Verified Lustre Normalization with Node Subsampling

Timothy Bourke    Paul Jeanmaire
Basile Pesin      Marc Pouzet

Inria Paris
École normale supérieure, CNRS, PSL University

ESWEEK 2021 - EMSOFT
Tuesday, October 12
11:00am - 11:15am EDT
• Widely used in safety-critical applications: Aerospace, Defense, Rail Transportation, Heavy Equipment, Energy, Nuclear...

• Scade 6: Qualified compiler for a Lustre-like language

• Our work: Verified compilation in an Interactive Theorem Prover (Coq)
• Widely used in safety-critical applications: Aerospace, Defense, Rail Transportation, Heavy Equipment, Energy, Nuclear…
• Scade 6: Qualified compiler for a Lustre-like language
• Our work: Verified compilation in an Interactive Theorem Prover (Coq)
Verified Lustre Normalization with Node Subsampling

Bourke, Jeanmaire, Pesin, Pouzet
Lustre: example

“count **down** from $n$, restarting every time $res$ is true.”

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

Bourke, Jeanmaire, Pesin, Pouzet
Verified Lustre Normalization with Node Subsampling
“count down from $n$, restarting every time $res$ is true.”

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

<table>
<thead>
<tr>
<th>res</th>
<th>F</th>
<th>F</th>
<th>F</th>
<th>T</th>
<th>F</th>
<th>F</th>
<th>F</th>
<th>F</th>
<th>F</th>
<th>F</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>...</td>
</tr>
</tbody>
</table>
“count down from \( n \), restarting every time \( res \) is true.”

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

<table>
<thead>
<tr>
<th>res</th>
<th>F</th>
<th>F</th>
<th>F</th>
<th>T</th>
<th>F</th>
<th>F</th>
<th>F</th>
<th>F</th>
<th>F</th>
<th>F</th>
<th>…</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>…</td>
</tr>
<tr>
<td>cpt</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>…</td>
<td></td>
</tr>
</tbody>
</table>
“count down from $n$, restarting every time $res$ is true.”

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

<table>
<thead>
<tr>
<th>res</th>
<th>F</th>
<th>F</th>
<th>F</th>
<th>T</th>
<th>F</th>
<th>F</th>
<th>F</th>
<th>F</th>
<th>F</th>
<th>…</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>…</td>
</tr>
<tr>
<td>norm1$1$</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>…</td>
</tr>
<tr>
<td>cpt</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>…</td>
</tr>
</tbody>
</table>
“count down from \( n \), restarting every time \( res \) is true.”

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

<table>
<thead>
<tr>
<th>res</th>
<th>F</th>
<th>F</th>
<th>F</th>
<th>T</th>
<th>F</th>
<th>F</th>
<th>F</th>
<th>T</th>
<th>F</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>...</td>
</tr>
</tbody>
</table>

| norm2$1 | T | F | F | F | F | F | F | F | F | ... |
| norm2$2 | 0 | 5 | 4 | 3 | 5 | 4 | 3 | 2 | 1 | ... |
| norm1$1 | 6 | 5 | 4 | 3 | 5 | 4 | 3 | 2 | 1 | ... |
| cpt    | 6 | 5 | 4 | 6 | 5 | 4 | 3 | 2 | 1 | ... |
Unnesting & Distribution function

\[ \lfloor c \rfloor = ([c], []) \]
\[ \lfloor x \rfloor = ([x], []) \]
Unnesting & Distribution function

\[
\begin{align*}
[c] &= ([c], []) \\
[x] &= ([x], []) \\
[e_1 \oplus e_2] &= ([e'_1], eqs'_1) \leftarrow [e_1] \\
                    &\quad ([e'_2], eqs'_2) \leftarrow [e_2] \\
                    &\quad ([e'_1 \oplus e'_2], eqs'_1 \cup eqs'_2)
\end{align*}
\]
Unnesting & Distribution function

\[
\lfloor c \rfloor = ([c], [])
\]
\[
\lfloor x \rfloor = ([x], [])
\]
\[
\lfloor e_1 \oplus e_2 \rfloor = ([e'_1], eqs'_1) \leftarrow [e_1]
\]
\[
\quad ([e'_2], eqs'_2) \leftarrow [e_2]
\]
\[
\quad ([e'_1 \oplus e'_2], eqs'_1 \cup eqs'_2)
\]
\[
\lfloor (e_1, \ldots, e_n) \text{ fby } (f_1, \ldots, f_m) \rfloor = ([e'_1, \ldots, e'_k], eqs'_1) \leftarrow [e_1, \ldots, e_n]
\]
\[
\quad ([f'_1, \ldots, f'_k], eqs'_2) \leftarrow [f_1, \ldots, f_m]
\]
\[
\quad ([x_1, \ldots, x_k], [x_1 = e'_1 \text{ fby } f'_1, \ldots, x_k = e'_k \text{ fby } f'_k] \cup eqs'_1 \cup eqs'_2)
\]

\[
(x, y) = \begin{cases} 
(0, 0) & \text{if } \text{res} \\
((0, 0) \text{ fby } (x + 1, y - 1)) & \text{else}
\end{cases}
\]
Unnesting & Distribution function

\[ [c] = ([c], []) \]
\[ [x] = ([x], []) \]
\[ [e_1 \oplus e_2] = ([e_1'], eqs_1') \leftarrow [e_1] \]
\[ ([e_2'], eqs_2') \leftarrow [e_2] \]
\[ ([e_1' \oplus e_2'], eqs_1' \cup eqs_2') \]
\[ [(e_1, \ldots, e_n) \text{ fby} (f_1, \ldots, f_m)] = ([e_1', \ldots, e_k'], eqs_1') \leftarrow [e_1, \ldots, e_n] \]
\[ ([f_1', \ldots, f_k'], eqs_2') \leftarrow [f_1, \ldots, f_m] \]
\[ ([x_1, \ldots, x_k], [x_1 = e_1' \text{ fby} f_1', \ldots, x_k = e_k' \text{ fby} f_k'] \cup eqs_1' \cup eqs_2') \]
\[ [f(e_1, \ldots, e_n)] = ([e_1', \ldots, e_m'], eqs') \leftarrow [e_1, \ldots, e_n] \]
\[ ([x_1, \ldots, x_k], [(x_1, \ldots x_k) = f(e_1', \ldots, e_m')] \cup eqs') \]

\[(x, y) = \begin{cases} (0, 0) & \text{if res} \\ ((0, 0) \text{ fby} (x + 1, y - 1)) & \text{else} \end{cases} \]
\[ t_1 = 0 \text{ fby} (x + 1); \]
\[ t_2 = 0 \text{ fby} (y - 1); \]
\[ x = \begin{cases} 0 & \text{if res} \\ t_1 & \text{else} \end{cases}; \]
\[ y = \begin{cases} 0 & \text{if res} \\ t_2 & \text{else} \end{cases}; \]
Unnesting & Distribution in the Coq Proof Assistant

fresh identifier generation

• In OCaml:
  ```ocaml
  let next = ref 0;;
  let fresh () =
    next := !next + 1;
    "norm$" ^ (string_of_int !next);
  ``

• We are in a pure functional language

Use an explicit state (monad)

```coq
st ⊎ { (x, b) } fresh_ident b st = (x, st')
```

```coq
unnest_exp : exp nat -> exp
```

```coq
Fixpoint unnest_exp (e : exp) :
    Fresh (list exp × list equation) :=
```

```coq
let unnest_exp = λ e →
    match e with
    | EConst c → ret (EConst c, []);
    | EVar v → ret (EVar v, []);
    | EGroup sp e1 ann =
        do (e'), e1 = unnest_exp G false e
        ret (EGroup sp (hd e1) ann, eqp)
    | EGroup sp e1 e2 ann =
        do (e1', e2') = unnest_exp G false e;
        do (e', eqp) = unnest_exp G false e2;
        do (e', eqp) = unnest Exp G false e1;
        ret (EGroup sp (hd e1', e2') ann, eqp1+eqp2)
    | EIfte e1 e2 es es' anns in
        do (e1', eqp) = unnest_exp G false e1
        let env := unnest_env e1' es' anns in
        List.map (λ (x1, b1) → [x1, [b1]])
        (combine (List.map (λ (x, b) → x) (hd es)) env)
        (combine (List.map (λ (x, b) → x) (hd eqp)) env)
    | EWhile e1 e2 es anns =
        do (e1', eqp) = unnest_exp G false e1
        do (e1', eqp) = unnest_exp G false e2;
        let env := unnest_env e1' es' anns in
        if IsContr anns then
            ret (merge_env e1' env) eqp1+eqp2
        else
            do x1 = dig_for_anns (List.map (λ (x, y) → x) (hd y)) env;
            ret (List.map _ (λ (x, y) → x) (combine (List.map (λ (x, y) → x) (hd y)) env)) eqp1+eqp2
    | EIf c e1 e2 es es' anns =
        if IsDec c then
            ret (combine (List.map (λ (x, y) → x) (combine (List.map (λ (x, y) → x) (hd y)) env)) eqp1+eqp2)
        else
            do x1 = dig_for_anns (List.map (λ (x, y) → x) (combine (List.map (λ (x, y) → x) (hd y)) env)) eqp1+eqp2
```

Bourke, Jeanmaire, Pesin, Pouzet

Verified Lustre Normalization with Node Subsampling
In OCaml:

```ocaml
let next = ref 0;;
let fresh () =
  next := !next + 1;
"norm$"^(string_of_int !next);;
```
Fresh identifier generation

- In OCaml:

```ocaml
let next = ref 0;;
let fresh () =
  next := !next + 1;
"norm$"^(string_of_int !next);;
```

- We are in a pure functional language

- Use an explicit state (monad)

```
st ⊔ \{(x, b)\}

let fresh_ident b st = (x, st')

st ⊔ \{(x1, b1)\}

do x1 ← fresh_ident b1;

st ⊔ \{(x1, b1)\} ⊔ \{(x2, b2)\}

do x2 ← fresh_ident b2;
```

Fixpoint unnest_exp (e : exp) : Fresh (list exp * list equation) ann
Stream Semantics of Lustre

<table>
<thead>
<tr>
<th>res</th>
<th>F F F T F F F F F F F F F ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>6 6 6 6 6 6 6 6 6 6 ...</td>
</tr>
<tr>
<td>cpt</td>
<td>6 5 4 6 5 4 3 2 1 1 ...</td>
</tr>
</tbody>
</table>

every trigger { read inputs; calculate; write outputs; }
Stream Semantics of Lustre

<table>
<thead>
<tr>
<th>res</th>
<th>F F F T F F F F F F F ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>6 6 6 6 6 6 6 6 6 6 6 ...</td>
</tr>
<tr>
<td>cpt</td>
<td>6 5 4 6 5 4 3 2 1 ...</td>
</tr>
</tbody>
</table>

Inductive `sem_exp`:

\[ H(x) = vs \]

\[ H \vdash x \downarrow vs \]

\[ \text{Svar} \]

\[ \text{History} \rightarrow \text{exp} \rightarrow \text{list Stream} \rightarrow \text{Prop} \]

\[ \text{Svar: sem_var} H \ x \ vs \rightarrow \]

\[ \text{sem_exp} H (\text{Evar} \ x \ \text{ann}) [vs] \]

[...]
Stream Semantics of Lustre

<table>
<thead>
<tr>
<th>res</th>
<th>F F F T F F F F F ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>6 6 6 6 6 6 6 6 6 6 ...</td>
</tr>
<tr>
<td>npt</td>
<td>6 5 4 6 5 4 3 2 1 ...</td>
</tr>
</tbody>
</table>

every trigger { read inputs; calculate; write outputs; }

**Inductive sem_exp:**

\[ H(x) = vs \]

\[ \forall x \downarrow vs \]

| Svar: sem_var H x vs \rightarrow sem_exp H (Evar x ann) [vs] | [...]

**Seq**

\[ H \vdash es \downarrow H(xs) \]

\[ \forall x s = es \]

| Seq: Forall2 (sem_exp H) es ss \rightarrow Forall2 (sem_var H) xs (concat ss) \rightarrow sem_equation H (xs, es) |
Stream Semantics of Lustre

<table>
<thead>
<tr>
<th>res</th>
<th>F</th>
<th>F</th>
<th>F</th>
<th>T</th>
<th>F</th>
<th>F</th>
<th>F</th>
<th>F</th>
<th>F</th>
<th>F</th>
<th>F</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>...</td>
</tr>
<tr>
<td>cpt</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

every trigger { read inputs; calculate; write outputs; }

\[
\begin{align*}
\text{Svar} & \quad \frac{H(x) = \nu s}{H \vdash x \downarrow \nu s} \\
\text{Seq} & \quad \frac{H \vdash \text{es} \downarrow H(x) \text{s}}{H \vdash x \text{s} = \text{es}}
\end{align*}
\]

\[
\text{Inductive sem_exp:}
\]

\[
\begin{align*}
\text{History} & \rightarrow \text{exp} \rightarrow \text{list Stream} \\
& \rightarrow \text{Prop} := \\
& | \quad \text{Svar: sem_var H x vs} \\
& \quad \quad \text{sem_exp H (Evar x ann)}[\text{vs}] \quad […]
\end{align*}
\]

\[
\text{with sem_equation:}
\]

\[
\begin{align*}
\text{History} & \rightarrow \text{equation} \\
& \rightarrow \text{Prop} := \\
& | \quad \text{Seq: Forall2 (sem_exp H) es ss} \\
& \quad \quad \text{Forall2 (sem_var H) xs (concat ss)} \\
& \quad \quad \text{sem_equation H (xs, es)}
\end{align*}
\]

\[
\text{Snode} \quad \frac{H(n.\text{in}) = \text{xs}}{H(n.\text{out}) = \text{ys} \quad \forall \text{eq} \in n.\text{eqs}, \quad G, H \vdash \text{eq}}
\]

\[
\begin{align*}
\text{node}(G, f) & \vdash n \\
& G \vdash f(\text{xs}) \downarrow \text{ys}
\end{align*}
\]
Unnesting & Distribution – correctness

∀ f xs ys, G ⊢ f(xs) ↓ ys

⇒ [G] ⊢ f(xs) ↓ ys

Untyped Lustre

Lustre

transcription

NLustre

expression initialization

Stc

Obc

Clight

Assembly

Verified Lustre Normalization with Node Subsampling
Unnesting & Distribution – correctness

\[ \forall f \, \text{xs, ys, } G \vdash f(\text{xs}) \Downarrow \text{ys} \implies [G] \vdash f(\text{xs}) \Downarrow \text{ys} \]

\[ [e] = (\text{es'}, \text{eqs'}) \]

\[ G, H, bs \vdash e \Downarrow \text{vs} \]
Unnesting & Distribution – correctness

∀f xs ys, G ⊢ f(xs) ⇓ ys
⇒ [G] ⊢ f(xs) ⇓ ys

Unnesting & Distribution Correctness

transcription

NLustre

expression initialization

Stc

Obc

Clight

Assembly

 Verified Lustre Normalization with Node Subsampling

Bourke, Jeanmaire, Pesin, Pouzet
Expression Initialization

\[
\begin{align*}
\left[x = (e_0 \circ fby \ e)^{ck}\right]^{fby} & = \begin{cases} 
\text{xinit} = \text{true}^{ck} \circ fby \text{false}^{ck} ; \\
px = \text{def}_{ty}^{ck} \circ fby \ e ; \\
x = \text{if} \ x\text{init} \ \text{then} \ e_0 \ \text{else} \ px ; 
\end{cases}
\end{align*}
\]
Expression Initialization

\[
x = (e_0 \text{ by } e)^c_k\]

\[
\begin{align*}
x_{\text{init}} &= \text{true}^c_k \text{ by } \text{false}^c_k; \\
p_x &= \text{def}^c_k \text{ by } e; \\
x &= \text{if } x_{\text{init}} \text{ then } e_0 \text{ else } p_x;
\end{align*}
\]

**Optimization:** avoid introducing several init registers

- Registers are costly in the final imperative program
- Use state monad to remember init registers introduced:
  Fresh A \((\text{ann} \ast \text{bool})\)
- Complicates the correctness proof with a non-local invariant
Clock system correctness

<table>
<thead>
<tr>
<th>x = 0 fby (x + 1)</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>8</th>
<th>9</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>x when b</td>
<td>0</td>
<td>1</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

A special type system based on *clocks* ensures that sampling is used correctly; e.g., programs like `x + (x when b)` that require unbounded buffers are rejected at compile time.
Clock system correctness

A special type system based on clocks ensures that sampling is used correctly; e.g., programs like \( x + (x \text{ when } b) \) that require unbounded buffers are rejected at compile time.

\[
\begin{array}{c|c|c|c|c|c|c|c|c|c|}
  x = 0 & \text{fby} (x + 1) & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 8 & 9 & \ldots \\
  b & T & T & F & F & T & T & T & F & F & \ldots \\
  x \text{ when } b & 0 & 1 & 4 & 5 & 6 & & & & & \ldots \\
\end{array}
\]

Fig. 12. Alignment between a clock (stream bool) and an expression (stream svalue)
A special type system based on clocks ensures that sampling is used correctly; e.g., programs like $x + (x \ \text{when} \ b)$ that require unbounded buffers are rejected at compile time.

$$
\begin{array}{c}
\text{tl } H, \text{tl } bs \leftarrow e^c_k \downarrow s \\
H, bs \leftarrow ck \downarrow (T \cdot b) \\
H, bs \leftarrow e \downarrow \{v\} \cdot s \\
\hline
H, bs \leftarrow e^c_k \downarrow \{v\} \cdot s
\end{array}$$

$$
\begin{array}{c}
\text{tl } H, \text{tl } bs \leftarrow e^c_k \downarrow s \\
H, bs \leftarrow ck \downarrow (F \cdot b) \\
H, bs \leftarrow e \downarrow \{\circ\} \cdot s \\
\hline
H, bs \leftarrow e^c_k \downarrow \{\circ\} \cdot s
\end{array}
$$

Fig. 12. Alignment between a clock (stream bool) and an expression (stream svalue)
A special type system based on clocks ensures that sampling is used correctly; e.g., programs like $x + (x \text{ when } b)$ that require unbounded buffers are rejected at compile time.

$$
\begin{array}{c|ccccccccccc}
  x = 0 & f b y (x + 1) & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 8 & 9 & \ldots \\
  b & T & T & F & F & T & T & T & F & F & \ldots \\
  x \text{ when } b & 0 & 1 & 4 & 5 & 6 & & & & & & \ldots \\
\end{array}
$$

$$
\begin{array}{cccc}
  \text{tl } H, \text{tl } bs \vdash e^{ck} \downarrow s \\
  H, bs \vdash ck \downarrow T \cdot b \\
  H, bs \vdash e \downarrow \langle v \rangle \cdot s \\
\end{array}
\begin{array}{cccc}
  \text{tl } H, \text{tl } bs \vdash e^{ck} \downarrow s \\
  H, bs \vdash ck \downarrow F \cdot b \\
  H, bs \vdash e \downarrow \langle \rangle \cdot s \\
\end{array}
\begin{array}{c}
  H, bs \vdash e^{ck} \downarrow \langle \rangle \cdot s \\
\end{array}
$$

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

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

$$\text{node } f \left(x^{ck_1}_1, \ldots, x^{ck_n}_n\right) \text{ returns } \left(y^{ck'_1}_1, \ldots, y^{ck'_m}_m\right)$$

and semantics $f(s_1, \ldots, s_n) \downarrow s'_1, \ldots, s'_m$, with $bs = \text{base-of}(s_1, \ldots, s_n)$, in any environment $H$ in which input variables are associated and aligned with input streams, $H, bs \vdash x^{ck_1}_1 \downarrow s'_1, \ldots, x^{ck_n}_n \downarrow s_n$, 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, $H, bs \vdash y^{ck'_1}_1 \downarrow s'_1, \ldots, y^{ck'_m}_m \downarrow s'_m$. 

---

Bourke, Jeanmaire, Pesin, Pouzet

**Verified Lustre Normalization with Node Subsampling**
A special type system based on *clocks* ensures that sampling is used correctly; e.g., programs like \( x + (x \text{ when } b) \) that require unbounded buffers are rejected at compile time.

\[
\begin{array}{cccccccccc}
x = 0 & fby & (x + 1) & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 8 & 9 & \ldots \\
b & T & T & F & F & T & T & F & F & \ldots \\
x \text{ when } b & 0 & 1 & 4 & 5 & 6 & \ldots 
\end{array}
\]

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

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

\[
\text{node } f \left(x_1^{ck_1}, \ldots, x_n^{ck_n}\right) \text{ returns } (y_1^{ck_1'}, \ldots, y_m^{ck_m'})
\]

and semantics \( f(s_1, \ldots, s_n) \downarrow s_1', \ldots, s_m' \), with \( bs = \text{base-of}(s_1, \ldots, s_n) \), 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, \( H, bs \vdash y_1^{ck_1'} \downarrow s_1', \ldots, y_m^{ck_m'} \downarrow s_m' \).
Clock system correctness

\[
\begin{array}{c|c|c|c|c|c|c|c|c|c|c|\ldots}
   x = 0 & \text{fby} & (x + 1) \\
   b & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 8 & 9 & \ldots \\
\hline
   b \text{ when} & 0 & 1 & 4 & 5 & 6 & 8 & \text{F} & \text{F} & \text{\ldots} & \ldots
\end{array}
\]

A special type system based on clocks ensures that sampling is used correctly; e.g., programs like \( x + (x \text{ when } b) \) that require unbounded buffers are rejected at compile time.

\[
\begin{align*}
    \text{tl } H, \text{tl } bs \vdash e^{ck} \downarrow s \\
    H, bs \vdash ck \downarrow T \cdot b & \quad H, bs \vdash e \downarrow \langle v \rangle \cdot s \\
    \hline
    H, bs \vdash e^{ck} \downarrow \langle v \rangle \cdot s
\end{align*}
\]

\[
\begin{align*}
    \text{tl } H, \text{tl } bs \vdash e^{ck} \downarrow s \\
    H, bs \vdash ck \downarrow F \cdot b & \quad H, bs \vdash e \downarrow \langle \rangle \cdot s \\
    \hline
    H, bs \vdash e^{ck} \downarrow \langle \rangle \cdot s
\end{align*}
\]

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

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

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

and semantics \( f(s_1, \ldots, s_n) \downarrow s'_1, \ldots, s'_m \), with \( bs = \text{base-of}(s_1, \ldots, s_n) \), 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, \( H, bs \vdash y_1^{ck'_1} \downarrow s'_1, \ldots, y_m^{ck'_m} \downarrow s'_m \).
A special type system based on clocks ensures that sampling is used correctly; e.g., programs like \( x + (x \text{ when } b) \) that require unbounded buffers are rejected at compile time.

\[
\begin{align*}
x = 0 \quad \text{fby} \ (x + 1) & \quad 0 \quad T \quad 1 \quad T \quad 2 \quad F \quad 3 \quad F \quad 4 \quad T \quad 5 \quad T \quad 6 \quad F \quad 8 \quad F \quad 9 \quad \ldots \\
b & \quad 0 \quad 1 \quad 4 \quad 5 \quad 6 \quad \ldots \\
x \text{ when } b & \quad 0 \quad 1 \quad 4 \quad 5 \quad 6 \quad \ldots
\end{align*}
\]

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

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

\[
\text{node } f \left( x_{1}^{ck_1}, \ldots, x_{n}^{ck_n} \right) \quad \text{return } \left( y_{1}^{ck_1'}, \ldots, y_{m}^{ck_m'} \right)
\]

and semantics \( f(s_{1}, \ldots, s_{n}) \downarrow s'_{1}, \ldots, s'_{m} \), with \( bs = \text{base-of}(s_{1}, \ldots, s_{n}) \), 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, \( H, bs \vdash y_{1}^{ck_1'} \downarrow s'_{1}, \ldots, y_{m}^{ck_m'} \downarrow s'_{m} \).
to prove $P(x + y)$, we need $P(x)$ and $P(y)$
Clock system correctness – causality and proof

- to prove $P(x + y)$, we need $P(x)$ and $P(y)$
- induction on equations is not enough: $(x, y) = (42, x)$
to prove $P(x + y)$, we need $P(x)$ and $P(y)$
induction on equations is not enough: $(x, y) = (42, x)$
• to prove $P(x + y)$, we need $P(x)$ and $P(y)$
• induction on equations is not enough: $(x, y) = (42, x)$
• to prove $P(x + y)$, we need $P(x)$ and $P(y)$
• induction on equations is not enough: $(x, y) = (42, x)$

\[
\begin{align*}
\text{let } (t_1, t_2) &= \begin{cases} 
(x + 1, t_1) & \text{if } b \\
(x - 1, -t_1) & \text{else}
\end{cases} \\
y &= (0 \text{ fby } y) + (t_1 \times t_2)
\end{align*}
\]
to prove $P(x + y)$, we need $P(x)$ and $P(y)$

induction on equations is not enough: $(x, y) = (42, x)$

causal node $\rightarrow$ acyclic graph

```plaintext
node f(b : bool; x : int) returns (y : int)
var t1, t2 : int;
let
  (t1, t2) = if b
    then (x + 1, t1)
    else (x - 1, -t1);
y = (0 fby y) + (t1 * t2);
tel
```

graphs difficult to handle in a proof assistant: 1200 lines of Coq
Clock system correctness – causality and proof

• to prove $P(x + y)$, we need $P(x)$ and $P(y)$
• induction on equations is not enough: $(x, y) = (42, x)$
• causal node $\rightarrow$ acyclic graph

node f(b : bool; x : int) returns (y : int)
var t1, t2 : int;
let
  (t1, t2) = if b
              then (x + 1, t1)
              else (x - 1, -t1);
y = (0 fby y) + (t1 * t2);
tel

• graphs difficult to handle in a proof assistant: 1200 lines of Coq
• induction on a topological ordering of the nodes of the graph
• look only to the left of fby: the fby operator forces alignment
Clock system correctness – causality and proof

- to prove $P(x + y)$, we need $P(x)$ and $P(y)$
- induction on equations is not enough: $(x, y) = (42, x)$
- causal node $\rightarrow$ acyclic graph

```plaintext
node f(b : bool; x : int) returns (y : int)
var t1, t2 : int;
let
    (t1, t2) = if b 
        then (x + 1, t1) 
        else (x - 1, -t1);
    y = (0 fby y) + (t1 * t2);
```

- graphs difficult to handle in a proof assistant: 1200 lines of Coq
- induction on a topological ordering of the nodes of the graph
- look only to the left of `fby`: the `fby` operator forces alignment
- intricate proof: around 2000 lines of Coq proof script
node current(d : int; ck : bool; x : int when ck)
node current(d : int; ck : bool; x : int when ck)
always present
Node Subsampling

node current(d : int; ck : bool; x : int when ck)

always present only present when ck is
node current(d : int; ck : bool; x : int when ck)
  always present  only present when ck is

Compile an instance of this node to Obc:

```plaintext
if (ck) {
  elab$4 := exp;
};
time := current(i1).step(0, ck, elab$4)
```
Node Subsampling

node current(d : int; ck : bool; x : int when ck)

always present

only present when ck is

Compile an instance of this node to Obc:

if (ck) {
    elab$4 := exp;
}

only defined when ck = true

time := current(i1).step(0, ck, elab$4)
node current(d : int; ck : bool; x : int when ck)

always present

only present when ck is

Compile an instance of this node to Obc:

if (ck) {
    elab$4 := exp;
}
;

\[\text{time} := \text{current}(\text{i1}).\text{step}(0, \text{ck}, \text{elab$4})\]

6.5.2.2 Function calls

Constraints

1. The expression that denotes the called function\(^{80}\) shall have type pointer to function returning \texttt{void} or returning an object type other than an array type.

2. If the expression that denotes the called function has a type that includes a prototype, the number of arguments shall agree with the number of parameters. Each argument shall have a type such that its value may be assigned to an object with the unqualified version of the type of its corresponding parameter.

Semantics

3. A postfix expression followed by parentheses () containing a possibly empty, comma-separated list of expressions is a function call. The postfix expression denotes the called function. The list of expressions specifies the arguments to the function.

4. An argument may be an expression of any object type. In preparing for the call to a function, the arguments are evaluated, and each parameter is assigned the value of the corresponding argument.\(^{81}\)

5. If the expression that denotes the called function has type pointer to function returning an object type, the function may change the value determined as specified in 6.8.6.4.

6. If the expression that denotes the called function has type pointer to function returning an object type other than an array type, these are called the default arguments.

7. If the expression that denotes the called function has type pointer to function returning an object type other than an array type, these are called the default arguments.

8. If the function is defined with a type that does not include a prototype, and the types of the arguments are not compatible with the types of the parameters, the behavior is undefined.

9. If the function is defined with a type that includes a prototype, and the types of the arguments differ from the number of parameters, the behavior is undefined.

10. If the expression that denotes the called function has a type that does not include a prototype, or the expression is a function call, the value returned is the result of evaluating the function, and may be assigned to an object with the unqualified version of the type of its corresponding parameter.

11. If the expression that denotes the called function has a type that includes a prototype, the value of the expression that denotes the called function is adjusted to have a pointer type as described in 6.9.1.

12. If the expression that denotes the called function has a prototype, the function is returned.

13. If the expression that denotes the called function has type pointer to function returning an object type other than an array type, the function is returned.

14. If the expression that denotes the called function has a type that includes a prototype, and the expression is a function call, the value may be assigned to an object with the unqualified version of the type of its corresponding parameter.

15. If the expression that denotes the called function has a type that does not include a prototype, or the expression is a function call, the value returned is the result of evaluating the function, and may be assigned to an object with the unqualified version of the type of its corresponding parameter.

16. If the expression that denotes the called function has a type that includes a prototype, the value of the expression that denotes the called function is adjusted to have a pointer type as described in 6.9.1.

17. If the expression that denotes the called function has a prototype, the function is returned.

18. If the expression that denotes the called function has type pointer to function returning an object type other than an array type, the function is returned.

19. If the expression that denotes the called function has a type that includes a prototype, and the expression is a function call, the value may be assigned to an object with the unqualified version of the type of its corresponding parameter.

20. If the expression that denotes the called function has a type that does not include a prototype, or the expression is a function call, the value returned is the result of evaluating the function, and may be assigned to an object with the unqualified version of the type of its corresponding parameter.

21. If the expression that denotes the called function has a type that includes a prototype, the value of the expression that denotes the called function is adjusted to have a pointer type as described in 6.9.1.

22. If the expression that denotes the called function has a prototype, the function is returned.

23. If the expression that denotes the called function has type pointer to function returning an object type other than an array type, the function is returned.

24. If the expression that denotes the called function has a type that includes a prototype, and the expression is a function call, the value may be assigned to an object with the unqualified version of the type of its corresponding parameter.

25. If the expression that denotes the called function has a type that does not include a prototype, or the expression is a function call, the value returned is the result of evaluating the function, and may be assigned to an object with the unqualified version of the type of its corresponding parameter.

26. If the expression that denotes the called function has a type that includes a prototype, the value of the expression that denotes the called function is adjusted to have a pointer type as described in 6.9.1.

27. If the expression that denotes the called function has a prototype, the function is returned.

28. If the expression that denotes the called function has type pointer to function returning an object type other than an array type, the function is returned.

29. If the expression that denotes the called function has a type that includes a prototype, and the expression is a function call, the value may be assigned to an object with the unqualified version of the type of its corresponding parameter.

30. If the expression that denotes the called function has a type that does not include a prototype, or the expression is a function call, the value returned is the result of evaluating the function, and may be assigned to an object with the unqualified version of the type of its corresponding parameter.

31. If the expression that denotes the called function has a type that includes a prototype, the value of the expression that denotes the called function is adjusted to have a pointer type as described in 6.9.1.

32. If the expression that denotes the called function has a prototype, the function is returned.

33. If the expression that denotes the called function has type pointer to function returning an object type other than an array type, the function is returned.

34. If the expression that denotes the called function has a type that includes a prototype, and the expression is a function call, the value may be assigned to an object with the unqualified version of the type of its corresponding parameter.

35. If the expression that denotes the called function has a type that does not include a prototype, or the expression is a function call, the value returned is the result of evaluating the function, and may be assigned to an object with the unqualified version of the type of its corresponding parameter.

36. If the expression that denotes the called function has a type that includes a prototype, the value of the expression that denotes the called function is adjusted to have a pointer type as described in 6.9.1.

37. If the expression that denotes the called function has a prototype, the function is returned.

38. If the expression that denotes the called function has type pointer to function returning an object type other than an array type, the function is returned.

39. If the expression that denotes the called function has a type that includes a prototype, and the expression is a function call, the value may be assigned to an object with the unqualified version of the type of its corresponding parameter.

40. If the expression that denotes the called function has a type that does not include a prototype, or the expression is a function call, the value returned is the result of evaluating the function, and may be assigned to an object with the unqualified version of the type of its corresponding parameter.

41. If the expression that denotes the called function has a type that includes a prototype, the value of the expression that denotes the called function is adjusted to have a pointer type as described in 6.9.1.

42. If the expression that denotes the called function has a prototype, the function is returned.

43. If the expression that denotes the called function has type pointer to function returning an object type other than an array type, the function is returned.

44. If the expression that denotes the called function has a type that includes a prototype, and the expression is a function call, the value may be assigned to an object with the unqualified version of the type of its corresponding parameter.

45. If the expression that denotes the called function has a type that does not include a prototype, or the expression is a function call, the value returned is the result of evaluating the function, and may be assigned to an object with the unqualified version of the type of its corresponding parameter.

46. If the expression that denotes the called function has a type that includes a prototype, the value of the expression that denotes the called function is adjusted to have a pointer type as described in 6.9.1.

47. If the expression that denotes the called function has a prototype, the function is returned.

48. If the expression that denotes the called function has type pointer to function returning an object type other than an array type, the function is returned.

49. If the expression that denotes the called function has a type that includes a prototype, and the expression is a function call, the value may be assigned to an object with the unqualified version of the type of its corresponding parameter.

50. If the expression that denotes the called function has a type that does not include a prototype, or the expression is a function call, the value returned is the result of evaluating the function, and may be assigned to an object with the unqualified version of the type of its corresponding parameter.

51. If the expression that denotes the called function has a type that includes a prototype, the value of the expression that denotes the called function is adjusted to have a pointer type as described in 6.9.1.

52. If the expression that denotes the called function has a prototype, the function is returned.

53. If the expression that denotes the called function has type pointer to function returning an object type other than an array type, the function is returned.

54. If the expression that denotes the called function has a type that includes a prototype, and the expression is a function call, the value may be assigned to an object with the unqualified version of the type of its corresponding parameter.

55. If the expression that denotes the called function has a type that does not include a prototype, or the expression is a function call, the value returned is the result of evaluating the function, and may be assigned to an object with the unqualified version of the type of its corresponding parameter.

56. If the expression that denotes the called function has a type that includes a prototype, the value of the expression that denotes the called function is adjusted to have a pointer type as described in 6.9.1.

57. If the expression that denotes the called function has a prototype, the function is returned.

58. If the expression that denotes the called function has type pointer to function returning an object type other than an array type, the function is returned.

59. If the expression that denotes the called function has a type that includes a prototype, and the expression is a function call, the value may be assigned to an object with the unqualified version of the type of its corresponding parameter.

60. If the expression that denotes the called function has a type that does not include a prototype, or the expression is a function call, the value returned is the result of evaluating the function, and may be assigned to an object with the unqualified version of the type of its corresponding parameter.

61. If the expression that denotes the called function has a type that includes a prototype, the value of the expression that denotes the called function is adjusted to have a pointer type as described in 6.9.1.

62. If the expression that denotes the called function has a prototype, the function is returned.

63. If the expression that denotes the called function has type pointer to function returning an object type other than an array type, the function is returned.

64. If the expression that denotes the called function has a type that includes a prototype, and the expression is a function call, the value may be assigned to an object with the unqualified version of the type of its corresponding parameter.

65. If the expression that denotes the called function has a type that does not include a prototype, or the expression is a function call, the value returned is the result of evaluating the function, and may be assigned to an object with the unqualified version of the type of its corresponding parameter.
node current(d : int; ck : bool; x : int when ck)

always present

only present when ck is

Compile an instance of this node to Obc:

if (ck) {
    elab$4 := exp;
}

only defined when ck = true

time := current(i1).step(0, ck, elab$4)
Node Subsampling

```c
node current(d : int; ck : bool; x : int when ck)
```

always present

only present when ck is

Compile an instance of this node to Obc:

```c
if (ck) {
    elab$4 := exp;
}
```

only defined when ck = true

- Already formalized in CompCert’s Clight semantics
- Appears as a proof obligation in our end-to-end proof
Node Subsampling

\[
\text{node current}(d : \text{int}; \ ck : \text{bool}; \ x : \text{int} \ \text{when} \ \text{ck})
\]

always present \hspace{2cm} \text{only present when ck is}\hspace{2cm} \\

Compile an instance of this node to Obc:

```plaintext
if (ck) {
    \text{elab}\$4 := \text{exp};
};
\text{time} := \text{current}(i1).\text{step}(0, \ \text{ck}, \ \text{elab}\$4)
```
**Node Subsampling**

```
node current(d : int; ck : bool; x : int when ck) {
    always present
}

Compile an instance of this node to Obc:
```

```java
if (ck) {
    elab$4 := exp;
}

};

time := current(i1).step(0, ck, elab$4)
```

1. Add validity assertions during compilation:

```java
if (ck) {
    elab$4 := exp;
}

};

time := current(i1).step(0, ⟨ck⟩, elab$4)
```
Node Subsampling

node current(d : int; ck : bool; x : int when ck)
  always present
  only present when ck is

Compile an instance of this node to Obc:

if (ck) {
  elab$4 := exp;
};
time := current(i1).step(0, ck, elab$4)

1. Add validity assertions during compilation:

if (ck) {
  elab$4 := exp;
};
time := current(i1).step(0, ⟨ck⟩, elab$4)

2. Extra compilation pass to initialize variables:

if (ck) {
  elab$4 := exp;
} else {
  elab$4 := 0;
};
time := current(i1).step(0, ⟨ck⟩, ⟨elab$4⟩)
Node Subsampling

\[
\text{node current}(d: \text{int}; \ ck: \text{bool}; \ x: \text{int} \ \text{when} \ ck) \\quad \underline{\text{always present}} \quad \underline{\text{only present when} \ ck \ \text{is}}
\]

Compile an instance of this node to Obc:

\[
\text{if} \ (ck) \ {\}
\quad \text{elab}\$4 := \text{exp};
\]

\[
time := \text{current}(i1).\text{step}(0, \ ck, \ \text{elab}\$4)
\]

1. Add validity assertions during compilation:

\[
\text{if} \ (ck) \ {\}
\quad \text{elab}\$4 := \text{exp};
\]

\[
time := \text{current}(i1).\text{step}(0, \\
\quad \langle \text{ck} \rangle, \ \text{elab}\$4)
\]

- Guarantees that variables in function calls are always defined.
- Recover Obc $\rightsquigarrow$ Clight proof
- Programs without subsampling are unchanged

2. Extra compilation pass to initialize variables:

\[
\text{if} \ (ck) \ {\}
\quad \text{elab}\$4 := \text{exp};
\]

\[
\text{else} \ {\}
\quad \text{elab}\$4 := 0;
\]

\[
time := \text{current}(i1).\text{step}(0, \langle \text{ck} \rangle, \\
\quad \langle \text{elab}\$4 \rangle)
\]
• More expressive source language and associated semantic model
• Formally verified normalization algorithm
  » Separate concerns and proofs over 3 functions
  » Requires correctness of clock system
• Allow node subsampling in source language
  » Add “validity assertions” and explicit initialization
• End-to-end machine-checked proof connecting the dataflow semantics of an expressive source language with the low-level assembly semantics.
• Source code and online demo: https://velus.inria.fr