Verified Lustre Normalization with Node Subsampling

TIMOTHY BOURKE, PAUL JEANMAIRE, BASILE PESIN, and MARC POUZET, Inria, France and École normale supérieure, CNRS, PSL University, France

Dataflow languages allow the specification of reactive systems by mutually recursive stream equations, functions, and boolean activation conditions called clocks. Lustre and Scade are dataflow languages for programming embedded systems. Dataflow programs are compiled by a succession of passes. This article focuses on the normalization pass which rewrites programs into the simpler form required for code generation.

Vélus is a compiler from a normalized form of Lustre to CompCert's Clight language. Its specification in the Coq interactive theorem prover includes an end-to-end correctness proof that the values prescribed by the dataflow semantics of source programs are produced by executions of generated assembly code. We describe how to extend Vélus with a normalization pass and to allow subsampled node inputs and outputs. We propose semantic definitions for the unrestricted language, divide normalization into three steps to facilitate proofs, adapt the clock type system to handle richer node definitions, and extend the end-to-end correctness theorem to incorporate the new features. The proofs require reasoning about the relation between static clock annotations and the presence and absence of values in the dynamic semantics. The generalization of node inputs requires adding a compiler pass to ensure the initialization of variables passed in function calls.

CCS Concepts: • **Software and its engineering → Formal language definitions**; **Software verification**; **Compilers**; • **Computer systems organization → Embedded software**.

Additional Key Words and Phrases: stream languages, verified compilation, interactive theorem proving

ACM Reference Format:
Timothy Bourke, Paul Jeanmaire, Basile Pesin, and Marc Pouzet. 2021. Verified Lustre Normalization with Node Subsampling. *ACM Trans. Embedd. Comput. Syst.* 1, 1, Article 1 (January 2021), 24 pages. https://doi.org/10.1145/3477041

1 INTRODUCTION

The development of critical embedded systems is often based on block-diagram models that permit the definition and interconnection of temporal behaviors. Such models can be understood as functions defined on streams of values. They are compiled down to imperative code whose cyclic execution calculates the values step-by-step. This idea is the basis of the academic Lustre language [11] and its industrial successor Scade 6 [8]. The Vélus project [5, 6] aims to formalize the central features of these languages, their type systems [9] and their compilation schemes [2], in the Coq interactive theorem prover [10]. It builds on the CompCert verified C compiler [14] to provide an end-to-end proof between a dataflow semantics for Lustre programs and the imperative semantics of the assembler generated from them.

This article appears as part of the ESWEEK-TECS special issue and was presented in the International Conference on Embedded Software (EMSOFT), 2021.

Authors' address: Timothy Bourke, timothy.bourke@inria.fr; Paul Jeanmaire, paul.jeanmaire@inria.fr; Basile Pesin, basile.pesin@inria.fr; Marc Pouzet, marc.pouzet@di.ens.fr, Inria, Paris, France, École normale supérieure, CNRS, PSL University, Paris, France.

1539-9087/2021/1-ART1 $15.00
https://doi.org/10.1145/3477041

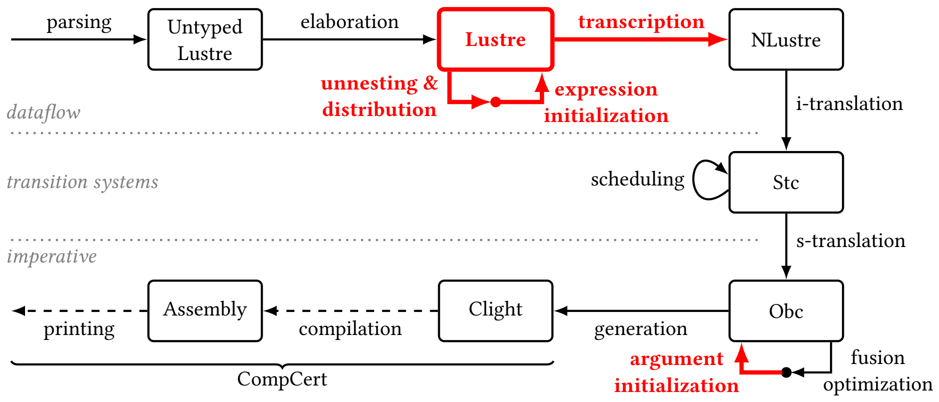

Fig. 1. Architecture of the Vélus compiler ; new elements in bold.

In this article, we present extensions to the Vélus compiler to enrich its input language beyond the normalized form treated in prior work. These extensions increase the practicality of the compiler and require us to address several interesting technical issues. In particular, we (a) formalize in an interactive theorem prover the semantics of an unrestricted input language that includes operators on lists of streams, and nodes with subsampled inputs and outputs; (b) prove the correctness of its clock type system; (c) implement the normalization of programs into the form required by the existing compiler and prove that this transformation is correct; and (d) modify the existing compiler to satisfy requirements imposed by CompCert on function arguments. The associated artefact is available at https://velus.inria.fr/emsoft2021.

The architecture of the Vélus compiler is shown in figure 1 with our modifications shown in bold. *Parsing* is followed by *elaboration* which adds typing annotations and produces a program in the abstract syntax of *Lustre*. Our normalization pass is divided into two source-to-source transformations, which are detailed in subsequent sections. The first does *unnesting* of expressions to place certain operators in their own equations and *distribution* of operators over lists. The second simplifies the form of certain operators to make *expression initialization* explicit and thereby simplify later transformations. Our *transcription* pass transforms programs into the abstract syntax of *NLustre*, which encodes the normalized form and is accepted by the rest of the compiler. We do not modify *i-translation* or *scheduling* [6]. The former transforms stream equations into named state *i*nstances acted upon by the step and reset constraints of the *Stc* transition system language. The latter orders the constraints in anticipation of the next pass. That pass is *s-translation*, which transforms the ordered constraints into a *s*equence of operations, of the *Obc* imperative language, on local and state variables. We make some changes to this pass and the target language, and add a new algorithm to ensure *argument initialization*. The *fusion optimization* and *generation* of *Clight* code are mostly unchanged [5].

Example of normalization. The program in figure 2 is typical of control operators [15, §1.2.2] provided by Scade and Simulink libraries. It comprises two stream functions, also called *nodes*. Normalisation transforms it into the program shown in figure 3. This example not only shows what normalization does, it also shows why it is useful. Until now, Vélus could not compile the more natural program at left; it would have to be manually rewritten into the form at right.

The first node is called count_down. It has two input streams called res and n, and an output stream cpt. Its body contains a single equation that defines cpt with a conditional expression that

```
node count_down(res : bool; n : int)
returns (cpt : int)
let
  cpt = if res then n else (n fby (cpt - 1));
tel

node rising_edge_retrigger(i : bool; n : int)
returns (o : bool)
var edge, ck : bool; v : int;
let
  edge = i and (false fby (not i));
  ck = edge or (false fby o);
  v = merge(ck;
        count_down((edge, n) when ck);
        0 when not ck);
  o = v > 0;
tel
```

Fig. 2. Example: source Lustre program

```
node count_down(res : bool; n : int)
returns (cpt : int)
var norm1$1, norm2$2 : int; norm2$1 : bool;
let
  norm2$1 = true fby false;
  norm2$2 = 0 fby (cpt - 1);
  norm1$1 = if norm2$1 then n else norm2$2;
  cpt = if res then n else norm1$1;
tel

node rising_edge_retrigger (i : bool; n : int)
returns (o : bool)
var edge, ck, norm1$1, norm1$2 : bool;
    v : int; elab$4 : int when ck;
let
  norm1$2 = false fby (not i);
  edge = i and norm1$2;
  norm1$1 = false fby o;
  ck = edge or norm1$1;
  elab$4 = count_down(edge when ck, n when ck);
  v = merge(ck; elab$4; 0 when not ck);
  o = v > 0;
tel
```

Fig. 3. Example: generated NLustre program

i	F	**T**	T	T	F	F	F	T	F	T	F	F	F	F	···
n	3	**3**	3	3	3	3	3	3	3	3	3	3	3	3	···
edge	F	**T**	F	F	F	F	F	T	F	T	F	F	F	F	···
ck	F	**T**	**T**	**T**	**T**	F	F	T	T	T	T	T	T	F	···
edge when ck		**T**	F	F	F			T	F	T	F	F	F		···
count_down(...)		**3**	**2**	**1**	**0**			3	2	3	2	1	0		···
v	0	**3**	**2**	**1**	**0**	0	0	3	2	3	2	1	0	0	···
o	F	**T**	**T**	**T**	**F**	F	F	T	T	T	T	T	F	F	···

Fig. 4. Example trace of the rising_edge_retrigger node

takes the value of n when res is true and the value of the subexpression n fby (cpt − 1) otherwise. The fby operator is equal to the first value from its left-hand argument "followed by" the stream of values from its right-hand argument: it gives an initialized delay. Here the result is a stream cpt that counts backward taking the current value of n initially and whenever res is true.

The normalization of count_down gives the node of the same name in figure 3. In the equation for cpt, the fby has been replaced by the new local variable norm1$1. An initialization variable norm2$1 has been added and used in the definition of norm1$1 to take an initial value from n and all other values from norm2$2. Importantly, every fby is initialized by a constant.

The second node, rising_edge_retrigger, waits for a rising edge, that is, an F followed directly by a T, on its input i and only then emits the value T *n* times on its only output. Figure 4 shows an example trace. The bold values highlight the first rising edge on i, the value of n at that instant and, on the last line, the response of the node on o.

The node body comprises four equations, three of which define local variables. The first detects rising edges by comparing successive values of i. The equation for ck determines when a countdown is active: either a rising edge was detected or the previous output was true. The equation for v uses the two sampling operators when and merge. A when filters one or more streams according to the value of its second argument. For example, the value of edge when ck is *present* with the value of edge at instants where ck is true—as the trace shows. At other instants it is *absent*—the column in the trace is empty. In the expression count_down((edge, n) when ck), the first node is applied to two streams filtered by ck. This instance is thus slower than its context. A merge combines two streams. If the first argument is true, the value comes from the second argument, which must be present and the third argument must be absent, and inversely when the first argument is false. The value of v comes from the instance of count_down when ck is true and from a stream of zeros otherwise. The last equation ensures a true output only when the countdown is strictly positive.

The result of normalizing rising_edge_retrigger is shown in figure 3. In the equation defining v, the node instance has been unnested and replaced by the variable elab$4. In the defining expression of this new variable, the when operator has been distributed over the list of arguments (edge, n). In general, distributivity is also applied to arguments of merge, if-then-else, and fby.

The fbys in the definitions of edge and ck have been unnested, but each is already initialized by a constant, so further simplifications are unnecessary.

Subsampled inputs and outputs. In the example, the inputs and outputs of the only node instance, count_down((edge, n) when ck), are all sampled on the *base clock* of the instance (ck = true). It is occasionally useful to be able to define and instantiate nodes where some inputs and outputs are sampled less often than others. A simple but practical example is the following well-initialized version [15, §1.2.2] of the original Lustre's [11] problematic current operator.

```
node current(d : int; ck : bool; x : int when ck) returns (y : int);
let
  y = merge(ck; x; (d fby y) when not ck);
tel
```

The signals d, ck, and y are all sampled on the base clock, but x is only required when ck is true. Until now, Vélus did not accept such programs.

The clocks of Lustre are best seen as primitives of a core language into which more convenient control structures, like activation blocks and hierarchical automata, are compiled [8, §V]. In the same way, if at a smaller scale, the example above shows that node subsampling permits the current operator to be specified as a library function, whereas it would otherwise have to be added to the core language as a primitive. The treatment of node subsampling also removes a somewhat artificial restriction from the semantic model even if, all in all, the source language remains tied to the compilation schema of one clock per equation and one function call per node instance.

Combining normalization and subsampling. While normalization and node subsampling can be treated independently of one another, there are two good reasons for addressing them together. First, they both contribute to providing a source language that allows abstraction and composition of causal stream equations without arbitrary syntactic restrictions. That is, programmers can encapsulate a behaviour, as in the current node above, and then freely combine it with other expressions. This lifting of syntactic restrictions also allows a more streamlined semantic model: normalization permits a simpler treatment of equations and subsampling removes a constraint on node applications. Second, showing normalization correctness requires reasoning about the clock system and with node subsampling the invariants are subtler and the proofs more difficult.

$$\begin{array}{rcl}
e & ::= & c \\
 & | & x \\
 & | & \diamond\, e \\
 & | & e \oplus e \\
 & | & e^{+}\ \texttt{fby}\ e^{+} \\
 & | & e^{+}\ \texttt{when}\ x \\
 & | & \texttt{merge(}\ x\ ;\ e^{+}\ ;\ e^{+}\ \texttt{)} \\
 & | & \texttt{if}\ e\ \texttt{then}\ e^{+}\ \texttt{else}\ e^{+} \\
 & | & f\ \texttt{(}\ e^{+}\ \texttt{)} \\
eq & ::= & x^{+} = e^{+}\ ;
\end{array}$$

Fig. 5. Lustre: equations

$$\begin{array}{rcl}
e & ::= & c \\
 & | & x \\
 & | & \diamond\, e \\
 & | & e \oplus e \\
 & | & e\ \texttt{when}\ x \\
ce & ::= & e \\
 & | & \texttt{merge(}\ x\ ;\ ce\ ;\ ce\ \texttt{)} \\
 & | & \texttt{if}\ e\ \texttt{then}\ ce\ \texttt{else}\ ce \\
eq & ::= & x = ce \\
 & | & x = c\ \texttt{fby}\ e \\
 & | & x^{+} = f\ \texttt{(}\ e^{+}\ \texttt{)}
\end{array}$$

Fig. 6. NLustre: equations

$$\begin{array}{rcl}
n & ::= & \texttt{node}\ f\ \texttt{(}\ d^{+}\ \texttt{)}\ \texttt{returns}\ \texttt{(}\ d^{+}\ \texttt{)} \\
 & & (\texttt{var}\ d^{+}\ ;)^{?} \\
 & & \texttt{let}\ eq^{*}\ \texttt{tel} \\
d & ::= & x^{ck}_{ty} \\
G & ::= & n^{+}
\end{array}$$

Fig. 7. Nodes and Programs

2 LUSTRE: SYNTAX, SEMANTICS, AND CAUSALITY IN COQ

We now present our formalization of Lustre in Coq and the target syntax of the existing compiler.

2.1 Syntaxes

Figure 5 shows the syntax of expressions and equations in the source language. The class of expressions includes constants c, variables x, unary operators $\diamond$, and binary operators $\oplus$. The fby, when, merge, and if operators may be applied to lists of expressions. An expression may thus produce several streams. This allows node instances $f(e^{+})$, possibly having multiple outputs, to be used freely in other expressions. An equation associates a list of variables with a list of expressions.

The syntax of nodes and programs is given in figure 7. A node receives one or more inputs, produces one or more outputs, and, optionally, declares local variables. Its body contains a list of equations. A program is a list of one or more nodes.

In the abstract syntax, expressions are annotated by their type and clock type. Where necessary, we will write e^{ck} to denote an expression e annotated with a clock type ck. In Lustre, the clock types are constrained by typing rules [9] to guarantee that all streams in a program are synchronous and thus computable in bounded memory. For instance, the operands of a binary arithmetic operation must have the same clock type while those of a merge must be complementary, as is the case in the example of figure 2, at lines 13 to 15. An expression like x + (x when c) is not synchronous because its correct implementation requires a buffer whose size depends on the value of c, which cannot, in general, be determined statically.

The syntax of clock types is $ck ::= \bullet \mid ck \text{ on } x \mid ck \text{ onot } x$. The first case is for the base clock relative to a context. The other two cases classify the sampling of a stream according to a boolean variable x. For example, given an expression e with clock type $\bullet$ and a boolean variable x with the same clock, the expression e when x has clock $\bullet$ on x.

Earlier versions of Vélus only accept the normalized language shown in figure 6. Its expressions are classified into simple expressions e and control expressions ce, neither of which allows lists of expressions. There are three types of equations: those defined by control expressions, those defined by a fby, and those defined by a node instance. Every correct syntactic element, except node instances, corresponds to exactly one stream. The normalized form is less convenient for programmers but is important in the compilation scheme because it isolates fbys and node instances in anticipation of their treatment in the generation of imperative code.

The objective of normalisation is thus to transform a program from the syntax of figure 5 into the syntax of figure 6 while preserving its overall input/output semantics.

$$\frac{s \equiv \mathsf{const}\ bs\ [\![\mathtt{c}]\!]}{G, H, bs \vdash \mathtt{c} \Downarrow [s]}$$

$$\mathsf{const}\ (\mathrm{T} \cdot bs)\ c = \langle c \rangle \cdot \mathsf{const}\ bs\ c$$
$$\mathsf{const}\ (\mathrm{F} \cdot bs)\ c = \langle\rangle \cdot \mathsf{const}\ bs\ c$$

(a) Constants

$$\frac{G, H, bs \vdash \boldsymbol{e} \Downarrow H(\boldsymbol{x})}{G, H, bs \vdash \boldsymbol{x} = \boldsymbol{e}}$$

(b) Equations

$$\frac{H(x) = s \qquad G, H, bs \vdash \boldsymbol{e_t} \Downarrow \boldsymbol{ts} \qquad G, H, bs \vdash \boldsymbol{e_f} \Downarrow \boldsymbol{fs} \qquad \mathsf{merge}\ s\ \boldsymbol{ts}\ \boldsymbol{fs} \doteq \boldsymbol{vs}}{G, H, bs \vdash \mathtt{merge(}\ x\mathtt{;}\ \boldsymbol{e_t}\mathtt{;}\ \boldsymbol{e_f}\mathtt{)} \Downarrow \boldsymbol{vs}}$$

$$\frac{\mathsf{merge}\ cs\ ts\ fs \doteq vs}{\mathsf{merge}\ (\langle\rangle \cdot cs)\ (\langle\rangle \cdot ts)\ (\langle\rangle \cdot fs) \doteq \langle\rangle \cdot vs}$$

$$\frac{\mathsf{merge}\ cs\ ts\ fs \doteq vs}{\mathsf{merge}\ (\langle \mathrm{T} \rangle \cdot cs)\ (\langle t \rangle \cdot ts)\ (\langle\rangle \cdot fs) \doteq \langle t \rangle \cdot vs}$$

$$\frac{\mathsf{merge}\ cs\ ts\ fs \doteq vs}{\mathsf{merge}\ (\langle \mathrm{F} \rangle \cdot cs)\ (\langle\rangle \cdot ts)\ (\langle f \rangle \cdot fs) \doteq \langle f \rangle \cdot vs}$$

(c) Rule and operator for merge

$$\frac{\mathsf{node}(G, f) \doteq n \qquad H(n.\mathbf{in}) = \boldsymbol{xs} \qquad H(n.\mathbf{out}) = \boldsymbol{ys} \qquad \forall eq \in n.\mathbf{eqs},\ G, H, (\mathsf{base\text{-}of}\ \boldsymbol{xs}) \vdash eq}{G \vdash f(\boldsymbol{xs}) \Downarrow \boldsymbol{ys}}$$

(d) Nodes

Fig. 8. Lustre: Selected semantic rules and operators

2.2 Semantics

The semantic model for a node relates lists of input streams and lists of output streams. In our formalization, streams of values are modeled by the coinductive type stream svalue, where svalue is a *synchronous value* with two constructors: $\langle v \rangle$ for the presence of a boolean, integer, or floating-point value, and $\langle\rangle$ for the absence of value at an instant.

The predicate $G, H, bs \vdash e \Downarrow \boldsymbol{vs}$ relates, in a program G, for a *history* H and a *base clock* bs, the expression e to the list of streams $\boldsymbol{vs}$. The history H associates each variable to a stream. Together with the encoding of presence and absence, it essentially represents the kind of trace shown in figure 4. The base clock bs is a stream of booleans that encodes the activation rate of the context.

We use bold characters to represent lists. So, $G, H, bs \vdash \boldsymbol{es} \Downarrow \boldsymbol{vs}$ is the lifting of the previous predicate to a list of expressions $\boldsymbol{es}$ with concatenation of individual components into the list of streams $\boldsymbol{vs}$.

The predicates for expressions, equations, and nodes are defined by mutual induction over the syntax from figures 5 and 7. Coinductive functions and relations are used within the cases to relate streams, typically via point-wise applications to lists. A selection of definitions is presented in figure 8 and explained below. We introduce other definitions as needed in the following sections.

The semantic rule for constants, figure 8a, exemplifies the general idea. It associates a constant c to a list containing a single stream s. This stream is equivalent to the application of the semantic operator const to the base clock and the value assigned by CompCert to the constant, denoted $[\![\cdot]\!]$. Since const is a total function it can be defined by coinduction: the value of the base clock at each instant determines whether the constant is present or absent.

The semantic rule and operator for merge are presented in figure 8c. The rule requires that the guard variable, x, be associated in H with a stream s and that the lists of expressions $\boldsymbol{e_t}$ and $\boldsymbol{e_f}$ evaluate, respectively, to lists of streams $\boldsymbol{ts}$ and $\boldsymbol{fs}$. A merge operator on s is lifted pointwise over $\boldsymbol{ts}$,

$\boldsymbol{fs}$, and $\boldsymbol{vs}$. This operator is defined as a coinductive[1] relation between the guard stream, two branch streams, and a result stream. There are three cases for the heads of the streams: (i) all four simultaneously absent, (ii) the guard stream present with value T, the second stream present, the third stream absent, and the last stream taking its value from the second stream, and (iii) the guard stream present with value F, the second stream absent, the third stream present, and the last stream taking its value from the third stream. The operator is not defined if the guard value is not boolean or if the streams are not synchronized as described. The '$\doteq$' symbol before the result stream is just for readability, it has no formal meaning.

The predicate for equations is denoted $G, H, bs \vdash eq$ and defined by the single rule shown in figure 8b. An equation constrains a history H by requiring that it maps each variable at left to the corresponding element in the list of streams associated with the expression at right.

The predicate for nodes $G \vdash f(\boldsymbol{xs}) \Downarrow \boldsymbol{ys}$ states that, in a program G, the node called f relates a list of input streams $\boldsymbol{xs}$ to a list of output streams $\boldsymbol{ys}$. It requires, figure 8d, that the definition of f in G is a node n and that there exists a history H that (i) associates the input variables $n.\mathbf{in}$ to the input streams, (ii) associates the output variables $n.\mathbf{out}$ to the output streams, and (iii) satisfies the constraints imposed by each of the node equations, $n.\mathbf{eqs}$. Importantly, the universal quantification over equations means that the semantics is invariant to their reordering. The base-of operator gives a base clock that is true at an instant iff at least one of the input streams is present.

The main characteristics of the model we implemented in Coq are visible in this partial presentation and can be compared with formal pen-and-paper models. The rules associating expressions with streams are standard. The operators give a *synchronous semantics* where the presence and absence of values is explicit, as opposed to a *Kahn semantics* where it is not. Their definitions closely resemble those of, for instance, Colaço and Pouzet [9, §3.2], with the difference that our model does not accept finite streams whereas the latter model uses them in a least fix point construction. This resemblance is reassuring: the compiler correctness theorem guarantees that the generated code correctly implements the semantics specified in Coq, but the aptness of the definitions can only be judged against the accepted meaning of the source language. The predicates in our definitions describe what a program means but do not prescribe how it should be evaluated or compiled. That said, we have yet to prove that they are satisfied by valid programs, that is, those accepted by the compiler and that do not perform illegal operations.

Earlier definitions [5, 6] follow the rigid structure imposed by the normalized syntax and reasoning focuses on the three cases for equations: node instantiations, fby equations, and control expressions. In the new model, the cases for expressions are central. Otherwise, a significant difference is the elimination of a technical "respects-clock" requirement in the predicate for nodes, cf. [6, figure 6] and figure 8d. We develop this point in section 3.

Simpler alternatives? An alternative to the approach adopted here is to restrict node instantiations to the roots of equation expressions and, since they would then be less useful, to forbid lists of expressions everywhere but in node arguments. The resulting language would be less convenient for writing programs but easier to handle in algorithms, semantic models, and proofs. Each stream would be explicitly named and each subexpression would represent a single stream.

Another alternative to the idiosyncratic treatment of argument lists and implicit flattening is to use tuples [15]. This would factorize but not eliminate the treatment of lists of streams.

2.3 Causality

Many proofs about the normalized language are based on a "well-scheduled" predicate [5, §3.2]. Basically, the normalized equations are ordered so that the equation defining a variable comes

[1]Coinductive rules are written with a double horizontal bar to distinguish them from rules defined inductively over terms.

$$\frac{}{\mathsf{IsFreeLeft}\ x\ 0\ x} \qquad \frac{\mathsf{IsFreeLeftList}\ x\ k\ \boldsymbol{e0s}}{\mathsf{IsFreeLeft}\ x\ k\ (\boldsymbol{e0s}\ \mathtt{fby}\ \boldsymbol{es})} \qquad \frac{k < \mathsf{numstreams}\ \ (f(\boldsymbol{es})) \quad \exists k'\ e,\ e \in \boldsymbol{es} \land \mathsf{IsFreeLeft}\ x\ k'\ e}{\mathsf{IsFreeLeft}\ x\ k\ (f(\boldsymbol{es}))}$$

Fig. 9. Selected IsFreeLeft rules

before, or after in the case of a fby-equation, any other equation in which it is used. Many properties of NLustre can be conveniently shown by induction on a list of well-scheduled equations.

In Lustre, however, this technique no longer suffices. Consider, for instance, the simple equation x, y = (1, x). It has a valid solution, $x = \langle 1\rangle \cdot \langle 1\rangle \cdots$ and $y = \langle 1\rangle \cdot \langle 1\rangle \cdots$, but a naive application of schedulability would determine that both x and y (at left in the equation) depend on x (free in an expression at right), which is forbidden. The fact that expressions may be freely nested also complicates matters, since it is no longer possible to simply partition variables into those defined directly and those defined via a fby.

These are old problems and the solution is well known [11, §III.A]: construct a graph of variable dependencies, ignoring the expressions at right of a fby, and ensure that it does not contain any cycles. The question is how to translate this technique into an interactive theorem prover.

Our solution begins with the predicates IsFreeLeft $x\ k\ e$, which signifies that the variable x is required for the kth stream associated with e, and IsFreeLeftList $x\ k\ es$, which refers to the kth stream associated with a list of expressions. The cases for variables, fbys, and node instances are shown in figure 9. Importantly, the rule for fbys does not consider the subexpressions at right, *es*. Now we can state that a variable x *depends on* y only if there is an equation $xs = es$ where x is the kth element of xs and IsFreeLeft $y\ k\ es$. It is straightforward to implement a function that takes a list of equations and builds a map from variables to the sets of variables on which they depend. It is trickier in Coq to implement the standard depth-first search algorithm to look for cycles in this data structure because the algorithm's guaranteed termination cannot be deduced by a simple syntactic analysis. We prove that if our algorithm succeeds then there exists a *directed acyclic graph* of the inverse dependency relation. Such a graph is defined inductively by the following three rules.

$$\frac{}{\langle \emptyset, \emptyset \rangle} \qquad \frac{\langle V, E\rangle}{\langle V \cup \{x\}, E\rangle} \qquad \frac{\langle V, E\rangle \quad x, y \in V \quad x \neq y \quad y \not\rightarrow^{*}_{E} x}{\langle V, E \cup \{x \rightarrow y\}\rangle}$$

A central property of such graphs is that their vertices can be ordered topologically. We prove that for any $\langle V, E\rangle$ there is a list of all elements in V that satisfies the following predicate.

$$\frac{}{\mathsf{TopoOrder}\ \langle V, E\rangle\ []} \qquad \frac{x \in V \quad \neg\mathsf{In}\ x\ xs \quad (\forall w,\ w \rightarrow^{*}_{E} x \implies \mathsf{In}\ w\ xs)}{\mathsf{TopoOrder}\ \langle V, E\rangle\ (x :: xs)}$$

Putting all of this together: A node is *causal* only if its input, local, and output variables form the vertices of an acyclic graph with an edge from x to y whenever either x depends on y, or y is free in the clock type of x. We exploit the existence of a topological ordering to prove an induction principle for causal nodes where the hypothesis holds for all variables that are free at left in an expression. Our principle is weak in that it gives no information for the expressions at right of a fby, but it suffices for the proof of normalization correctness because the typing rule for fby requires that its subexpressions have the same clock.

The causality of a program could be checked directly after elaboration but the resulting property is not needed to reason about unnesting and distribution. By placing the check just before expression

$$\frac{\mathsf{fby}\ xs\ ys \doteq vs}{\mathsf{fby}\ (\langle\rangle \cdot xs)\ (\langle\rangle \cdot ys) \doteq \langle\rangle \cdot vs} \qquad \frac{\mathsf{fby}_1\ y\ xs\ ys \doteq vs}{\mathsf{fby}\ (\langle x\rangle \cdot xs)\ (\langle y\rangle \cdot ys) \doteq \langle x\rangle \cdot vs}$$

$$\frac{\mathsf{fby}_1\ v\ xs\ ys \doteq vs}{\mathsf{fby}_1\ v\ (\langle\rangle \cdot xs)\ (\langle\rangle \cdot ys) \doteq \langle\rangle \cdot vs} \qquad \frac{\mathsf{fby}_1\ y\ xs\ ys \doteq vs}{\mathsf{fby}_1\ v\ (\langle x\rangle \cdot xs)\ (\langle y\rangle \cdot ys) \doteq \langle v\rangle \cdot vs}$$

Fig. 10. The fby semantic operator (coinductive predicate) for Lustre

$$\begin{aligned}\mathsf{fby}_{\mathsf{NL}}\ v\ (\langle\rangle \cdot ys) &= \langle\rangle \cdot (\mathsf{fby}_{\mathsf{NL}}\ v\ ys)\\ \mathsf{fby}_{\mathsf{NL}}\ v\ (\langle y\rangle \cdot ys) &= \langle v\rangle \cdot (\mathsf{fby}_{\mathsf{NL}}\ y\ ys)\end{aligned}$$

Fig. 11. The $\mathsf{fby}_{\mathsf{NL}}$ semantic operator (coinductive function) for NLustre

initialization, whose proof does require the property, we avoid having to show that unnesting and distribution preserves causality.

3 TRANSCRIPTION AND CLOCK SYSTEM CORRECTNESS

Transcription follows normalization. The algorithm is trivial. It simply transforms an already normalized program from the Lustre syntax (figure 5) into the NLustre syntax (figure 6). The difficulty is that proving semantics preservation requires also proving that the clock type system is correct. The latter property holds for any Lustre program, normalized or not, and is also required when reasoning about earlier passes. We present it here first where the motivation is most evident.

3.1 The fby Operator and Stream Alignment

In Lustre, the left and right elements of a fby are expressions that produce streams. The usual formalization, see for instance [9, figure 2], is expressed in Coq by the pair of coinductive relations shown in figure 10. The fby relation waits for initial values on incoming streams *xs* and *ys*, passes the one from *xs* directly, and memorizes the other as the first argument of fby_1. The fby_1 relation holds the memorized value until subsequent values are present on the input streams and then passes it to the output while memorizing the next value. Only the first value of *xs* is ever used, but, importantly, the relation enforces the synchronization of its presence and absence with *ys*.

In NLustre, the left element of a fby is a constant that is not interpreted as a stream but rather as a static initialization parameter. This facilitates code generation by providing an initial value for the generated state variable. The semantic operator is total and can thus be expressed in Coq as the coinductive function of type value × stream svalue → stream svalue shown in figure 11.

The differences between the semantic operators for fby in Lustre and NLustre spread into the other constructions. Consider, in particular, the equation x = true fby (not x) that is valid in both languages. In Lustre, the true is a constant expression and presence or absence of the associated stream is determined by the base clock. For a base clock that is always true, for example, the stream is $\langle T\rangle \cdot \langle T\rangle \cdot \langle T\rangle \cdots$. Given the definitions for fby and equations, the only possible value for x is $\langle T\rangle \cdot \langle F\rangle \cdot \langle T\rangle \cdot \langle F\rangle \cdots$. In NLustre, on the other hand, the true is just an initial parameter for $\mathsf{fby}_{\mathsf{NL}}$ and does not impose any constraints on the stream for x. As a consequence, any stream of the form $(\langle\rangle^* \cdot \langle T\rangle \cdot \langle\rangle^* \cdot \langle F\rangle)^\omega$ would be valid.

Such nondeterminism is unwanted. Ideally the source semantics reflects what the generated code actually does. This is why the existing Vélus compiler imposes the respects-clock constraint in the

$$\dfrac{\begin{array}{c}\mathsf{tl}\,H, \mathsf{tl}\,bs \vdash e^{ck} \Downarrow s \\ H, bs \vdash ck \Downarrow \mathrm{T} \cdot b \qquad H, bs \vdash e \Downarrow \langle v \rangle \cdot s\end{array}}{H, bs \vdash e^{ck} \Downarrow \langle v \rangle \cdot s} \qquad \dfrac{\begin{array}{c}\mathsf{tl}\,H, \mathsf{tl}\,bs \vdash e^{ck} \Downarrow s \\ H, bs \vdash ck \Downarrow \mathrm{F} \cdot b \qquad H, bs \vdash e \Downarrow \langle\rangle \cdot s\end{array}}{H, bs \vdash e^{ck} \Downarrow \langle\rangle \cdot s}$$

Fig. 12. Alignment between a clock (stream bool) and an expression (stream svalue)

NLustre node semantics. The constraint requires that the streams associated to certain expressions be synchronized with their clock types. This means defining a semantic predicate $H, bs \vdash ck \Downarrow b$ to associate a clock type with a boolean stream. The clock type $\bullet$ is associated with the base clock of the context. The stream associated with ck on x is T only if the stream for the subclock ck is T and the stream for x is $\langle \mathrm{T} \rangle$. It is $\langle \mathrm{F} \rangle$ if the stream for ck is F and the stream for x is $\langle\rangle$, or if the stream for ck is T and the stream for x is $\langle \mathrm{F} \rangle$. Otherwise it is undefined. The stream for ck onot x is defined similarly. The respects-clock predicate presupposes the *alignment* of certain expressions and their clock types. The formal definition of alignment is shown in figure 12. There are two cases for an expression e with clock ck. If the expression is present with some value, then it is aligned only if the clock is T. If the expression is absent, then it is aligned only if the clock is F. The tl operator destructs a stream and returns its tail. It is lifted implicitly to environments in the obvious way.

In Lustre, on the contrary, the clock types are not interpreted in the semantic model. Rather than assume the alignment property by explicitly stating it as a requirement in the semantic rules, we prove that is a consequence of those rules together with the rules for clock typing.

3.2 Correctness of the Clock System

The semantics of NLustre in the existing compiler mandates that source programs satisfy the alignment property. In addition to eliminating a source of nondeterminism, this property gives information on presence and absence that is required by the correctness proof of the translation to imperative code. In this work, rather than assume this property, we prove that it is implied by the semantic model presented in section 2.2 for any well-clocked, causal Lustre program that has a semantics. This also solves the main difficulty in proving the transcription pass correct.

Theorem 3.1. *Given a causal, well-clocked Lustre node with signature*

$$\texttt{node f (}x_1^{ck_1}, \ldots, x_n^{ck_n}\texttt{) returns (}y_1^{ck'_1}, \ldots, y_m^{ck'_m}\texttt{)}$$

and semantics f($s_1, \ldots, s_n$) $\Downarrow s'_1, \ldots, s'_m$, with $bs = \texttt{base-of}(s_1, \ldots, s_n)$, in any environment H in which input variables are associated and aligned with input streams, $H, bs \vdash x_1^{ck_1} \Downarrow s_1, \ldots, x_n^{ck_n} \Downarrow s_n$, and output variables are associated with output streams, $H \vdash y_1 \Downarrow s'_1, \ldots, y_m \Downarrow s'_m$, those output streams are aligned with the corresponding output clock types, $H, bs \vdash y_1^{ck'_1} \Downarrow s'_1, \ldots, y_m^{ck'_m} \Downarrow s'_m$.

The arbitrary environment, H, in the correctness theorem allows for the interpretation of clocks which may depend on input and output variables. At each node, we require that the input streams are aligned. This assumption is satisfied inductively for a program's internal nodes and automatically for its main node whose inputs must be supplied in every cycle. The lemma attests the correctness of the clock type system, for all Lustre programs, by showing that the static annotations and the semantic model coincide.

The proof of theorem 3.1 follows by mutual induction on the syntax of expressions and equations using the principle introduced in section 2.3. Constants are aligned with the base clock of the enclosing node by definition. For variables, an invariant is needed: if x is declared with clock type ck and associated in the environment H with the stream s, then s is aligned with ck. For inputs, this invariant is true by assumption; for other variables it is given by the induction hypothesis. The case

$$
\begin{aligned}
\lfloor \mathsf{c} \rfloor &= ([\mathsf{c}], [\,]) \\
\lfloor \mathsf{x} \rfloor &= ([\mathsf{x}], [\,]) \\
\lfloor e_1 \oplus e_2 \rfloor &= ([e_1'], \boldsymbol{eqs}_1') \leftarrow \lfloor e_1 \rfloor \\
&\quad\ ([e_2'], \boldsymbol{eqs}_2') \leftarrow \lfloor e_2 \rfloor \\
&\quad\ ([e_1' \oplus e_2'], \boldsymbol{eqs}_1' \cup \boldsymbol{eqs}_2') \\
\lfloor (e_1, \ldots, e_n) \text{ when } b \rfloor &= ([e_1', \ldots, e_m'], \boldsymbol{eqs}') \leftarrow \lfloor e_1, \ldots, e_n \rfloor \\
&\quad\ ([e_1' \text{ when } b, \ldots, e_m' \text{ when } b], \boldsymbol{eqs}') \\
\lfloor (e_1, \ldots, e_n) \text{ fby } (f_1, \ldots, f_m) \rfloor &= ([e_1', \ldots, e_k'], \boldsymbol{eqs}_1') \leftarrow \lfloor e_1, \ldots, e_n \rfloor \\
&\quad\ ([f_1', \ldots, f_k'], \boldsymbol{eqs}_2') \leftarrow \lfloor f_1, \ldots, f_m \rfloor \\
&\quad\ ([x_1, \ldots, x_k], [x_1 = e_1' \text{ fby } f_1', \ldots, x_k = e_k' \text{ fby } f_k'] \cup \boldsymbol{eqs}_1' \cup \boldsymbol{eqs}_2') \\
\lfloor f(e_1, \ldots, e_n) \rfloor &= ([e_1', \ldots, e_m'], \boldsymbol{eqs}') \leftarrow \lfloor e_1, \ldots, e_n \rfloor \\
&\quad\ ([x_1, \ldots, x_k], [(x_1, \ldots x_k) = f(e_1', \ldots, e_m')] \cup \boldsymbol{eqs}')
\end{aligned}
$$

Fig. 13. Unnesting and distribution of expressions, selected cases

for fbys is delicate as the hypothesis only applies to the expressions at left. We exploit the facts that the semantic rule, figure 10, guarantees the synchronization of all input and output streams, and that the fby expression and its two subexpressions all have the same clock type. In the case for node applications, we treat the possibility of unnamed output streams by extending the environment with bindings for anonymous variables.

4 UNNESTING AND DISTRIBUTION

We now present the first of the two normalization passes that transform a Lustre program into the subset treated by the transcription pass of the previous section. It both unnests instantiations and fbys, putting them in distinct equations, and distributes the fby, when, merge and if constructions over their argument lists. The result is a program where each expression represents a single stream, except for node instances whose every output stream is nevertheless assigned directly to a variable.

4.1 Unnesting and Distribution Algorithm

The unnesting and distribution of an expression e is denoted $\lfloor e \rfloor = ([e_1', ..., e_m'], \boldsymbol{eqs})$. It produces a list of expressions due to distribution, for example, the expression (edge, n) when ck, from figure 2, becomes (edge when ck, n when ck). It also produces a list of equations due to unnesting. Folding this operation over the expressions $e_1, \ldots, e_n$ is denoted $([e_1', \ldots, e_m'], \boldsymbol{eqs}') \leftarrow \lfloor e_1, \ldots, e_n \rfloor$, where $e_1', \ldots, e_m'$ and $\boldsymbol{eqs}'$ are concatenations of the individual results.

Several cases of this function are presented in figure 13. Constants and variables are not changed and do not add any new equations. Binary operators are treated recursively and the new equations generated for each subexpression are concatenated in the result. The expressions at left in a when are treated recursively and the original when is distributed over the resulting expressions. For fbys, the recursively generated expressions are combined pair-by-pair into new equations defining fresh variables, and the list of those variables is returned together with the new equations. The case for node instantiations, $\lfloor f(e_1, \ldots, e_n) \rfloor$, is similar but does not require distribution after unnesting. We do not show the cases for the merge or if, as they are similar to that of the fby. In the full definition, we add special cases for subexpressions where unnesting is not required by the grammar of figure 6. The aim is to minimize transformations to the original program. For instance, if a node instance already appears directly in an equation, there is no need to introduce a new equation, and likewise for a fby that produces a single stream. In fact, we prove that this function is idempotent: for any program G, $\lfloor\lfloor G\rfloor\rfloor = \lfloor G\rfloor$.

Generating fresh variables. Unnesting subexpressions requires the introduction of new local variables. For this reason, the function that performs unnesting and distribution is structured using a state monad. It manipulates a state of type `fresh_st` = (`ident` × `list` (`ident` × (`ty` × `ck`))), where the first component is used to generate the next new identifier and the second tracks those that have already been introduced together with their types and clock types.

The type `ident` is a synonym for the positive integers. For compatibility with later CompCert stages, identifiers are registered in an external mutable table together with their string representations. We cannot, however, simply axiomatize a function "`newident : unit → ident`" to return fresh identifiers since, for instance, the valid proposition `newident` () = `newident` () would reduce to the inconsistent one $n = n + 1$. Instead we require an external function that maps two identifiers to a third identifier: `gensym : ident → ident → ident`. The returned identifier is associated in the external table with the concatenation of the string associated with the first argument, the "`$`" character, and the second argument rendered as a decimal string. A runtime check ensures that the first argument does not already contain a "`$`". Proofs rely on two axiomatized properties of gensym: it produces identifiers that differ from those in source files, which the lexer prevents from containing "`$`", and if $x \neq x'$ or $y \neq y'$ then `gensym` $x\ y \neq$ `gensym` $x'\ y'$. Each compilation pass generates identifiers using a different prefix. Several examples can be found in figure 3: `elab$4`, `norm1$1`, `norm2$1`, and `norm2$2`.

The state monad is a function taking a state as input and returning a result and updated state. We abstract it with a type constructor for a result type A: `Fresh A` = `fresh_st` → (`A` × `fresh_st`). Such functions are built from the standard monadic operators `ret` : `A` → `Fresh A`, which returns the given value and passes the input state unchanged, and `bind` : `Fresh A` → (`A` → `Fresh B`) → `B`, which sequences two functions by passing the return value and output state of the first to the second. We additionally equip the state monad with the function fresh_ident : (ty × ck) → Fresh ident that returns an identifier produced by applying gensym to the current prefix and the internal counter. In the updated state, the counter is incremented and the returned identifier is associated with the given type and clock type in the internal list. After applying the unnesting and distribution function to the equations within a node, the internal list is extracted and appended to the local variable list.

4.2 Unnesting and Distribution Correctness

THEOREM 4.1. *The unnesting and distribution function preserves the input/output semantics of any well-typed and well-clocked program G:* $\forall f\ \boldsymbol{xs}\ \boldsymbol{ys},\ G \vdash f(\boldsymbol{xs}) \Downarrow \boldsymbol{ys} \implies \lfloor G\rfloor \vdash f(\boldsymbol{xs}) \Downarrow \boldsymbol{ys}$.

The final theorem is stated in terms of a whole program G and builds on a lemma for lists of equations, but the core of the correctness proof focuses on the unnesting and distribution of expressions: $\lfloor e\rfloor = (es', \boldsymbol{eqs'})$. If the semantics of e, relative to an environment H, is a list of streams $\boldsymbol{vs}$, that is, $G, H, bs \vdash e \Downarrow \boldsymbol{vs}$, then we must show that the produced expressions $\boldsymbol{es'}$ have the same semantics after extending H to satisfy the produced equations $\boldsymbol{eqs'}$. That is, we must show

$$\exists H',\ H \sqsubseteq H' \ \wedge\ G, H', bs \vdash \boldsymbol{eqs'} \ \wedge\ G, H', bs \vdash es' \Downarrow \boldsymbol{vs}.$$

The extended history is denoted H'. It must *refine* the original one, $H \sqsubseteq H'$, meaning that all variables defined in H are defined in H' with the same value, $\forall y\ v,\ H(y) = v \implies H'(y) = v$. Additionally, H' must satisfy any new equations. In the Coq proof, the statement described here is augmented with technical clauses about the identifiers in the before and after states of the monad, and the domains of H and H'.

```
node count_down(res:bool; n:int)
returns (cpt:int)
var norm$1 : int;
let
  norm1$1 = n fby (cpt - 1);
  cpt = if res then n else norm1$1;
tel
```

res	F	T	F	F	F	F	T	F	⋯
n	3	3	3	3	3	3	3	3	⋯
norm1$1	3	2	2	1	0	-1	-2	2	⋯
cpt	3	3	2	1	0	-1	3	2	⋯

Fig. 14. Unnested version of count_down and an example execution

The proof follows by induction over the syntax of expressions. The most interesting cases are those for fbys and node instanciations, where unnesting occurs. When a subexpression e_x is unnested from the expression e, a new equation $x = e_x$ is introduced and e_x is replaced by x in e to give a new expression e'. Assuming $G, H, bs \vdash e \Downarrow \boldsymbol{vs}$, we must show the existence of an H' that (i) refines H, (ii) satisfies the constraint imposed by the new equation, $G, H', bs \vdash x = e_x$, and (iii) ensures $G, H', bs \vdash e' \Downarrow \boldsymbol{vs}$. Since e has a semantics under H, so too do its subexpressions. There is thus a stream associated with e_x that we shall name v_x, $G, H, bs \vdash e_x \Downarrow v_x$. We can now introduce $H' = H[x \mapsto v_x]$ as a witness. The technical clauses mentioned above allow us to conclude that x, a fresh variable, is not in the domain of H. From this it follows directly that $H \sqsubseteq H'$. Since H' refines H and x does not occur in e_x, it follows from the original semantics of e that $G, H', bs \vdash x = e_x$, and, in turn, that $G, H', bs \vdash e' \Downarrow \boldsymbol{vs}$.

Where reasoning in previous work [1] focuses on substitution within expressions, the central object in our induction is the construction of a chain of histories to incorporate fresh variables while preserving the meaning of existing expressions.

The function on expressions is iterated over the list of equations within a node. The corresponding proof proceeds by induction on the list and ultimately exhibits an H' that refines the original H and incorporates the new equations that unnesting introduces. This H' provides the history required to exhibit the semantics of a node using the rule in figure 8d. Since this history refines the original one, the outputs of the transformed node match those of the original one.

As an example, figure 14 shows the result of applying unnesting to the count_down node from figure 2. The grid shows a possible behavior; it illustrates the updated history that associates the new variable norm1$1 with a stream, while maintaining the original association for cpt.

The distribution of operators over their lists of arguments involves a technical detail. An expected property of the unnesting and distribution function, $\lfloor e \rfloor = (\boldsymbol{es}', \boldsymbol{eqs}')$, is that the length of the produced list of expressions, $\boldsymbol{es}'$, equal the number of streams associated with the input expression, e. The actual implementation of the cases presented in figure 13 relies on type annotations to generate $\boldsymbol{es}'$. The proof thus requires that e be well-typed, which the elaboration pass ensures.

5 EXPRESSION INITIALIZATION

The result of unnesting and distribution is almost in normal form. It only remains to ensure that fbys be initialized by constant expressions.

5.1 Expression Initialization Algorithm

We start by defining a subset of expressions: a *slow constant* is a constant wrapped in zero or more whens. We will write c^{ck} for the constant c wrapped in whens so as to have the clock type ck. For example, $\text{true}^{\bullet \text{ on b on c}}$ represents true when b when c. This notation intentionally mimics the notation for showing an expression together with its clock type annotation.

The role of expression initialization is to transform an equation of the form x = (e0 fby e)^{ck}, where e0 is not already a slow constant, into the three equations shown below at right.

$$\left\lfloor \mathsf{x = (e0\ fby\ e)}^{ck} \right\rfloor_{\mathsf{fby}} = \begin{cases} \mathsf{x = if\ xinit\ then\ e0\ else\ px;} \\ \mathsf{xinit = true}^{ck}\ \mathsf{fby\ false}^{ck}; \\ \mathsf{px = def}_{ty}^{ck}\ \mathsf{fby\ e;} \end{cases}$$

The new equation for x uses a fresh variable xinit to choose between the value of the initial expression e0 and the value of a second fresh variable px (we reuse the state monad to generate fresh identifiers). The equation for xinit is only true at the first instant of presence of streams with clock type ck. The whens in the slow constants synchronize the fby. The equation for px delays the stream associated with e. Its initial value is def_{ty}, an arbitrary constant of type ty. Since our language only manipulates boolean, integer, and floating-point values, it is always possible to choose such a constant (False, 0, 0.0). The value of the constant is never used in the definition of x.[2] These three equations are in normal form because only slow constants are used at left of the fbys. The subsequent transcription pass removes the whens from such constants.

As an example of this transformation, the unnested version of count_down in figure 14 is transformed into the fully normalized form in figure 3.

For two equations containing fbys with the same clock type, a naive implementation of the above schema would produce two identical initialization equations. For example, the normalization of (x, y) = (x0, y0) fby (y, x) would give rise to two equations identically defined by true fby false. It is especially important to avoid this because each fby requires its own state memory in the generated imperative code. We thus use the state monad to memoize and reuse generated initialization equations. As we describe in the next section, this optimization complicates the correctness proof since it requires non-local reasoning.

5.2 Expression Initialization Correctness

Theorem 5.1. *Expression initialization preserves the semantics of any well-typed, well-clocked, and causal program G:* $\forall f\ \boldsymbol{xs}\ \boldsymbol{ys},\ G \vdash f(\boldsymbol{xs}) \Downarrow \boldsymbol{ys} \implies \lfloor G \rfloor_{\mathsf{fby}} \vdash f(\boldsymbol{xs}) \Downarrow \boldsymbol{ys}$.

The core of the proof is to show that the semantics of an equation x = e0 fby e is preserved in the new equations generated from it. As in the proof for unnesting and distribution, a history variable is extended to incorporate the new identifiers. We reuse the technique presented in section 4.2 based on successive refinements and technical clauses about the set of identifiers in the monad state. An additional difficulty is to reason from the semantics of the initial fby to show that expressions involving slow constants, two new fbys, and an if construction each also have a semantics, and that their composition matches the original one. The basic idea is to apply rule inversion on the predicate of the original fby to obtain the semantics of its constituent expressions, to rely on clock system correctness to interrelate their streams and clock types, and to reestablish the semantics of the new elements by applying the predicates as introduction rules.

Slow constants c^{ck}. A constant c can be associated directly with the stream const bs c (figure 8a). Obtaining the stream for c^{ck} is more difficult because it requires associating the clock type ck with a stream and, as explained in section 3, the semantic predicates do not directly encode this information. Happily, the lemmas used in the proof of theorem 3.1 also imply that if $G, H, bs \vdash e \Downarrow \boldsymbol{vs}$ and e has the clock type ck then $\forall v \in \boldsymbol{vs},\ G, H, bs \vdash ck \Downarrow$ abstract-clock v, where abstract-clock simply maps present values to true and absent values to false. Now, since the original fby expression is associated with a stream of svalues, its clock type ck can be associated with a stream of booleans

[2]It would be better to define px = pre e, but Vélus does not yet support the uninitialized delay operator, pre.

and, furthermore, the two streams are aligned. The passage from the semantics of a constant and clock, for example, c and • on b1 on b2, to the semantics of the expression for their slow constant, c when b1 when b2, follows easily by induction. Putting all of this reasoning together allows to obtain streams for the three slow constants in the produced equations, true^{ck}, false^{ck}, and def_{ty}^{ck}. The proofs in Coq are quite technical due to the invariants required by the clock correctness lemmas, which also require that a program be well typed, well clocked, and causal.

Delay equations. We must obtain streams for the fby expressions that define xinit and px in the produced equations. To do so, we simply apply the $\mathsf{fby}_{\mathsf{NL}}$ operator, from figure 11, to the streams obtained for the slow constants or associated with the expression e in the original fby equation. Passing from this operator to the coinductive predicate used for Lustre, figure 10, exploits the fact that the component streams are aligned.

Choosing a value with if. Finally, we must show that the initial equation x = e0 fby e and its replacement x = if xinit then e0 else px denote the same stream. Applying rule inversion to the semantic predicate for the former gives fby $y_0\ y \doteq z$, that is, the relevant semantic operator applied to streams associated with, respectively, e0, e, and x. From this, the results presented above, and a semantic operator for the conditional operator, we construct the equivalent stream $\mathsf{ite}\,(\mathsf{fby}_{\mathsf{NL}}\,\mathsf{true}\,(\mathsf{const}\;cs\,\mathsf{false}))\;y_0\;(\mathsf{fby}_{\mathsf{NL}}\,\mathsf{def}_{ty}^{ck}\;y)$, where cs is the boolean stream for the clock type ck of the original expression. This gives the desired semantics to the generated equation.

Causality Preservation. The correctness proof for expression initialization only applies to causal programs. That is why program causality is checked just beforehand, at the dot in figure 1, as described in section 2.3. The same property is also required for the subsequent transcription pass. Rather than re-execute the graph-based check, we prove that expression initialization preserves the causality of a program. This excludes a source of error and gives a more efficient compiler.

The core of the proof treats a list of equations within a causal node. Since the node is causal, there exists an acyclic graph with an edge from y to x iff the variable y is required transitively by the defining expression or clock type of x. The proof involves showing that such an acyclic graph also exists for the equations produced by the expression initialization function. Consider the treatment of an equation x = $(\mathsf{e0}\ \mathsf{fby}\ \mathsf{e})^{ck}$. The original graph contains an arc to x from any variable y that is required by e0 or ck. According to the definition of IsFreeLeft, the requirements of e are not considered. The function will add a new equation for px and the new graph must contain an edge from px to x, and from any variable in ck to px. There are no cycles in the updated graph, since there are none in the original one and it already has edges from each variable in ck to x. The function will either add a new equation for xinit or reuse an existing one. In either case, the initialization equation only depends on variables in ck, which are already required for x and so the graph can be extended without introducing cycles. The Coq proof is structured around an invariant that takes the monad state into account.

6 NODE SUBSAMPLING

The preceding three sections present algorithms and proofs to enable the richer expression language of figure 5. In this section, we describe the removal of a restriction on clock types in node interfaces. In previous versions of Vélus, "•" is the only valid clock type for node input and output variables, that is, for instantiating the ck annotations of figure 7. With our modifications, the clock type of an input variable may depend on other input variables, and that of an output variable may depend on input variables and other output variables. The node called current at the end of the introduction is a simple example.

$$\frac{\begin{array}{cc}\text{wc\_env } n.\mathbf{in} & \text{wc\_env } (n.\mathbf{in} +\!\!+\, n.\mathbf{out} +\!\!+\, n.\mathbf{vars}) \\ \text{wc\_env } (n.\mathbf{in} +\!\!+\, n.\mathbf{out}) & \text{wc\_equation } G\ (n.\mathbf{in} +\!\!+\, n.\mathbf{out} +\!\!+\, n.\mathbf{vars})\ n.\mathbf{eqs}\end{array}}{\text{wc\_node } G\ n}$$

$$\frac{}{\text{wc\_clock } \boldsymbol{nenv}\ \bullet} \qquad \frac{\begin{array}{c}\text{wc\_clock } \boldsymbol{nenv}\ ck \\ x_{ty}^{ck} \in \boldsymbol{nenv}\end{array}}{\text{wc\_clock } \boldsymbol{nenv}\ (ck \text{ on } x)} \qquad \frac{\begin{array}{c} sub\ x = no \\ \text{instck } bk\ sub\ ck = \text{Some } ck'\end{array}}{\text{Well-Inst } bk\ sub\ x_{ty}^{ck}\ (no, ty', ck')}$$

$$\frac{\begin{array}{cc}\text{wc\_exp } G\ \boldsymbol{es} & \text{Well-Inst } bk\ sub\ n.\mathbf{in}\ (\text{annots } \boldsymbol{es}) \\ \text{node}(G, f) \doteq n & \text{Well-Inst } bk\ sub\ n.\mathbf{out}\ \boldsymbol{anns}\end{array}}{\text{wc\_exp } G\ \ (f(\boldsymbol{es}))^{\boldsymbol{anns}}} \qquad \frac{}{x \doteq \text{None}} \qquad \frac{}{x \doteq \text{Some } x}$$

$$\frac{\text{wc\_exp } G\ \boldsymbol{es} \qquad \text{Forall2 } (\lambda x\ (no, ty, ck),\ x \doteq no \wedge x_{ty}^{ck} \in \boldsymbol{nenv})\ \boldsymbol{xs}\ (\text{annots } \boldsymbol{es})}{\text{wc\_equation } G\ \boldsymbol{nenv}\ (\boldsymbol{xs} = \boldsymbol{es})}$$

Fig. 15. A selection of clock typing predicates for subsampling nodes

As far as semantics is concerned, the generalization for node sampling only requires removing a minor constraint from the previous rule for nodes [5, §3.1]. Much more effort was required to update the typing predicates for clocks. While prior formalizations [9] present these predicates precisely, translating them into Coq requires encoding a substitution mechanism in the rule for node instances, and handling dependencies between nested node instances. The biggest difficulty we faced, however, was adapting the generation of imperative code. When a stream is absent in a synchronous program, the value of the corresponding variable in the generated code is undefined. By default, such values may be passed in the function calls generated from subsampling node instances. Unfortunately, the resulting behavior is undefined in the C standard and in Clight. We propose an algorithm to avoid this problem by initializing such variables if necessary.

6.1 Clock Typing

We now present a subset of clock typing predicates that expresses our treatment of node subsampling in Coq. They can be compared with formal pen-and-paper definitions [9, §4].

The predicate wc_node, shown in figure 15, asserts that a node is *well clocked* in a program G only if (i) the input declarations form a consistent clocking environment, (ii) so too does the concatenation of input and output declarations, (iii) so too does the concatenation of input, output, and local declarations, and (iv) every equation is well clocked in the complete environment. A consistent clocking environment is one where every clock type is well formed with respect to the other declarations: wc_env $nenv = \forall x_{ty}^{ck} \in nenv,$ wc_clock $nenv\ ck$. The wc_clock predicate, also shown in figure 15, requires a variable x to have the same clock ck as the stream it filters.

Every expression is annotated with a triple (no, ty, ck) giving its optional stream name, type, and clock type. The optional stream name encodes dependencies between streams. For most expressions it is None, but for a variable x it may be Some x, and for each output of a node instance it is Some s, where s is either a declared variable or an *anonymous variable*. Anonymous variables are introduced during elaboration and constrained to be unique with respect to all other declared and anonymous variables in the enclosing node. They encode dependencies between nested node instances and become declared variables during unnesting and distribution (like elab\$4 in figure 3).

```
if (ck) {
  elab$4 := count(i0).step(0, 1, ⟨reset⟩)
};
time := current(i1).step(0, ⟨ck⟩, elab$4)
```

Fig. 16. Obc produced by s-translation

```
if (ck) {
  elab$4 := count(i0).step(0, 1, ⟨reset⟩)
} else {
  elab$4 := 0
};
time := current(i1).step(0, ⟨ck⟩, ⟨elab$4⟩)
```

Fig. 17. Obc after argument initialization

The case of wc_exp that defines when a node instance is well clocked is shown in figure 15. It requires the existence of a base clock type bk and a substitution function sub that partially maps variables from the signature of the instantiated node to any stream names in its context. The Well-Inst predicate is applied pair-wise to node input declarations and the annotations of the argument expressions, and to node output declarations and the annotations of the node instance expression. It requires that any stream names are accounted for in the substitution function and that the declared node clock types are correctly instantiated to give the clock type annotations. The instck function is defined recursively as follows using the standard option monad.

$$\text{instck } bk\ sub\ \bullet = \text{Some } bk$$

$$\text{instck } bk\ sub\ (ck \text{ on } x) = instck\ bk\ sub\ ck \gg= (\lambda ck',\ sub\ x \gg= (\lambda x',\ \text{Some } (ck' \text{ on } x')))$$

where $\gg=$ is the monadic bind operator: (None $\gg= f$) = None and (Some $x \gg= f$) = $f\ x$.

Finally, an equation is well clocked if its defining expressions are well clocked and if each pair of defined variable and corresponding annotation is coherent, that is, has the same optional stream name and a matching declaration in the environment *nenv*.

6.2 Compilation to Imperative Code

As figure 1 shows, after normalization and transcription, the equations of an NLustre program are translated into transition constraints in an intermediate language called Stc, these constraints are scheduled, and then translated into a sequence of commands in a simple imperative language called Obc. Consider, for example, the following Lustre equation adapted from a classic example [7].

```
time = current(0, ck, count((0, 1, reset) when ck));
```

The Obc fragment produced for this equation is shown in figure 16. Unnesting has introduced a local variable (elab$4), scheduling has ensured it will be written before being read, and translation has replaced the node instances with calls to step methods on explicit instances (i0 and i1), possibly guarded by conditional statements. Nested conditional statements are generated recursively according to the clock types of the equations. The assignment to elab$4 is guarded because the clock type of the node instance is • on *ck*. The assignment to time is not guarded because the clock type of the node instance is •. The when operators have simply been removed; sampling now occurs implicitly. We explain the special brackets around reset and ck in the next section.

Notice, in the example fragment, that the value of elab$4 passed to the second step invocation is undetermined when ck is false. We modified the semantics of Obc to allow this, but a direct translation into Clight is problematic. The C99 standard makes it clear that passing an undefined value in a function call is not well defined: *if an lvalue does not designate an object when it is evaluated, the behavior is undefined* [13, §6.3.2.1], *in preparing for the call to a function, the arguments are evaluated, and each parameter is assigned the value of the corresponding argument* [13, §6.5.2.2]. This interpretation is formalized in the semantic model of Clight [3, figure 10]. Evaluating an internal function call requires first evaluating the argument expressions one-by-one (eval_exprlist).

$$\frac{env\ x = vo}{menv, env \vdash x \Downarrow vo} \qquad \frac{menv_{\mathrm{v}}\ x = \mathsf{Some}\ v}{menv, env \vdash \mathtt{st}(x) \Downarrow \mathsf{Some}\ v} \qquad menv, env \vdash c \Downarrow \mathsf{Some}\ [\![c]\!]$$

$$\frac{menv, env \vdash e_1 \Downarrow \mathsf{Some}\ v_1 \qquad menv, env \vdash e_2 \Downarrow \mathsf{Some}\ v_2 \qquad [\![v_1 \oplus v_2]\!] = \mathsf{Some}\ v'}{menv, env \vdash e_1 \oplus e_2 \Downarrow \mathsf{Some}\ v'}$$

$$\frac{menv, env \vdash e \Downarrow \mathsf{Some}\ v}{menv, env \vdash \langle e \rangle \Downarrow \mathsf{Some}\ v} \qquad \frac{menv, env \vdash e \Downarrow \mathsf{Some}\ v}{p, menv, env \vdash x := e \Downarrow (menv, env[x \mapsto v])}$$

Fig. 18. Obc: selected semantic rules for expressions and statements

The type cast applied to each (sem_cast) is required to give some value, but casting the value given to uninitialized local variables (Vundef) to anything other than the void type gives None.

There are several ways to circumvent this problem. We could target CompCert's Cminor language, which allows undefined arguments, but this would require reproducing and reverifying argument stacking. We could fork CompCert, modify the Clight semantics, and update its proofs. We could, as does Scade 6 KCG, inline nodes that subsample, but inlining is not implemented in Vélus and we were curious to evaluate a different approach. We could change the encoding of method calls in Clight to pass a pointer to a `struct` containing the argument values, or store some of them as state variables, but this would complicate code generation and the associated correctness proofs. Finally, we could simply initialize all local variables, rely on CompCert to optimize some unnecessary writes and otherwise accept the slight overhead. We essentially adopt this solution, but with two refinements. First, we exploit our correctness proof and some tweaks to the semantics of Obc to avoid introducing unnecessary writes as explained in section 6.3. Second, we add a compilation pass that rewrites Obc programs by adding initializations as explained in section 6.4. The new compilation pass uses a heuristic to minimize duplicate writes while trying not to add too many new writes. Our approach has no overhead for programs that do not instantiate nodes with subsampling.

6.3 Translation to Obc

The Obc language comprises the expressions e and statements s described by the following grammar.

$$\begin{aligned} e &::= x \mid \mathtt{st}(x) \mid c \mid \diamond e \mid e \oplus e \mid \langle e \rangle \\ s &::= x := e \mid \mathtt{st}(x) := e \mid \mathtt{if}\ e\ \{s\}\ \mathtt{else}\ \{s\} \mid s\ ;\ s \mid xs := cls(i).f(es) \mid \mathtt{skip} \end{aligned}$$

Expressions are variables x, state variables $\mathtt{st}(x)$, constants c, unary operators $\diamond$, binary operators $\oplus$, and the new *validity assertions* $\langle e \rangle$ introduced in this article. Statements are assignments to variables and to state variables, conditionals, sequential compositions, method calls, and no-ops. A method call, $xs := cls(i).f(es)$, specifies result variables xs, the class being invoked cls, a state instance i, a class method f, and argument expressions es. The big-step semantics of expressions and statements is defined relative to a partial mapping from variable identifiers to values (env), and a pair of partial mappings ($menv$) from state variables to values ($menv_{\mathrm{v}}$) and from instance variables to, recursively, pairs of partial mappings ($menv_{\mathrm{o}}$).

The predicate $menv, env \vdash e \Downarrow vo$ asserts that, in environments $menv$ and env, expression e evaluates to optional value vo. The predicate $p, menv, env \vdash s \Downarrow (menv', env')$ asserts that, in a program p and environments $menv$ and env, statement s produces updated environments $menv'$ and env'. A selection of rules are shown in figure 18. The only remarkable feature is the treatment of partial values. An expression x evaluates to None if x is not defined in the variable environment,

$$\frac{}{\mathsf{NormArg}\ \mathsf{base}\ e} \qquad \frac{}{\mathsf{NormArg}\ ck\ c} \qquad \frac{}{\mathsf{NormArg}\ ck\ x} \qquad \frac{\mathsf{NormArg}\ ck\ e}{\mathsf{NormArg}\ (ck\ \mathsf{on}\ x)\ (e\ \mathtt{when}\ x)}$$

Fig. 19. Normalization condition on argument clocks and expressions in NLustre

that is, if no earlier assignment to it has occurred. All other expressions are required to evaluate to some value. In particular, the validity assertion is only defined when its argument is not None. The meaning of assignment statements is only defined for expressions that do not evaluate to None. Importantly, however, the rule for method calls (not shown) is not restricted in this way, which makes it possible to pass and return variables even when they have not been written.

The correctness proof for s-translation requires showing that if a source program has a semantics, then so too does its translation into Obc. A central invariant relates presence or absence in the source program to the execution or not of generated statements. Translating the clock type of an equation into nested conditionals ensures that the code for a node instance is only executed when at least one of its inputs is present. On one hand, this guarantees that variables with the same clock type as the equation are always defined when the corresponding method is called. We encode this fact by wrapping such variables in validity assertions: as for reset and ck in figure 16. This syntactic addition is justified in the proof and exploited in a subsequent compilation pass. On the other hand, the introduction of subsampling requires care for expressions containing variables with slower clock types. Consider, for example, the following slight variation on the earlier equation.

```
time = current(0, ck, count((0, 1, reset) when ck) + (1 when ck));
```

A naive translation to `time := current(i1).step(0,⟨ck⟩, elab$4 + 1)` would be incorrect, because when ck is false, elab\$4 is not defined, and thus the expression elab\$4 + 1 does not have a semantics—nor would it in C or Clight. To ensure that the generated Obc always has a semantics, and thus permit a correctness proof, we require that all node argument expressions in the source program satisfy the predicate of figure 19. The predicate is defined relative to the corresponding clock type in the node signature. For parameters of clock type •, any argument expression is allowed since the clocking rules and correctness invariant guarantee that any variables it contains are present when the node is activated. Constants and variables are also always allowed. Otherwise, one or more levels of sampling may be stripped away provided that the expression being sampled is present on the base clock of the node. The Coq implementation of unnesting and distribution ensures that NormArg is always satisfied by unnesting argument expressions whenever necessary.

6.4 Argument Initialization

The generated Obc has validity assertions for variable arguments that are always defined at method calls. We now present an *argument initialization* pass that adds validity assertions to other variable arguments and justifies them by adding initializing assignments at earlier points in the program. This pass occurs after fusing adjacent conditionals and before generating Clight (see figure 1).

The function that adds assertions and assignments to an Obc statement has the interface: $(s', \mathit{required}', \mathit{sometimes}, \mathit{always}) = \text{add\_defaults}\ \mathit{required}\ s$. It takes a set *required* of variables to be initialized and a statement s to transform. It returns (i) s', an updated statement, (ii) $\mathit{required}'$, a set of variables that must be initialized before s' is executed, (iii) *sometimes*, a set of variables that are sometimes but not always written by s', and (iv) *always*, a set of variables that are always written by s'. We prove that $\mathit{always} \cap \mathit{sometimes} = \emptyset$.

```
Definition add_valid e (es, req) :=
  match e with
  | x ⇒ (⟨e⟩ :: es, req ∪ {x})
  | _ ⇒ ( e  :: es, req)

Fixpoint add_defaults req s :=
  match s with
  | skip ⇒ (s, req, ∅, ∅)
  | st(x) := e ⇒ (s, req, ∅, ∅)
  | x := e ⇒ (s, req − {x}, ∅, {x})

  | xs := f(o).m(es) ⇒
    let (es′, req′) :=
        fold_right add_valid ([], req − xs) es in
    (xs := f(o).m(es′), req′, ∅, xs)

  | s1 ; s2 ⇒
    let (t2, req2, st2, al2) := add_defaults req s2 in
    let (t1, req1, st1, al1) := add_defaults req2 s1 in
    (t1 ; t2, req1, (st1 − al2) ∪ (st2 − al1), al1 ∪ al2)

  | if e { s1 } else { s2 } ⇒
    let (t1, req1, st1, al1) := add_defaults ∅  s1 in
    let (t2, req2, st2, al2) := add_defaults ∅  s2 in
    let (alreq1, alreq2) := (al1 ∩ req, al2 ∩ req) in
    let (w1, w2) := (alreq2 − alreq1, alreq1 − alreq2) in
    let w := ((st1 ∩ req) − w1) ∪ ((st2 ∩ req) − w2) in
    let (al′1, al′2) := (al1 ∪ w1, al2 ∪ w2) in
    let (st′1, st′2) := (st1 − w1, st2 − w2) in
    (add_writes w (if e { add_writes w1 t1 } else { add_writes w2 t2 }),
     (((req − alreq1) − alreq2) ∪ req1 ∪ req2) − w,
     (st′1 ∪ st′2 ∪ (al′1 − al′2) ∪ (al′2 − al′1)) − w,
     (al′1 ∩ al′2) ∪ w)
```

Fig. 20. Function for adding initializations to an Obc program

The Coq definition of the function is presented in figure 20.[3] The cases for skip and state assignment are straightforward. For assignment, the written variable is removed from the required set and added to the always set. For method calls, we remove the variables *xs* at left of the equation from the required set before folding add_valid over the argument expressions: any variable at the root of an expression is wrapped in a validity assertion and added to the list of required variables. In other words, we assert that such variables are defined and add them to the set of variables to initialize beforehand. The case for sequential composition works backward, propagating the required sets before recalculating the sometimes and always sets.

The case for conditionals is the most involved. It makes recursive calls with empty *required* arguments and then calculates the writes w_1 to add in the branch before s_1, the writes w_2 to add in the branch before s_2, and the writes w to add before the conditional statement itself. In the branch for s_1 we add writes for required variables that are always written by s_2 but not always written by s_1. The branch for s_2 is treated similarly. The set of writes added before the conditional includes all required variables that are sometimes but not always written in either of the two branches, taking care to remove variables initialized by the newly added writes in the two branches. The other definitions calculate the updated required, sometimes, and always sets. We do not show the definition of add_writes w s. It simply returns a sequence of assignments of default values to the elements of w followed by the statement s.

The add_defaults function is applied to the body of each method in each class of a program. For a method, in terms of the add_defaults invocation above, the method outputs are passed as the initial *required* set, and the updated body is add_writes $(required' - in)\ s'$. That is, the outputs must be initialized and any pending writes, excluding input variables, are added before the updated body. Requiring the initialization of outputs is not strictly necessary, but it does simplify the implementation and correctness proof.

The add_defaults function is designed for the Obc generated from Lustre programs and embodies a compromise between initializing variables unnecessarily and adding many initializing writes. Adding writes at the leaves of nested conditionals may greatly increase the statement size, but

[3]For readability, we have simplified some minor details and sometimes treat lists as sets.

	Files	Spec.	Code	Proofs	Admin.	Total
Lexing/Parsing (Menhir [?])	3	0	787	0	0	787
Elaboration	1	156	776	338	48	1 318
Lustre	8	2 625	495	5 174	287	8 581
Unnesting & Expression Initialization	10	2 735	285	8 424	370	11 814
Transcription	8	577	161	1 754	386	2 878
NLustre $\rightsquigarrow$ Obc	58	5 393	1 006	9 944	1 985	18 328
Argument Initialization	1	304	72	1 371	47	1 794
Fusion, Obc $\rightsquigarrow$ Clight	8	2 219	636	5 187	282	8 324
Common definitions & driver	23	4 901	418	8 648	783	14 750
Total	120	18 910	4 636	40 840	4 188	68 574

Table 1. Files and Lines of Code in the prototype excluding blank lines and comments.

always adding them before conditional statements increases the number of unnecessary writes. Our function may sometimes add writes to branches unnecessarily, but doing better would require essentially reconstructing the clock type information lost in the transformation to Obc.

The result of applying argument initialization to the fragment presented earlier is shown in figure 17. When ck is false, elab$4 is initialized to a default value, thus justifying the validity assertion added to the method call for current and guaranteeing the generation of valid Clight code.

Correctness. A statement s' generated by add_defaults does not necessarily calculate exactly the same result as the original statement s, so we formalize the correctness invariants using the refinement relation that was used between history variables in section 4.2. Basically, any variable defined by evaluating s is defined with the same value by evaluating s'. Lifting this invariant from statements to methods and then to classes and programs involves shoehorning a technical invariant on the initial environments of method calls, but is otherwise straightforward. The proofs require a predicate stating that no variable is every written more than once. We show that this predicate holds for the code produced by s-translation and after the optimization that fuses conditionals, even if it is not preserved by add_defaults.

7 PROTOTYPE IMPLEMENTATION

The work presented in earlier sections extends the Vélus compiler whose architecture is shown in figure 1. All of the compilation passes except parsing and scheduling are implemented together with the associated models and proofs in Coq. An executable is produced by extracting an OCaml program and compiling it. Table 1 gives an idea of the size of our extension, in bold, relative to the size of the original compiler. We do not count the relatively small changes to the Obc language. The number of lines are calculated by an instrumented version of coqwc. For each definition that is not a Theorem, Lemma, Corollary, or similar, we check whether it appears in the extracted OCaml program, in which case it is considered code. The other lines are divided into specifications (semantic models, type and clock rules), proof scripts, and administrative details (library imports and module instantiations).

The new proofs for Lustre and its normalization add a significant number of lines to the project for an overall code to total ratio of 1:15. That said, we did not put much effort into proof automation.

As a simple experiment, we compiled non-normalized versions of the programs from [5, Fig. 12] with our prototype. The non-normalized programs are on average 14% smaller than the corresponding normalized versions. The code produced is identical provided that one maintains individual equations for fby expressions that are used multiple times (as in landing_gear and prodcell) as common subexpressions are not eliminated.

8 RELATED WORK

Our work contributes to efforts to formally specify and verify programming languages and their compilers. In particular, it is part of a tradition [?] of domain-specific languages for real-time embedded controllers. These so called *synchronous languages* have long been rigorously defined with the ideal of *What You Prove Is What You Execute (WYPIWYE)* [? , §5]. It is natural to apply computer-based tools to the specification and compilation of these languages. The challenges are to find appropriate definitions for models and algorithms, and to develop the necessary invariants and proof schemas.

We focus on the application of interactive theorem provers, or proof assistants, to compilation algorithms. There are three basic approaches [14, §2.2]. A compilation pass can be *verified* directly, its results can be checked by a *verified validator*, or it can produce *proof carrying code*, that is, code together with a proof of correctness. We choose to directly verify the normalization pass. Auger [1, §7.1], on the other hand, uses an OCaml function to calculate equation introductions and substitutions, and then validates the results in Coq. He implements unnesting and distribution but does not need to initialize expressions because his source language [1, §5.1] only allows fbys with constants at left. He does not treat node subsampling, clock system correctness, or imperative code generation. Verified validation has advantages, programming in OCaml is less restrictive than it is in Coq and the proofs are simpler, but, arguably, it gives less insight into the algorithms used and requires rechecking compilation runs. We show that verified normalization is feasible. Other work on verified Lustre compilation [5, 16] does not address node sampling or normalization.

In terms of semantic models and type systems, our Coq encoding directly adapts previous formal pen-and-paper definitions [9]. As expected, encoding them in an interactive proof assistant involves a number of technical details, especially around the use of partial relations, coinductive streams, and our choice to work with lists rather than tuples. Boulmé and Hamon [4] specify a higher-order dataflow language as a shallow embedding in Coq, but do not address compilation. Clock alignment is guaranteed by an encoding with dependent types. We do not know how to apply this idea to the deep embedding used by our compiler.

The term *translation validation* was coined [?] for checking the compilation of the synchronous language Signal [?]. Signal has many similarities to Lustre, including conditionally executed equations, delay equations, and a semantic model based on streams with absence and presence. The original Signal compiler is based on an intermediate language called DC+, which is comparable to the NLustre and Stc languages used in Vélus, in that it consists of sets of conditionally activated equations to variables classed as either “volatile” or “memorized”. The Translation Validation Tool (TVT) [?] generates proof obligations for comparing a transition system model for a DC+ program to one for the C code generated from it. This approach was later adapted for checking the compilation of a class of Simulink models [?]. These validation tools are not themselves verified. They do not need to treat normalization since DC+ and the block diagram models of Simulink are already in normal form. It is unclear whether node subsampling is treated or whether the flattening of subsystem blocks is validated.

Work on translation validation was continued [?] in the context of the Polychrony toolset, which is based on Signal. It does not address normalization, since Signal equations are already in a normalized form [? , §2.1]. While in principle Signal has a more expressive clock system than does Lustre and its functions may abstract over subsampled inputs and outputs [**?** , Listing 3.3]; in practice, semantic models and validation algorithms only treat "synchronous" functions where all inputs and outputs are present and absent simultaneously [**?** , §2.1][**?** , p.38].

The semantics of Signal has been modeled in Coq using co-inductive definitions of primitive processes [**?**]. This work focuses on reasoning about a small example with synchronous functions, that is, where all inputs and outputs are present or absent simultaneously. Neither normalization, node subsampling, nor compilation are addressed.

Our work can be compared with the verified compilation of general-purpose languages. The main difference is in the semantic models of the source and intermediate languages. Lustre focuses on compositions of temporal behaviours. The value of an expression is thus modeled as a stream representing an evolution over time and a node represents a function between lists of streams. The equations within a node give rise to a conjunction of constraints that determine the overall behaviour of the node. This is in contrast to imperative languages that are usually modeled using an operational semantics, that is, as transitions between states of an abstract machine. For example, the state for CompCert's Clight language includes environments for global and local variables, a representation of blocks in memory, a call stack, and a possibly infinite trace of external events [**?**]. The Vélus compiler establishes a formal link between these two types of semantic models.

Compilers for imperative languages often use intermediate representations in static single assignment (SSA) form. Similarly to Lustre, this form requires each variable to be uniquely defined by a single expression and emphasizes "causal" dependencies between variables. That said, semantic models of SSA form are closer to those of imperative languages. For example, the Coq model used to encode an intermediate representation for LLVM [**?**] is based on successive updates to an environment mapping variables to values, with a control state that successively transitions between and within blocks of instruction sequences. Even in formalizations where the semantics within a block does not depend on instruction ordering [**?**], the overall model remains operational.

CakeML [**?**] is a specification and verified compiler with read-eval-print-loop for an ML language in the HOL interactive theorem prover. Its operational semantics is defined by an interpreter [**?**] that reduces expressions to values. The state of the interpreter is more abstract than that of a Clight program—it tracks value and reference bindings and a trace of external calls, and handles function closures and abstract data types—but the semantic model is still based on consecutive steps, which is quite different to our model of constraints between streams.

The depth-first search algorithm that we use to construct a witness for the absence of dependency cycles is at the core of Tarjan's and Kosaraju/Sharir's algorithms for finding strongly connected components, both of which have previously been specified and verified in interactive theorem provers [**? ?**]. Rather than use a "fuel" argument to reason about termination [**?** , §4], we use a recursive formulation based on dependent types and programmed with tactics [**?** , §4].

9 CONCLUSION

We have presented the specification and end-to-end proof in an interactive theorem prover of the semantics and compilation algorithms for a significant subset of Lustre that includes subsampling nodes. We build on the existing Vélus and CompCert compilers, adding new passes to normalize source programs into the form required by the backend and to ensure the initialization of variables passed in function calls. The result is a working prototype and a theorem that the semantics of source programs are reproduced by the generated assembly code. Though we omit them from figure 5, the actual compiler supports the modular reset [6, 12] and initialization (->) operators. The former does not pose any particular problems and the latter is treated during expression initialization by exploiting the fact that e0 -> e is equivalent to if (true fby false) then e0 else e.

ACKNOWLEDGMENTS

We thank L. Brun for his many contributions to the Vélus compiler, J.-L. Colaço for his explanations about Scade, X. Leroy for his explanations and advice on CompCert, and A. Guatto for his suggestions. This work was supported by the Bpifrance "Programme d'Investissements d'Avenir" in the ES3CAP project and the ANR JCJC FidelR 19-CE25-0014 project "Fidelity in Reactive Systems Design and Compilation".

REFERENCES

[1] C. Auger. 2013. *Compilation certifiée de SCADE/LUSTRE.* Ph.D. Dissertation. Univ. Paris Sud 11.

[2] D. Biernacki, J.-L. Colaço, G. Hamon, and M. Pouzet. 2008. Clock-directed modular code generation for synchronous data-flow languages. In *LCTES 2008.* ACM Press, 121–130.

[3] S. Blazy and X. Leroy. 2009. Mechanized Semantics for the Clight Subset of the C Language. *J. Automated Reasoning* 43, 3 (Oct. 2009), 263–288.

[4] S. Boulmé and G. Hamon. 2001. Certifying Synchrony for Free. In *LPAR 2001 (LNCS)*, Vol. 2250. Springer, 495–506.

[5] T. Bourke, L. Brun, P.-É. Dagand, X. Leroy, M. Pouzet, and L. Rieg. 2017. A Formally Verified Compiler for Lustre. In *PLDI 2017.* ACM Press, 586–601.

[6] T. Bourke, L. Brun, and M. Pouzet. 2020. Mechanized Semantics and Verified Compilation for a Dataflow Synchronous Language with Reset. *Proc. of the ACM on Programming Languages* 4, POPL (Jan. 2020), 1–29.

[7] P. Caspi, D. Pilaud, N. Halbwachs, and J.A. Plaice. 1987. LUSTRE: A declarative language for programming synchronous systems. In *POPL 1987.* ACM Press, 178–188.

[8] J.-L. Colaço, B. Pagano, and M. Pouzet. 2017. Scade 6: A Formal Language for Embedded Critical Software Development. In *TASE 2017.* IEEE Computer Society, 4–15.

[9] J.-L. Colaço and M. Pouzet. 2003. Clocks as First Class Abstract Types. In *EMSOFT 2003 (LNCS)*, Vol. 2855. Springer, 134–155.

[10] Coq Development Team. 2019. *The Coq proof assistant reference manual.* Inria. v. 8.9.

[11] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. 1991. The synchronous dataflow programming language LUSTRE. *Proc. IEEE* 79, 9 (Sept. 1991), 1305–1320.

[12] G. Hamon and M. Pouzet. 2000. Modular Resetting of Synchronous Data-Flow Programs. In *PPDP 2000.* 289–300.

[13] ISO/IEC 9899:1999(E) 1999. *Programming languages—C* (2 ed.). Standard. ISO/IEC.

[14] X. Leroy. 2009. Formal verification of a realistic compiler. *Comms. ACM* 52, 7 (2009), 107–115.

[15] M. Pouzet. 2006. *Lucid Synchrone, v. 3. Tutorial and reference manual.* Université Paris-Sud.

[16] G. Shi, Y. Gan, S. Shang, S. Wang, Y. Dong, and P.-C. Yew. 2017. A Formally Verified Sequentializer for Lustre-Like Concurrent Synchronous Data-Flow Programs. In *ICSE-C 2017.* IEEE, 109–111.