sum: int);
let prev = 0 fby inc;
    sum = 0 fby (sum + inc);
tel
```

inc	1	2	3	4	5	6	7	...
prev	0	1	2	3	4	5	6	...
sum	1	3	6	10	15	21	28	...

Figure 4: Simple accumulator in NLustre

```
1 node acc(inc: int, rst: bool)                    lustre
2 returns (sum: int);
3 let
4   sum = reset (0 fby (sum + inc)) every rst;
5 tel
```

inc	1	3	5	2	9	2	4	...
rst	F	F	F	T	F	F	F	...
sum	0	1	4	0	2	11	13	...

Figure 5: Accumulator with reset in NLustre

1.3.4 — Reset signals

The equations using `last`, `fby` operators, and node calls shown in Section 1.3.3 give the intuition that an equation "stores" actively the current (and previous if needed) state of its variables. Thus, it may be interesting in some cases to reset the equation to enforce the value of the variables as if it were again the first instant of the equation. In NLustre, the indication to reset is given to an equation through *reset signals*, which are a list of boolean variables: if at least one of them evaluate to T, then the equation is reset. For clarity, the syntactic (list of boolean variables) and the semantic (boolean stream) representation of reset signals will be denoted by the same variables.

Figure 5 shows an example of a simple counter reset each time the variable `rst` is set to T (columns highlighted in gray). Moreover, in Lustre, where reset signals with more complex expressions are allowed, one can write an example as Figure 6. This shows the *modular* property of the `reset`, introduced in [HP00] as *local reset*, since the additional rule can be added or removed without modifying any other equation [BBP18]. This allows to easily reset expressions without passing a boolean stream through intermediate node calls.

The semantics of `reset` can be formally defined through "masking". The idea has been introduced in [BBP19] and developed in [BPP23], where $\mathbf{mask}^k_{k'}\ rs\ xs$ is defined formally in Figure 7.

In this definition, the boolean stream rs corresponds to the stream of the `reset` condition. An *instance* of the value stream xs consists of the section of the stream between two T's on the reset stream. The instances are numbered: the instance k is located between the kth (included) and $(k+1)$th (excluded) occurences of T in the reset stream, or before the first occurence of T if $n = 0$.

```
1 node acc(inc: int) returns (sum: int);           lustre
2 let sum = 0 fby (sum + inc); tel
3
4 node foo(in: int) returns (out: int);
5 var (count: int);
6 let
7   count = 0 fby (count + 1);
8   reset out = acc(in) every (count mod 3 = 0);
9 tel
```

in	1	3	5	2	9	2	4	6	...
out	0	1	4	0	2	11	0	4	...

Figure 6: Accumulator automatically reset in Lustre

$$\mathbf{mask}^k_{k'}(\mathrm{F}\cdot rs)(v\cdot xs) := (\text{if } k' = k \text{ then } v \text{ else } \langle\rangle)\cdot \mathbf{mask}^k_{k'}\ rs\ xs$$

$$\mathbf{mask}^k_{k'}(\mathrm{T}\cdot rs)(v\cdot xs) := (\text{if } k'+1 = k \text{ then } v \text{ else } \langle\rangle)\cdot \mathbf{mask}^k_{k'+1}\ rs\ xs$$

$$\mathbf{mask}^k := \mathbf{mask}^k_0$$

Figure 7: Definition of **mask**

Figure 8 shows an example of the spliting of the value stream xs into instances cut by occurrences of T in the reset stream rs. The resulting stream with reset is obtained after merging all the masked streams. Figure 9 shows the sampling with masks for Figure 5.

xs	0	1	2	3	4	5	6	7	8	9	...
rs	F	F	T	F	F	F	T	F	T	F	...
mask^0	0	1									...
mask^1			2	3	4	5					...
mask^2							6	7			...
mask^3									8	9	...

Figure 8: Example of **mask**

inc	1	3	5	7	9	2	4	...
rst	F	F	T	F	F	T	F	...
mask^0 **rst sum**	1	4						...
mask^1 **rst sum**			5	12	21			...
mask^2 **rst sum**						2	6	...
sum	1	4	5	12	21	2	6	...

Figure 9: Sampling with masks for Figure 5

1.4 — Variable names

In Vélus, the variables defined in a source program can have any standard character in their name, except "`$`". Indeed, this character is reserved for the generation of new variables during compilation. For example, during the `last` normalization phase (see Figure 2), all the generated variables are prefixed by "`last$`". This helps keep track of the source of the new variables throughout compilation. Moreover, a variable is called *atomic* if it does not contain "`$`", and is prefixed by a reserved keyword otherwise. The list of such keywords can be found in `Ident.v`.

2 — Node inlining

From here, everything but definitions have been introduced, i.e., all the mentioned lemmas, and the function `lift_reset` (see Section 3.3.3) have been added or proved during my internship.

2.1 — Description

Node inlining is the equivalent of inline expansion in languages with functions: it is a manual or compiler optimization that replaces a node call with the body of the called node. In Vélus, two benefits can be found in this optimization: as in other languages, it may reduce the worst-case execution time, and it will increase the number of programs that can be compiled by Vélus.

Indeed, the scheduling pass (see Figure 2) orders all the equations in a node so that each only depends on variables defined earlier. However, not all valid programs can be scheduled in this way, because all the output of a node instance are assumed to depend on all the inputs, even when this is not the case within the node definition.

In the left part of Figure 10, the variable `z` is used on the left and right sides of the same equation (highlighted in red), leading to an error because this variable is used in its definition without a delay (`last` or `fby`). However, the variable `xi` in node `f` is never used, so this program could be compiled with node inlining. The right part of Figure 10 shows a semantically equivalent program after the new pass. It is now well-scheduled. The algorithm to obtain this program will be described in Section 2.2. The main goal of node inlining is to fix such cases (this example is very simple, see Appendix C for a more interesting example with the same issues).

2.2 — Implementation

The code fragments are simplified, compared to the implementation in Rocq, to avoid having to explain non-interesting parts. The real implementations can be found in the files mentionned in Appendix A.

```
1  node f(wi, xi : int)                    lustre
2  returns (wo, xo : int);
3  let wo = wi + 1;
4      xo = wi * 4;
5  tel
6
7  node g(x : int)
8  returns (y : int);
9  var z : int;
10 let (y, z) = f(x + 2, z + 3);
11 tel
```

```
1  node g(x : int)                         lustre
2  returns (y : int);
3  var z : int;
4  let
5    wo' = wi' + 1;
6    xo' = wi' * 4;
7    wi' = x + 2;
8    xi' = z + 3;
9    y = xo';
10   z = wo';
11 tel
```

Figure 10: Example of an ill-scheduled program before (left) and after (right) inlining

2.2.1 — Location

The first question was to choose where to place the new compilation pass. There are two possibilities: in Lustre, between the compilation of switch blocks and local blocks to exploit the possibility of defining local blocks in the syntax of Lustre; and in NLustre before the expression inlining to use the following optimization passes. I chose to put it in NLustre because its much simpler syntax compared to Lustre.

2.2.2 — Which equations to inline

In Vélus as in any other compiler that implements inlining, not all functions/nodes are inlined. Instead, the user can specify by hand which must be inlined and/or a heuristic is used to select the ones that should be inlined. For testing purposes, all the nodes were inlined during this internship, but this is not efficient at all: big nodes must not be inlined as code duplication will drastically increase the size of the generated C code.

What should be done in future work is to add a boolean for each node call in NLustre indicating if the node should be inlined or not. This would be computed by an external OCaml function which would be called by Rocq during compilation. As every property has been proved regardless of the equation being inlined or not, this does not have any impact on correctness. No such algorithm has been discussed or implemented during the internship as it was not the main goal.

2.2.3 — High-level description

The idea here is to replace `App` equations that should be inlined with the equations of the called node. To avoid name conflicts between the variables already defined and the one coming from the inlined node, all the new variables are renamed using the `Fresh` monad described in [Pes23, sec. 4.2]. Moreover, for simplicity, this optimization relies on subsequent compilation passes, and in particular on expression inlining and dead equation elimination (see Figure 2). Indeed, these two passes remove trivial equations and superfluous variables (e.g., those of the form `x = y`).

Node inlining generates three types of equations:
- the equations of the inlined node, after having renamed their variables;
- new equations to link the input variables of inlined nodes to the arguments of node calls;
- new equations to link the output variables the inlined nodes to the assigned variables.

Figure 11 shows an example of inlining: the blue equations (dotted) come from the inlined node after having renamed its variables, the red ones (solid lines) link inputs to call arguments, and the green ones (dashed) link outputs to assigned variables.

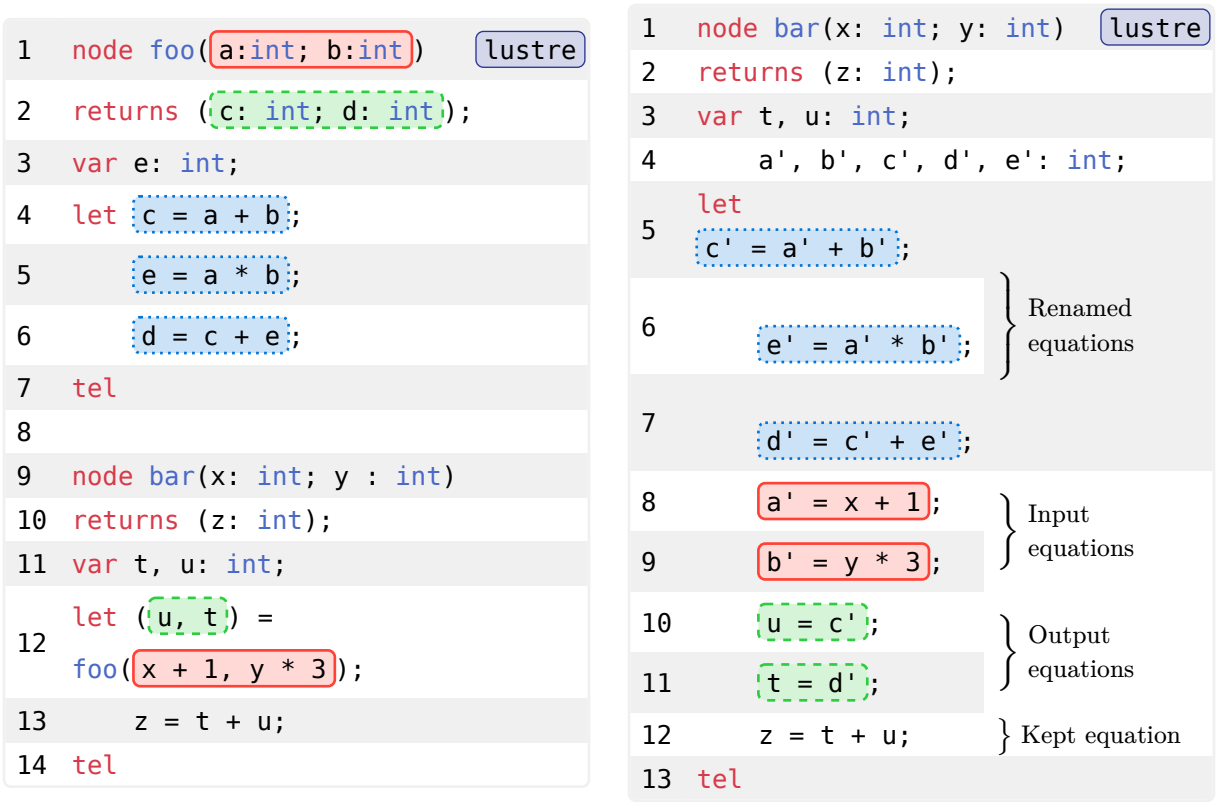

Figure 11: Detailed example of node inlining

Finally, the reset signals are *lifted*: the reset signals attached to the node call are distributed over the renamed equations and concatenated on existing reset signals. Intuitively, ensure that equations are reset at the same times after inlining (see Section 3.3.3).

2.2.4 — Precise implementation

In Vélus, a compilation pass is a tranformation of the global environment, which contains nodes, types and external function declarations. The goal of this part is then to show the construction of a function of type `global->global` that inlines every node call that should be (see Section 2.2.2).

Figure 12 is an adaptation of the precise implementation in Rocq of the algorithm that inlines a single equation. Small details have been omitted for clarity (an interesting but not important detail is developed in Appendix D). Following the color code of Figure 11, the expressions in blue are linked with the renaming of the equations of the inlined node, the red ones with input variables and call arguments, and the green ones with output and assigned variables. The variable `G` is the global environment, containing all the nodes, types and external functions of the program.

The implementation is quite straightforward: it is the direct translation in Rocq of the high-level description in Section 2.2.3. Some details remain to be exaplined: the use of `lift_reset` on renamed equations, which will be detailed in Section 3.3.3, and the fresh states `st`, ..., `st3`. These variables are internal states of the `Fresh` monad (introduced in [Pes23, part 4.2]), which guarantees some properties, among them the most important are that variables generated successively, like `new_in`, `new_out` and `new_vars`, do not contain any duplicate, and that they are different from the other variables in the node (see Section 3.1).

Using `inline_equation`, it is now possible to define `inline_equations` that inlines a list of equations by iterating over them. To use the property on fresh states, it is necessary to pass the output state of call of `inline_equation` to its successor. This operation is not complicated but defining it precisely

```
Definition inline_equation                                                    Rocq
  (G : global) (eq : equation) (st : fresh_st)
  : list equation * list var * list var * list var * fresh_st :=
  match eq with
  | EqDef _ _ _ | EqFby _ _ _ _ _ | EqLast _ _ _ _ _ => ([eq], [], [],
  [], st)
  | EqApp vars ckbase f args rs =>
    (* Case: vars =ck (reset f every rs)(args) *)
    let n := find_node f G in
    let '(new_in, st1) := fresh_idents n.(n_in) st in
    let '(new_out, st2) := fresh_idents n.(n_out) st1 in
    let '(new_vars, st3) := fresh_idents n.(n_vars) st2 in
    let sub := Env.from_list (map fst (new_in ++ new_out ++ new_vars)) in
    let renamed_eqs :=
      map (lift_reset rs) (map (rename_equation sub) n.(n_eqs))
    in
    let in_eqs := map2 (fun '((_, var_in), (_, ck, _)) arg =>
      EqDef var_in (rename_clock sub ckbase ck) (Eexp arg)
    ) new_in args in
    let out_eqs := map2 (fun var '((_, var_out), (ty, ck, _)) =>
      EqDef var (rename_clock sub ckbase ck) (Evar var_out ty)
    ) vars new_out in
    (renamed_eqs ++ in_eqs ++ out_eqs,
     map (rename_var_clock sub ckbase) new_in,
     map (rename_var_clock sub ckbase) new_out,
     map (rename_var_clock sub ckbase) new_vars,
    st3)
  end.
```

Figure 12: Function to inline an equation, adaptation of `inline_equation` in `NI.v`

is laborious, so its concrete implementation is skipped here. It is now possible to define `inline_node`, which is the important function proof-wise.

Figure 13 shows that the final transformation, that runs the node inlining algorithm in a node, expands all the node calls that should be inlined (see Section 2.2.2). This shows that the precise implementation is close to the high-level description, which is a good point for clarity.

The final tranformation to define is the function `node_inlining` of type `global->global`, mentioned at the begining of this part. As it is not necessary to ensure that variables are distinct across different nodes, the same fresh state can be used in multiple `inline_node` invocations. Finally, to make sure that inlining a node call inside a node call is done in the right order (meaning that a node call should only be inlined once: an equation calling a node cannot be inlined after being copied by another inlining),

```
Program Definition inline_node
  (G : global) (n : node) (st : fresh_st) : node := let
  '(new_eqs, new_in, new_out, new_vars, _) := inline_equations G n.(n_eqs) st
  in {| n_name := n.(n_name);
        n_in := n.(n_in);
        n_out := n.(n_out);
        n_vars := n.(n_vars) ++ new_in ++ new_out ++ new_vars;
        n_eqs := new_eqs; |}.
```

Figure 13: Function running the node inlining algorithm in all the equations of a node
Adaptation of `inline_node` in `NI.v`

```
Definition node_inlining (G : global) : global :=
  fold_right (fun n gl => {|
    types := gl.(types);
    externs := gl.(externs);
    nodes := (inline_node gl n init_st) :: gl.(nodes);
  |}) {| types := G.(types); externs := G.(externs); nodes := []; |} G.(nodes).
```

Figure 14: Function running the node inlinig pass, from `NI.v`

the transformation must be done by "folding" over the global environment. As the nodes are stored in the inverse order of their appearance in the source program (the first node is at the end of the node list in the global environment), this function can be implemented with a `fold_right` as in Figure 14. This function has been added to the main function of the compiler in `Velus.v`, with a new flag to enable/disable it (mainly for test purposes).

3 — Correctness of the node inlining

To reestablish compiler correctness after the addition of node inlining pass, some properties must be verified. The goal is to ensure the syntactic properties at the creation of the node in Figure 13, which are part of the type `node` itself (see Figure 15), and the semantic properties to complete the lemma `behavior_nl_to_cl` in `NLCorrectness.v`, which states the correctness of the compilation of NLustre to Clight (see Figure 2). During this internship, there was not enough time to complete all the required proofs: only the syntactic, typing and some important semantic properties have been proved so far.

3.1 — Syntactic properties

3.1.1 — Presentation

The syntactic properties of NLustre nodes are self-contained (see Figure 15): the only hypotheses that can be used are the definition of the new node (in Figure 13) and the syntactic proofs of the old node (coming from the transcription phase). In particular, semantic properties cannot be used here, which will be important in a later explaination.

All the syntactic properties associated with a node (and therefore to be preserved) can be seen in `NLSyntax.v`, and here translated in natural language:

- `n_ingt0`: there is at least one input, which is necessary to have the base clock of a node
- `n_outgt0`: there is at least one output
- `n_defd`: the variables defined, at the left side of the equations, are a permutation of output and local variables
- `n_lastd1`: last variables (see Section 1.3.3) are initialized by an EqLast equation

```
Record node {prefs : PS.t} : Type := mk_node {                                  Rocq
  n_name      : ident;                                  (* name *)
  n_in        : list (ident * (type * clock));          (* inputs *)
  n_out       : list (ident * (type * clock));          (* outputs *)
  n_vars      : list (ident * (type * clock * bool));   (* local variables *)
  n_eqs       : list equation;                          (* equations *)
  ... (* Syntactic properties *)
}.
```

Figure 15: Definition of a node
Adaptation of the type `node` in `NLSyntax.v`

- `n_lastd2`: after their initialization, last variables can only be defined by control expression (so external function calls are forbidden)
- `n_vout`: output variables cannot be defined with an EqFby equation
- `n_nodup`: there are no duplicate variable names
- `n_good`: all the variables are either atomic (see Section 1.4) or prefixed by a special identifier in the list `prefs`, and the name of the node is atomic

3.1.2 — Proofs

The proofs of preservation of syntactic properties are in `NI.v`.

The first two properties, `n_ingt0` and `n_outgt0`, are trivial because `n_in` and `n_out` are not modified in `inline_node` (see Figure 13). For `n_defd`, `n_lastd1`, `n_lastd2` and `n_nodup` the proofs follow the same schema: the property is first proved for the interesting case `EqApp`, then for any equation and finally for a list of equations. Then, the final lemma is used in the case of the list of equations of the node,from which the proof obligation follows almost directly.

Figure 16 shows a more detailed example for the `n_defd` property. The first lemma, proving the property for the `App` case, relies only on the definition of `inline_equation`: the proof is entirely by rewriting with no real difficulties. The second lemma, proving the propery for any equation, is very simple: it only makes a disjunction on the form of the equation, and applies the previous lemma in the only interesting case. The last lemma, proving the property in the general case, relies on an induction on the equation list, on the previous lemma and on the transitivity of the `Permutation` property (to split generated variables).

The properties `n_lastd1` and `n_lastd2` have the exact same proof scheme, so they are not more detailed. The proof for `n_nodup`, using same proof scheme, needed a little more work. Indeed, there are no duplicates among the variables generated by `inline_equations`[2]. This derives directly from the fact that two lists of idents being generated one after the other is the same as generating a big list of idents equal to the concatenation of the two lists[3]. It must then be shown that the generated variables are exactly the output of the `Fresh` monad, but this is false because the output variables of Figure 12 have their clocks renamed, and unfortunately clock renaming cannot commute with fresh ident generation. The solution is to show that variables generated before this clock renaming are exactly the output of a fresh ident generation[4]. Then, a property of the `Fresh` monad shows that idents of generated variables have no duplicate, and renaming clocks do not change idents[5]. This concludes the proof that newly introduced variables do not contain any duplicate. Then, it must be proved that there is no name

[2] Lemma `inlining_eqs_generated_variables_no_dup` in `NI.v`
[3] Lemma `fresh_idents_app` in `Fresh.v`
[4] Lemma `inlining_eqs_generated_variables_before_rename_clock_are_fresh_idents` in `NI.v`
[5] Lemma `inlining_eqs_generated_variables_before_rename_clock_map_fst` in `NI.v`

```
Lemma vars_defined_eq_app_inline G st vars ck f args rs :
  let '(new_eqs, new_in, new_out, new_vars, st') :=
    inline_equation G (EqApp vars ck f args rs) st in
  Permutation (vars_defined new_eqs) (new_in ++ new_out ++ new_vars ++ vars).

Lemma vars_defined_eq_inline G st eq :
  let '(new_eqs, new_in, new_out, new_vars, _) := inline_equation G eq st in
  Permutation (vars_defined new_eqs)
    (new_in ++ new_out ++ new_vars ++ var_defined eq).

Lemma vars_defined_eqs_inline G st eqs :
  let '(new_eqs, new_in, new_out, new_vars, _) := inline_equations G eqs st in
  Permutation (vars_defined new_eqs)
    (new_in ++ new_out ++ new_vars ++ vars_defined eqs).
```

Figure 16: Intermediate lemmas for the property `n_defd`
Adaptation of lemmas with the same names in `NI.v`

conflict between the previous variables `n.(n_in)`, `n.(n_out)` and `n.(n_vars)`, and the generated one: this is immediate thanks to the property `AtomOrGensym prefs`, that comes from `n.(n_good)`, which indicates that the already existing variables are either atomic (see Section 1.4) or have a prefix in the list `prefs`, whereas the generated variables all have `inline$` as a prefix, which is not in `prefs`.

The proof of `n_vout` was the trickiest one to find: I tried to follow the same proof scheme as `n_defd` but this did not work as intended. The solution was to use `n_nodup` to show that output variables cannot be located in newly generated variables, and to conclude by observing that an output variable was either not defined through an inlined `App`, so the property of the input node can be applied, or is defined through it, in which case the output variable is defined with an `Def` (see Section 2.2.4).

The final property `n_good` is immediate because the node name did not change so it is atomic, and all the other variables already had good names. The only trick here is to add the prefix `inline` to the list `prefs`, so that generated variables verify `AtomicOrGensym`.

In conclusion, syntactic properties were not hard to prove: most of the proofs were direct implementation of intuitive ideas. However, they provided a good entry point in the Vélus architecture, which was helpful for the following proofs. Some tricks had to be implemented (like the one described in Appendix D), but nothing hard.

3.2 — Typing preservation

3.2.1 — Presentation

An expression in NLustre is said to be *well-typed* in a context `Γ: list (ident * type * bool)` if, for every variable x of type ty that it contains, $(x, ty, \text{true}) \in \Gamma$ if x is used in a `last` expression (see Section 1.3.1), and $(x, ty, \text{false}) \in \Gamma$ otherwise[6]. Moreover, a clock ck is well-typed in a context Γ if every variable x of type ty of ck is such that $(x, ty, \text{false}) \in \Gamma$ (see `wt_clock` in `CETyping.v`).

Well-typing extends to equations in a context Γ if every variable, expression and clock of the equation is well-typed in Γ[7]. A node is said to be well-type in a global environment G (see Section 2.2.4) if it satisfies two conditions (see `wt_node` in `NLTyping.v`):

[6] Defined by `wt_exp`, `wt_cexp` and `wt_rhs` in `CETyping.v`
[7] Defined by `wt_equation` in `NLTyping.v`

- Each equation is well-typed in the environment Γ defined as the concatenation of input, output and local variables (the boolean is set to `false` for input and output variables, and to the boolean contained in the local variables otherwise, see Figure 15).
- Each type of an input, output or local variable is either a primitive type or declared in the global environment G.

However, these two properties were not sufficient to prove that well-typedness, a third condition has been added: each clock of an input, output or local variable is well-typed in the same environment Γ as the one used for equations. This property has been proved in the transcription phase (see Figure 2) to ensure its validity. Its usage is detailed in Section 3.2.2.

Finally, a global environment G is said to be well-typed if there are no duplicate nodes names, and if every node is well-typed in the global environment G restricted to the nodes defined previously[8].

3.2.2 — Proofs

The proofs of typing preservation are in `NITyping.v`.

The proofs of type preservation are mostly direct, following the same kind of case analysis as done for syntactic properties (see Section 3.1.2). Two facts are worth mentionning: the proof of type preservation of an equation and of a node cannot only rely on the typing properties of the input equation/node, and the typing properties are preserved under renaming.

Indeed, as node inlining requires information only present in the global environment, the typing properties must be true not only for the equation/node before inlining, but also for the nodes that will be inlined. Moreover, it must be known that called nodes are defined before the current one in the global environment. This is exactly the meaning of the predicate `Ordered_nodes` (defined in `NLOrdered.v`): proving it for a node is not direct, but an important lemma for this predicate indicate that `wt_global G` implies `Ordered_nodes G`[9]. The complete proof of type preservation, which states that `wt_global G` implies `wt_global (node_inlining G)`, follows by induction on nodes in the global environment. The well-typedness of the nodes of `G` are then used to show `Ordered_nodes`, and the well-typedness of the nodes of `node_inlining(G)` to show the property for the inlined nodes after a call to `inline_equation` on an `EqApp`.

Moreover, as the equations coming from the inlined node are renamed to avoid duplicates (see Section 2.2.4 and Section 3.1.2), it must be shown that renaming has no incidence on typing. This results in a series of lemmas of the form "`wt Γ x` implies `wt (rename_environment Γ) (rename x)`". Figure 17 shows an example of such lemma for the case of a simple expression.

In conclusion, typing proofs were not hard either. They were rather intuitive, and mostly direct but long. Moreover, they involved a large number of concepts from different parts of the compiler, which was really helpful not to be in difficulty during the semantic proofs. Finally, the definition of well-typedness changed by a little, which was necessary for its preservation. The change is intuitive, so no downside had to be noted about it.

```
1   Lemma wt_exp_rename (types : list type) (sub : Env.t ident)                Rocq
2     (Γ : list (ident * (type * bool)) (e : exp) :
3     wt_exp types Γ e -> wt_exp types (rename_environment Γ) (rename_exp sub e).
```

Figure 17: Typing preservation of an expression after renaming variables
Adaptation of `wt_exp_rename` in `NITyping.v`

[8] Defined by `wt_global` in `NLTyping.v`
[9] Lemma `wt_global_Ordered_nodes` in `NLTyping.v`

```
1 Inductive svalue : Type := absent | present (v : value).
2 Definition env : Type := ident -> option svalue.
3 Definition history : Type := nat -> env.
4 Definition vstream : Type := nat -> svalue.
```
Rocq

Figure 18: Definition of an hisory in Rocq, adapted from `IndexedStreams.v`

3.3 — Semantics preservation

3.3.1 — Presentation

The idea of semantics preservation is broadly the same as for type preservation: under the hypothesis that a node `f` has a "good semantics" in a global environment `G`, we must show that the same node has the same semantics in the global environment `node_inlining G`. In Vélus, NLustre has two equivalent semantics: an *indexed semantics* where `stream A` is defined as `nat -> A`, and a coindexed semantics where `Stream A` is defined as follows (from the Rocq standard library):

```
1 CoInductive Stream (A : Type) := Cons : A -> Stream A -> Stream A.
```
Rocq

A proof of equivalence between the two semantics [Bru20, part 2.4] is in `NLIndexedToCoind.v` and `NLCoindToIndexed.v`. For our proofs indexed semantics is easier to work with, so only the indexed representation of streams will be used here.

An environment is needed to associate variables, denoted by their identifiers, to their values at each instant. This is called *history*, and is defined in Figure 18. It is now possible to define the semantics of a variable with the predicate `sem_var_instant E x v`, whose meaning is "the variable `x` has value `v` in the environment `E`", and the predicate `sem_var H x v`, whose meaning is "at each instant `i`, the variable `x` has value `(v i)` in the environment `(H i)`" (see Figure 19).

This definition is extended to the expressions defined in Section 1.3.1, with the usual semantics for Clight operators and `case`, and with the semantics defined in Section 1.3.2 for `merge` and `when`. The operator `last` has a more complex semantics that will be detailed later. Semantics for equations and nodes are defined by mutual induction: the semantics of a node depends on the semantics of each equation, and the semantics of a node call through an `App` equation is the semantics of the called node. These definitions are called `sem_equation` and `sem_node` in `NLIndexedSemantics.v`, and will be detailed when needed. The meaning of `sem_equation G bk H e` is "the equation `e` has a good semantics in `H` with a base clock `bk` in the environment `G`". In the case of a node, `sem_node G f ins outs` means that "the node `f` associates the inputs `ins` to the outputs `outs` in the environment `G`". The formal definition is in Figure 20.

3.3.2 — Proof overview

The proof of semantics preservation is in `NICorrectness.v`.

The theorem of correctness (semantics preservation) is: `sem_node G f ins outs` implies `sem_node (node_inlining G) f ins outs`, to show that whether or not nodes are inlined, the semantics remains the same. As the definition of a node may change during the call to `node_inlining`, it is necessary to apply `SNode`, the only constructor of `sem_node` (see Figure 20). However, it requires to exhibit an

```
1 Definition sem_var_instant (E : env) (x : ident) (v : svalue) : Prop :=
2   E x = Some v.
3 Definition sem_var (H : history) (x : ident) (v : stream svalue) : Prop :=
4   forall (i : nat), sem_var_instant (H i) x (v i).
```
Rocq

Figure 19: Semantics of a variable, adapted from `IndexedStreams.v`

```
1 Inductive sem_node: (* ... *) -> Prop :=
2   | SNode (H: history) (f: ident) (ins outs: list vstream) (n: node):
3     find_node f G = n ->
4     sem_vars H (map fst n.(n_in)) ins -> sem_vars H (map fst n.(n_out)) outs ->
5     sem_clocked_vars (clock_of ins) H n.(n_in) ->
6     Forall (sem_equation (clock_of ins) H) n.(n_eqs) ->
7     sem_node f ins outs.
```

Figure 20: Adaptation of `sem_node`, from `NLIndexedSemantics.v`

history `H` such that every equation `e` of the node `n` has a good semantics with history `H` in environment `G` (line 5 of Figure 20), with `n` in `G` with name `f`. `H` contains all the values taken by the variables in `n`. However, providing such `H` is no easy task as it cannot be derived entirely from `sem_node G f ins outs`: indeed, each inlined node will come with many new variables that must be present in the provided `H`.

It is then necessary to prove that such an `H` exists. It can be made explicite by stating that each inlined node contains a history with an inversion on the semantics of the equation (see Figure 21). The complete `H` is then the disjoint union of all the environments corresponding to the inlined nodes, and of the one coming from the inversion of the hypothesis `sem_node G f ins outs`. A lemma states a major property needed for this sketch[10]: if an equation `eq` has a good semantics in the environment `G` with an history `H`, then either this equation is not a node call or is a node call not inlined, or either this equation is an inlined node call, in which case there exists an history `H'` such that all the generated equations have a good semantics in the environment `G` with an history `H ⊎ H'`.

From this property, three questions can be asked: How to generate such `H'`? Why the union of `H` and `H'` and not just `H'`? Why is the union disjoint? The intuitive idea is that by inverting a `sem_equation` with an `App` equation (see Figure 21), the semantics of the called node is then provided as an hypothesis, allowing us to do an inversion on this hypothesis (see Figure 20), which gives an history `H'`. This `H'` only contains (or at least can be restricted to) contain only the variables of the called, whereas `H` contains the variables of the callee node. As all of the variables have been proven to be dsjoint (in Section 3.1), it is possible to prove that `H` and `H'` are disjoint. For example, in Figure 11, `H` contains `x`, `y`, `z`, `t`, and `u` whereas `H'` contains `a'`, `b'`, `c'`, `d'`, and `e'`.

Unfortunately, this idea does not work directly. Inverting on `sem_equation G bk H (EqApp xs ckbase f args rs)` do not give `sem_node G f ins outs` for a node of name `f`, but `forall k, sem_node f (mask k rs ins) (mask k rs outs)` (see Figure 21). The difference here is crucial: it means that the semantics of the node does not hold directly for all the stream of inputs and outputs, but holds separately for each mask sampled by the reset signals (see Section 1.3.4). The key point is now to find a link between the semantics of the node that holds for all the input and output streams, knowing that it holds for each masked section of the same input and output, delimited by the reset signals.

```
1 Inductive sem_equation: stream bool -> history -> equation -> Prop :=
2   | SEqApp (H : history) (f : ident) (* ... *) : (* ... *) ->
3     (forall k, sem_node f (mask k rs ins) (mask k rs outs)) ->
4     sem_equation bk H (EqApp xs ck f args rs)
```

Figure 21: Adaptation of `sem_equation` for the case `App`, from `NLIndexedSemantics.v`

[10] Lemma `node_inlining_sem_node_exist_equation_history` in `NLIndexedSemantics.v`

```
Definition lift_reset (rs : list ident) (eq : equation) : equation :=
  match eq with
  | EqDef x ck e => EqDef x ck e
  | EqLast x ty ck c rs' => EqLast x ty ck c (rs ++ rs')
  | EqApp xs ck f args rs => EqApp xs ck f args (rs ++ rs')
  | EqFby x ck c e rs => EqFby x ck c e (rs ++ rs')
  end.
```

Figure 22: Definition of `lift_reset`, from `NLSyntax.v`

3.3.3 — Lift reset

For clarity, in this part i and j will always denote an instant, and k the number of a mask.

3.3.3.1 — Properties overview

The last issue introduced in the previous part is the main problem encountered during the internship. The goal is to have a property stating the semantics of all the equations of the called node for the inputs and outputs. Intuitively, these equations are reset by two lists of reset signals: the one coming from the equation, the inner reset signals, and the one coming from the fact that the node call, that inlined the equation, is also reset. This explains the use of the `lift_reset` function, introduced for this purpose and defined in Figure 22. Its definition is quite simple: it concatenates the inner and the outer reset signals. The lemma `sem_node_unmask`, whose sketch is in Figure 23, glues everything together: it states that if the semantics of a node is true for each mask of the inputs and outputs, then there exists a history `H'` for which each of the node equations satisfies the semantics after lifting the outer reset signals with which the node has been called. The proof of this lemma is not complete, but all the biggest issues have been addressed.

3.3.3.2 — Existence of a history `H'`

The existence of this history can be summed up in Figure 24. The goal is to provide an H' such that at each instant i, $H'(i)$ is the environment obtained from the inversion of the hypothesis `sem_node` for the corresponding mask k. Finding k from i is easy: it is the index of the mask delimited by `rs` at the instant i. This is exactly the number of times `rs` is T between 0 and i (included), which is the definition of the function `count` (see Figure 25). So, the mask k in which the instant i is located is exactly `count rs i`. To obtain H'_k, it is sufficient to do an inversion on the hypothesis `sem_node ...` with $k =$ `count rs i` (see Figure 20). However, we need to give a function of type `nat -> env`, and `sem_node` is a property, which cannot be inverted when the goal is not a property due to Rocq's type system. To solve this issue, we need to use the functional axiom of choice (which was already in use in the compiler), which is defined in Rocq's standard library[11]. It allows to convert a `forall (a : A), exists (b : B), R a b` into an `exists (f : A -> B), forall (a : A), R a (f a)`, which is exactly what is needed. Finally, to provide the history `H'`, we use the function `fH` provided by the functional axiom of choice: it ensures that for any mask `k`, `fH k` is the history obtained after inversion of `sem_node`. The history provided to apply `SNode` (see Figure 20) is then exactly the function `fun (i : nat) => (H i) ∪ (H' i)`, where `H' := fun i => fH (count rs i) i`: at each instant `i`, we take `fH` at the mask index $k =$ `count rs i` and the instant i (see Figure 24).

```
Lemma sem_node_unmask (*...*) :
  (forall (k: nat), sem_node G f (mask k rs ins) (mask k rs outs)) ->
    exists (H' : history), (* ... *) /\
    Forall (sem_equation G bk H') (map (lift_reset rs) n.(n_eqs)).
```

Figure 23: Sketch of lemma `sem_node_unmask`, adapted from `NLIndexedSemantics.v`

[11] In `Coq.Logic.IndefiniteDescription`

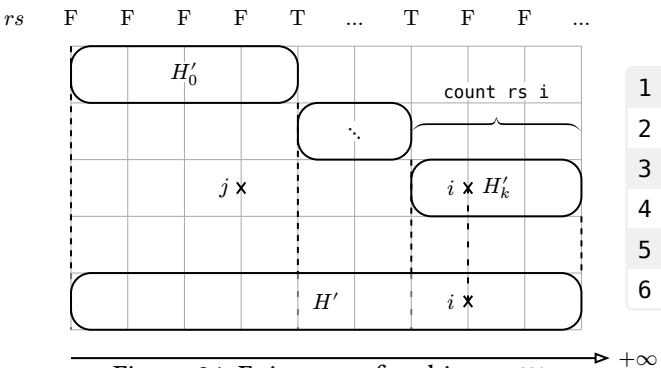

Figure 24: Existence of an history `H'`

```
1 Fixpoint count                                          Rocq
2   (rs : stream bool) (i : nat) : nat :=
3   let c := match i with
4     | 0 => 0
5     | S i => count rs i
6   end in if rs i then S c else c.
```

Figure 25: Definition of `count`, from `IndexedStreams.v`

3.3.3.3 — Proof of `sem_equation_lift_reset`

To show `sem_node_unmask`, now that `H'` has been defined, it is sufficient to reason on one equation at a time to prove the `Forall` statement. This introduces the biggest lemma of the internship whose proof is complete, `sem_equation_lift_reset`, and whose sketch is in Figure 26. Only the major hypotheses are displayed here: the first one states, see Figure 24, that the environment defined outside the corresponding mask, for example $H'_k(j)$, associates every variable to either `None` (undefined) or `Some absent` (defined but always absent). The second one indicates that the environment of the calling node `H` is disjoint from the the environment of the calling node. The third one comes directly from the hypothesis of the lemma `sem_node_unmask`, whose proof invokes this lemma.

The proof is built on a disjunction on the type of equation. In each case, a functional choice is done to exhibit variables needed by the constructor of `sem_equation` (see Appendix E). The case `Def` is direct because it does not hold any reset signal, so `lift_reset` has no effect on the semantics (see Figure 22). The other cases are much more difficult. In particular, one property for the `Last`/`Fby` cases (the proofs are almost exactly the same), and one property for the `App` case.

For the cases `Fby/Last`, let's denote the outer reset signals (coming from the node call) `rs` and the inner (coming from the equation) `rs'`. The property can be rewritten as in Figure 27. Here, `reset` and `fby` are not NLustre operators but Rocq functions used for the indexed semantics. Their definitions are detailed in Appendix F, and here is a brief description: in a context where `xs` is always present and `v` is constant, `reset v xs rs i` is equal to `v` if `rs i` is T, and `xs i` otherwise. Moreover, under the same assumption, `fby v xs 0` is equal to `v`, and `fby v xs (S i)` is equal to `xs i`. This property may seem conterintuitive but has been formally proved[12]. The first idea tested such property was to reason by induction on the instant `i`, but this did not work because of the absences. Instead here are the disjunctions (see schemas in Figure 28 to follow the cases): if `xs i` is absent then both sides are also absent so equal. Otherwise, `xs i` is present and two cases are to consider: `rs' i = true` and `rs' = false`.

- If `rs' i = true` (case ① in Figure 28), then both sides are reset, so by the definition of `reset` both sides are equal to `v`.

```
1 Lemma sem_equation_lift_reset (* ... *) :               Rocq
2 (* 1 *) (forall (k i : nat) (x : var_last),
3           k <> count rs i -> fH k i x = None \/ fH k i x = Some absent) ->
4 (* 2 *) (forall (i : nat), disjoint (H i) (fH (count rs i) i)) ->
5 (* 3 *)(forall (k : nat), sem_equation G (mask k rs bk) (fhk k) eq) ->
6   sem_equation G bk (fun i => (H i) ∪ (fH (count rs i) i)) (lift_reset rs eq).
```

Figure 26: Sketch of lemma `sem_equation_lift_reset`, adapted from `NLIndexedSemantics.v`

[12] Lemma `sem_reset_fby_lift_reset_extension` in `NLIndexedSemantics.v`

```
1  (forall (j : nat), count rs j <> count rs i -> xs j = absent) ->
2    reset v (fby v xs) rs' i = reset v (fby v xs) (fun j => rs j || rs' j) i
```

Figure 27: Simplified property for `Last`/`Fby` cases

- If `rs' i = false` , two other cases are to consider: `rs i = true` and `rs i = false`.
  - If `rs i = true` (case ② in Figure 28), then the right side is reset so is equal to `v`, and `i` is the first instant in which `xs i` is present (because of the hypothesis, and that `j < i -> count rs j <> count rs i` as `rs i = true`), so the left side must be equal the first value of the `fby`: both sides are equal to `v` (see lemma `reset_fby_first_value`).
  - The last case is the most complex and interesting: `rs i = false` and `rs' i = false`: the idea is to consider whether there exists a instant $j \in [\![0, i[\![$ such that $xs\ j \neq \text{absent} \vee (rs\ j \parallel rs'j = \text{T})$.
    - If no such j exists, then $\forall j \in [\![0, i[\![, xs\ j = \text{absent} \wedge rs\ j = \text{F} \wedge rs'j = \text{F}$, so by the same argument as in ②, both sides are equal to `v`.
    - Is such j exists, we take the maximal j such that $xs\ j \neq \text{absent}$ or $rs\ j \parallel rs'j = \text{T}$. Thus, $\forall j < m < i, xs\ m = \text{absent} \wedge rs\ m = \text{F} \wedge rs'm = \text{F}$.
      - If $xs\ j = \text{present}\ c$ (case ③ in Figure 28), then the instant i is not reset and no reset has to be propagated because of the absences, so both sides are equal to $xs\ i$.
      - Otherwise, $xs\ j = \text{absent}$ and $rs\ j \parallel rs'j = \text{T}$ (case ④ in Figure 28), so the reset signal in j is propagated to i, so both sides are equal to v.

All the cases are covered, which concludes the proof for the cases `Fby/Last`.

For the case `App` where the node with name `g` is called, with input streams `ins` and output streams `outs`, let's denote the outer reset signals (coming from the node call) `rs` and the inner (coming from the equation) `rs'`. The property can be rewritten as in Figure 29. The key idea is that knowing that the node `g` has a good semantics on every mask `m` induced by the reset signals `rs'`, we have to show that this is the case on every mask `l` induced by the reset signals `rs || rs'` (notation for `fun (i : nat) => rs i || rs' i`). This is summed up in Figure 31. As a well-typed node whose inputs and outputs are always absent has a good semantics[13], the case study can be limited to three possibilities:

- If the mask induced by `rs'` is contained in the mask induced by `rs` (case i_2 in Figure 31), then there exists a mask index induced by `rs || rs'` which is exactly the same as the one induced by `rs'`. In this case, the hypothesis on the corresponding mask induced by `rs'` solves the goal directly.

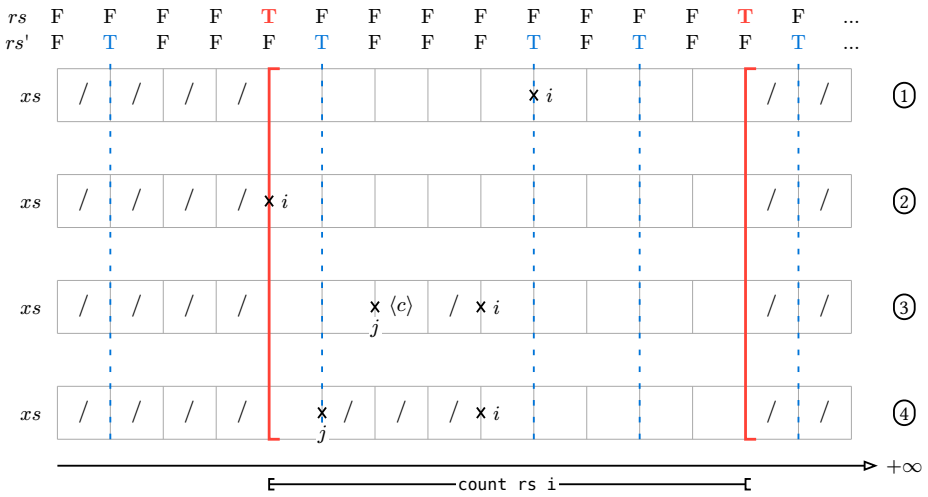

Figure 28: Example cases for the proof of `sem_reset_fby_lift_reset_extension`

[13] Lemma `sem_node_absent` in `NLIndexedSemantics.v`

```
(forall (j : nat), count rs j <> k -> ins j = absent ) ->
(forall (j : nat), count rs j <> k -> outs j = absent) ->
(forall m : nat, sem_node G g (mask m rs' ins) (mask m rs' outs)) ->
  sem_node G g (mask l (rs || rs') ins) (mask l (rs || rs') outs)
```

Figure 29: Simplified property for `App` case

```
Lemma count_ors_incl_iff (k : nat) (rs1 rs2 : cstream) :
  exists (n m : nat), forall (i : nat),
  count (ors rs1 rs2) i = k <-> count rs1 i = n /\ count rs2 i = m.
```

Figure 30: Definition of `count_ors_incl_iff`, from `IndexedStreams.v`

- If the mask induced by `rs'` ends but does not start, or starts but does not end in the mask induced by `rs` (cases i_1 and i_3 in Figure 31), then the semantics is preserved because extending by the left or by the right inputs and outputs with absences does not change the semantics of a node, because inputs and outputs are absent outside the mask induced by `rs`.

The key property to prove this lemma is given by `count_ors_incl_iff` in `IndexedStreams.v`, whose definition is in Figure 30, and which describes the "intersection" of the masks induced by `rs`, `rs'`, and `rs || rs'`. Other big but less interesting detail are discussed in Appendix E. This concludes the proof of `sem_equation_lift_reset`, which is the main lemma of the semantics preservation that has been found during this internship.

4 — Conclusion

During this internship, node inlining has been fully implemented, and many of the properties required to ensure compiler correctness were proved. The time spent on getting used to Vélus and on syntactic and typing proofs was expected, but the semantic preservation proofs were a much harder than what we thought at first. In particular, the modular reset brought problems that took weeks to solve entirely.

Adding node inlining in Lustre and not in NLustre would have solve a significant issue: no need to propagate the outer reset signals with a lift reset, creating a reset block would suffice (see Appendix B). The new blocks would then have been normalized by the last block compilation pass (see Figure 2). However, proving that the nodes before and after inlining share the same semantics may have been harder. The natural follow-up question is whether node inlining would have been in Lustre. This is not a priority for further development of Vélus, but it would be interesting to see this question answered.

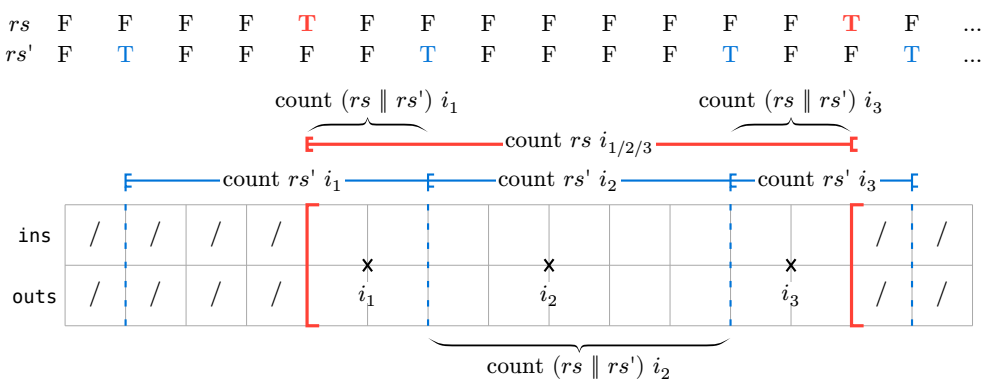

Figure 31: Schema of `sem_node_mask_or`

Appendixes

A — Full paths of source files

Abbreviation	Full path
CETyping.v	src/CoreExpr/CETyping.v
CommonList.v	src/Common/CommonList.v
Ident.v	src/Ident.v
IndexedStreams.v	src/IndexedStreams.v
NI.v	src/NLustre/NodeInlining/NI.v
NICorrectness.v	src/NLustre/NodeInlining/NICorrectness.v
NITyping.v	src/NLustre/NodeInlining/NITyping.v
NLCorrectness.v	src/NLCorrectness.v
NLCoindToIndexed.v	src/NLustre/NLCoindToIndexed.v
NLIndexedSemantics.v	src/NLustre/NLIndexedSemantics.v
NLIndexedToCoind.v	src/NLustre/NLIndexedToCoind.v
NLOrdered.v	src/NLustre/NLOrdered.v
NLSyntax.v	src/NLustre/NLSyntax.v
NLTyping.v	src/NLustre/NLTyping.v
Velus.v	src/Velus.v

B — Lustre syntax

Lustre is a much more complex language than NLustre. Here is the formal definition:

Expressions

$e ::= c$ (constant)

| C (enumeration value)

| x (variable)

| $\diamond\, e$ (unary operator)

| $e \oplus e$ (binary operator)

| e^+ `fby` e^+ (follow-by operator)

| $e^+ \rightarrow e^+$ (arrow operator)

| e^+ `when` $C(x)$ (sub-sampling)

| `merge` x $(C \Rightarrow e^+)^+$ (over-sampling)

| **case** e `of` $(C \Rightarrow e^+)^+$ (pattern matching)

| $f(e^+)$ (node call)

| (`reset` f `every` e)(e^+) (node call with reset)

Node declarations

$n ::=$ **node** $f(var^+)$ **returns** (var^+) blk

Blocks

blk ::= $x^+ = e^+$ (equation)

| **var** var^* **let** blk^+ **tel** (local block)

| **switch** e $(C$ **do** $blk^+)^+$ **end** (switch block)

| `last` $x = e$ (shared variable initialization)

| `reset` blk^+ `every` e (reset block)

| **automaton initially** $autinits$ $(\textbf{state}\ C\ autscope)^+$ **end** (weak automaton)

| **automaton initially** C $(\textbf{state}\ C\ \textbf{do}\ blk^+\ \textbf{unless}\ trans)^+$ **end** (strong automaton)

Automaton initialization

$autinits$ ::= C (default state)

| **if** e **then** C **otherwise** $autinits$ (conditional flow)

Automaton scopes

$autscope$ ::= **var** var^* **do** blk^+ **until** $trans^+$

Automaton transitions

$trans$ ::= **if** e **continue** C

| **if** e **then** C

Variable declarations

var ::= x : ty **on** ck

Type declarations

ty ::= **type** $tx = C^+$

The main differences with NLustre are:

- Operators like `fby` and node calls can occur in a sub-expression.
- Nested blocks, where`reset` can be applied to a block (of equations, of local blocks, ...).
- State automatons (compiled during State machines phase in Figure 2).

C — sin/cos program

The following program returns an approximation of sinus and cosinus, where the input starts at 0 and increases with a step `pas`. The formulas come from the Taylor expansion of sinus and cosinus. It is a direct adaptation of the test in `test/good/t00.zls` from Zélus [BP13], a synchronous language extended with Ordinary Differential Equations (ODEs) to program hybrid systems that mix discrete-time and continuous-time models. Currently the program in Figure 32 does not compile in Vélus, but it is the goal of node inlining with a better dependency analysis.

D — Usage of `map2_left` in `inline_equation`

In the function `inline_equation` (whose implementation is in Figure 12), a simplification has been made on the definition of `in_eqs` and `out_eqs`. Indeed, using the standard `map2` (whose implementation is in Figure 33) led to being unable of proving the syntactic property `n_defd` (see Section 3.1). This was due to the fact that using `map2` on two lists of different lengths returns a new list whose length is the minimum of the lengths of input lists. Moreover, during the syntactic proofs, there is no property specifying that a node call must have the good number of arguments and variables on the left side: it is implied by typing and semantics properties. So, there was no way of proving that `in_eqs` defines exactly the `new_in` variables: some variables could be lost due to the usage of `map2`.

```lustre
node euler_forward(h: real; x0: real; der: real)
returns (y: real);
let
  y = x0 fby (y + h * der);
tel

node euler_backward(h: real; x0: real; der: real)
returns (y: real);
let
  last y = x0;
  y = last y + h * der;
tel

node sin_cos(pas: real)
returns (sy: real; cy: real);
let
  sy = euler_forward(pas, 0.0, cy);
  cy = euler_backward(pas, 1.0, - sy);
tel
```

Figure 32: Lustre program computing `sinus` and `cosinus`

The idea I had to overcome this was to introduce a new function `map2_left` (whose definition is in Figure 34). The idea is to provide a defaut argument for the second argument of the function, to ensure that the length of the result is equal to the length of the first list (see the lemma `map2_left_length` in `CommonList.v`). This trick is enough to prove the syntactic properties, and does not impact the other proofs because typing properties ensure the equality between the lengths of the two lists: `map2_left` can then be rewritten into `map2` (see the lemma `map2_left_is_map2` in `CommonList.v`).

```rocq
Fixpoint map2 {A B C : Type}
  (f : A -> B -> C) (l : list A) (l': list B) : list C :=
  match l, l' with
  | [], _ | _, [] => []
  | a :: l, b :: l' =>
    f a b :: map2 f l l'
  end.
```

Figure 33: Definition of `map2`, from `CommonList.v`

```rocq
Fixpoint map2_left {A B C : Type}
  (default : B) (f : A -> B -> C) (l: list A) (l': list B) : list C :=
  match l, l' with
  | [], _ => []
  | a :: l, [] => f a default :: map2_left default f l []
  | a :: l, b :: l' => f a b :: map2_left default f l l'
  end.
```

Figure 34: Definition of `map2_left`, from `CommonList.v`

```
Lemma functional_choice :
  forall (A B : Type) (R : A -> B -> Prop),
    (forall x : A, exists y : B, R x y) ->
    (exists f : A -> B, forall x : A, R x (f x)).
```

Figure 35: Functional axiom of choice,
from `Coq.Logic.IndefiniteDescription`

E — Existential variables of `sem_equation`

E.1 — Presentation

In addition to showing that an history `H'` satisfiying `sem_node_unmask` exists (see Section 3.3.3.2), the functional choice property (whose definition is in Figure 35) is used once for each case in the proof of lemma `sem_equation_lift_reset`. This is needed to exhibit variables that must be provided when applying the constructor of `sem_equation` for the good equation variant.

For example, in the `Last` case, whose constructor is in Figure 36, the variables `xs`, `ys` and `rs` do not appear in the conclusion so they are existential variables. Moreover, as the main hypothesis tells that `forall (k : nat), sem_equation G (clock_of (mask k rs ins)) (fhk k) eq` (see Figure 26), it is not possible to give immediatly a definition for those variables[14]. Indeed, the situation is the same as in Section 3.3.3.2: for all mask index k, it is possible to give variables xs_k, ys_k and rs_k satisfying the properties, for this mask k. The functional axiom then gives us a function f which provides for each k the variables xs_k, ys_k and rs_k. Finally, using the same principle as in Section 3.3.3.2, the needed `xs` is equal to `fun i => let '(xsk, _, _, _) := f (count rs i) in xsk i`. Some variables however, like `ys` in the case `Last`, require more work.

E.2 — From syntactic to semantics reset signals

In this part, `Last` is taken as an example, but this is also true for the cases `Fby` and `App`.

A simplification that has been made in Section 1.3.4 cannot be use here: the syntactic and semantics representation of reset signals are here represented by different variables. In Figure 36, syntactic reset signals (list of identifiers) are denoted by `xrs`, and semantics reset signals (stream of booleans) are denoted by `rs`. The link between the two is done by the two properties `Forall2 (fun '(x, _) => sem_var H (Var x)) xrs ys` and `bools_ofs ys rs`. The idea here is to create a new variable `ys` of type `list (stream bool)` such that each variable in `xrs` has for semantics the stream at the same

```
Inductive sem_equation: stream bool -> history -> equation -> Prop :=
  | SEqLast (bk : stream bool) (H : history) (x : ident) (ty : type)
    (ck : clock) (c0 : cst) (xrs : list (ident * clock)) (xs : stream svalue)
    (ys : list (stream bool)) (rs : stream bool) :
      sem_var H (Var x) xs ->
      sem_clocked_var bk H (Var x) ck ->
      Forall2 (fun '(x, _) => sem_var H (Var x)) xrs ys ->
      bools_ofs ys rs ->
      sem_var H (Last x) (reset (sem_const c0) (fby (sem_const c0) xs) rs) ->
      sem_equation bk H (EqLast x ty ck c0 xrs).
```

Figure 36: `sem_equation` constructor for the case `Last`,
from `NLIndexedSemantics.v`

[14] After some discussion, we conjectured that it is possible for some variables using the history. However, this solution would be more complex to reason on it, so we chose to stay with the use of the functional axiom of choice.

```
ys' := map (
  fun idx => fun i => let '(_, ys, _, _) := f (count rs i) in ys[idx] i
) (seq 0 m)
```

Figure 37: Simplified definition of `ys'`

position in `ys`, and for each `i`, `rs i = true` if and only if at least one of the streams `y` of `ys` satisfies `y i = true`.

Intuitively, as the equation must be reset if at least one of the reset signals is true, the semantics of `xrs ++ xrs'` is `fun i => rs i || rs' i`, with `rs'` defined as `xs` above: `rs' := fun i => let '(_, _, rsk, _) := f (count rs i) in rsk i`. Moreover, the intermediate variable `ys` linked to `xrs ++ xrs'` would be `ys ++ ys'` for some `ys'`. One could think that it is sufficient to define `ys'` as `xs` or `rs'` above, but unfortunately this does not work. Indeed, if we keep the part of `f` linked to `ys`, denoted after `fys`, its type would be `fys : nat -> list (nat -> bool)`, and we want a variable `ys'` of type `list (nat -> bool)`. Here comes the main issue: inverting a stream of lists and a list of streams.

The solution that has been found is the following: take the variable at the same index of the list at each instant. This can work by the fact that for all `i`, the length of `fys i` is equal to the length of `xrs'`, denoted `m` (which can be easily proved). Then, the simplest way of doing it is by creating a list `[0, 1, ..., m - 1]`, then mapping each index to the good stream of `fys`. As a result, the provided variable `ys` is `ys ++ ys'`, where `ys'` is defined as in Figure 37. The lemmas `sem_reset_signals_variables_concat` and `sem_reset_signals_bools_ofs` then prove that the variables shown here are satisfy both properties.

E.3 — Changes in the proof in the case `App`

To hide the complexity induced by the use of the functional axiom of choice, the properties presented in Figure 27 and Figure 29 are simplified: for example, the variables xs and xs_k are assimilated. For the cases `Last` and `Fby`, this do not change the proof a lot, but the case `App` requires another big step in the proof. Indeed, in Figure 29, the hypothesis `forall m : nat, sem_node G g (mask m rs' ins) (mask m rs' outs)` is in reality `forall m : nat, sem_node G g (mask m rsk ins) (mask m rsk outs)`.

This change is in fact really important, because of the `count` function that depends on the all the values of the reset signals. The situation can be summed up in Figure 38: rs_k is equal to rs' on the mask of index k, and is equal to F everywhere else. The only thing missing to complete the proof is to show that any mask induced by `rs || rs'`, which is entirely contained in the k^{th} mask of `rs`, is also a mask of `rs || rsk`. This property is shown by the lemma `mask_extended_reset_on_window` (in `NLIndexedSemantics.v`), which shows the existence of an index `m` such that `forall xss i, mask l rs' i xss = mask m rsk xss i`. This derives directly from another property on `count`, `count_extended_reset` (in `NLIndexedSemantics.v`), whose definition is in Figure 39.

F — `reset`, `fby`, `doreset`, and `hold` functions

The functions `reset` and `fby` have been quickly introduced in Section 3.3.3.3, but their implementation is more complex that one could expect due to the absences. Indeed, absences "delay" reset signals and the left part of a `fby`. The definition of `fby` relies on another function `hold`, both defined in Figure 40, and the definition of `reset` relies on `doreset`, both defined in Figure 41. In particular, those definitions exhibit two facts on `fby` and `reset`:

- It confirms that using an induction on `i` alone is not suited to prove properties on those functions. It is however useful with another property that gives information on the previous values. For example, the lemma `fby_uninitialized` (in `NLIndexedSemantics.v`), whose definition is in Figure 42, is proved with an induction on the instant `i`, but this works due to the other hypothesis that states a property for all instant `j` such that `j < i`.

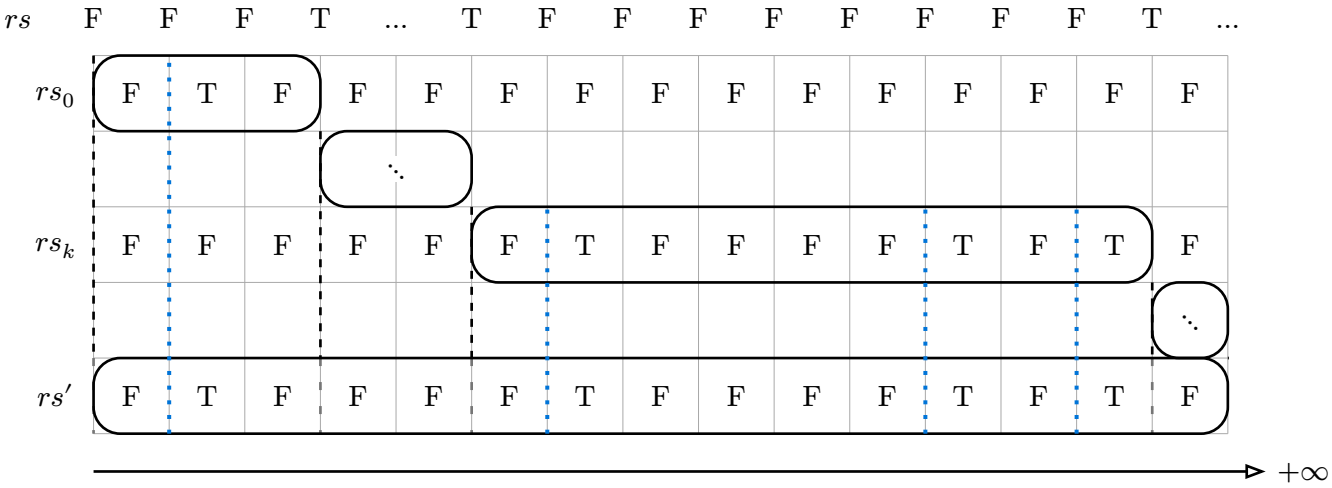

Figure 38: Definitions of rs_k and rs'

```
Lemma count_extended_reset (rs : cstream) (frs : nat -> cstream) (k n :
nat) :
  count rs n = k ->
  (forall (i : nat), count rs i <> k -> frs k i = false) ->
    exists (d : nat), forall (i : nat),
      count rs i = k ->
      count (fun j => frs (count rs j) j) i = d + count (frs k) i.
```

Figure 39: Lemma `count_extended_reset`, from `NLIndexedSemantics.v`

- It shows a key property of Lustre/NLustre nodes: the indepence from initial absence. This property, shown in [BPP23] and described in [Jea24, part 4.3], states intuitively that a function `f` associated to a node (like `fby` or `reset`), always verifies $\forall\ xs, f(\langle\rangle \cdot xs) \simeq \langle\rangle \cdot f(xs)$. This property is what made possible the proof of `sem_equation_lift_reset` in the case `App`, with the lemmas `sem_node_mask_or` and `sem_node_absent`.

```
Fixpoint hold (v : value) (xs : stream svalue) (i : nat) : value :=
  match i with
  | 0   => v
  | S j => match xs j with
           | absent     => hold v xs j
           | present hv => hv
           end
  end.

Definition fby (v : value) (xs : stream svalue) (i : nat) : svalue :=
  match xs i with
  | absent => absent
  | _      => present (hold v xs i)
  end.
```

Figure 40: Definition of `hold` and `fby`, from `NLIndexedSemantics.v`

```
Fixpoint doreset (xs: stream svalue) (rs: stream bool) (i : nat)
  : bool :=
  if rs i then true
  else match i with
       | 0 => false
       | S j => match xs j with
                | absent => doreset xs rs j
                | present _ => false
                end
       end.

Definition reset (v : value) (xs : stream svalue) (rs : stream bool) (i : nat)
  : svalue :=
  match xs i with
  | absent => absent
  | present x => if (doreset xs rs i) then present v else present x
  end.
```

Figure 41: Definition of `doreset` and `reset`, from `NLIndexedSemantics.v`

```
Lemma fby_uninitialized (v : value) (xs : vstream) (i : nat) :
  (forall (j : nat), j < i -> xs j = absent) ->
    xs i = absent \/ fby v xs i = present v.
Proof.
  intros H.
  induction i.
  1: {
    unfold fby.
    destruct (xs 0); auto.
  }

  destruct (xs (S i)) eqn: Hxs; [ auto | ].
  right.
  unfold fby.
  rewrite Hxs.
  f_equal.
  apply hold_first_value.
  assumption.
Qed.
```

Figure 42: Lemma `fby_uninitialized`, from `NLIndexedSemantics.v`

Bibliography

[Bou+17] T. Bourke, L. Brun, P.-É. Dagand, X. Leroy, M. Pouzet, and L. Rieg, "A formally verified compiler for Lustre," *SIGPLAN Not.*, vol. 52, no. 6, pp. 586–601, Jun. 2017, doi: 10.1145/3140587.3062358.

[BBP19] T. Bourke, L. Brun, and M. Pouzet, "Mechanized semantics and verified compilation for a dataflow synchronous language with reset," *Models, Algorithms, and Proofs for ``Mechanized Semantics and Verified Compilation for a Dataflow Synchronous Language with Reset''*, vol. 4, no. Popl, pp. 1–29, Dec. 2019, doi: 10.1145/3371112.

[Bou+21] T. Bourke, P. Jeanmaire, B. Pesin, and M. Pouzet, "Verified Lustre Normalization with Node Subsampling," *ACM Trans. Embed. Comput. Syst.*, vol. 20, no. 5s, Sep. 2021, doi: 10.1145/3477041.

[BPP23] T. Bourke, B. Pesin, and M. Pouzet, "Verified Compilation of Synchronous Dataflow with State Machines," *ACM Transactions on Embedded Computing Systems*, vol. 22, no. 5s, pp. 1–26, Sep. 2023, doi: 10.1145/3608102.

[Hal+91] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud, "The synchronous data flow programming language LUSTRE," *Proceedings of the IEEE*, vol. 79, no. 9, pp. 1305–1320, 1991, doi: 10.1109/5.97300.

[Ler09] X. Leroy, "Formal verification of a realistic compiler," *Commun. ACM*, vol. 52, no. 7, Jul. 2009, doi: 10.1145/1538788.1538814.

[ICc24] Inria, CNRS, and contributors, "Reference manual of the Rocq Prover." 2024. [Online]. Available: https://rocq-prover.org/doc/V9.0.0/refman/index.html

[Bru20] L. Brun, "Mechanized Semantics and Verified Compilation for a Dataflow Synchronous Language with Reset," 2020. [Online]. Available: https://www.leliobrun.net/files/thesis.pdf

[Pes23] B. Pesin, "Verified Compilation of a Synchronous Dataglow Language with State Machines," 2023. [Online]. Available: https://velus.inria.fr/phd-pesin/thesis.pdf

[Jea24] P. Jeanmaire, "Une sémantique dénotationnelle pour un compilateur synchrone vérifié," 2024. [Online]. Available: https://velus.inria.fr/phd-jeanmaire/these.pdf

[KKM24] H. Kanabar, K. Korban, and M. O. Myreen, "Verified Inlining and Specialisation for PureCake," in *Programming Languages and Systems*, S. Weirich, Ed., Cham: Springer Nature Switzerland, 2024, pp. 275–301. doi: 10.1007/978-3-031-57267-8_11.

[Bie+08] D. Biernacki, J.-L. Colaço, G. Hamon, and M. Pouzet, "Clock-directed modular code generation for synchronous data-flow languages," *SIGPLAN Not.*, vol. 43, no. 7, pp. 121–130, Jun. 2008, doi: 10.1145/1379023.1375674.

[CPP17] J.-L. Colaco, B. Pagano, and M. Pouzet, "Scade 6: A Formal Language for Embedded Critical Software Development," in *Eleventh International Symposium on Theoretical Aspect of Software Engineering (TASE)*, Sophia Antipolis, France, Sep. 2017.

[CP99] P. Caspi and M. Pouzet, "Lucid Synchrone: une extension fonctionnelle de Lustre," in *Journées Francophones des Langages Applicatifs (JFLA)*, Avoriaz, France: Inria, Feb. 1999. [Online]. Available: https://hal.science/hal-01574464

[JRH20] E. Jahier, P. Raymond, and N. Halbwachs, "The Lustre V6 Reference Manual." 2020.

[HP00] G. Hamon and M. Pouzet, "Modular resetting of synchronous data-flow programs," pp. 289–300, 2000, doi: 10.1145/351268.351300.

[BBP18] T. Bourke, L. Brun, and M. Pouzet, "Towards a verified Lustre compiler with modular reset," in *Proceedings of the 21st International Workshop on Software and Compilers for Embedded Systems*, in Scopes '18. Sankt Goar, Germany: Association for Computing Machinery, 2018, pp. 14–17. doi: 10.1145/3207719.3207732.

[BP13] T. Bourke and M. Pouzet, "Zélus: A Synchronous Language with ODEs," in *16th International Conference on Hybrid Systems: Computation and Control (HSCC'13)*, Philadelphia, USA, Mar. 2013, pp. 113–118. [Online]. Available: http://www.di.ens.fr/~pouzet/bib/hscc13.pdf