Verified Lustre Normalization with Node Subsampling

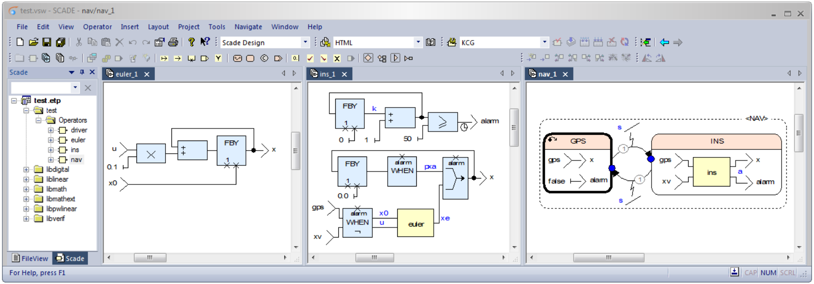
Timothy Bourke   Paul Jeanmaire
Basile Pesin   Marc Pouzet

Inria Paris

École normale supérieure, CNRS, PSL University

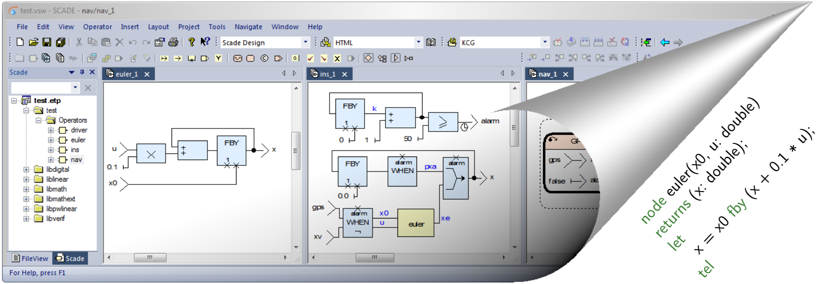
ESWEEK 2021 - EMSOFT
Tuesday, October 12
11:00am - 11:15am EDT

Block-Diagram Languages for Embedded Systems

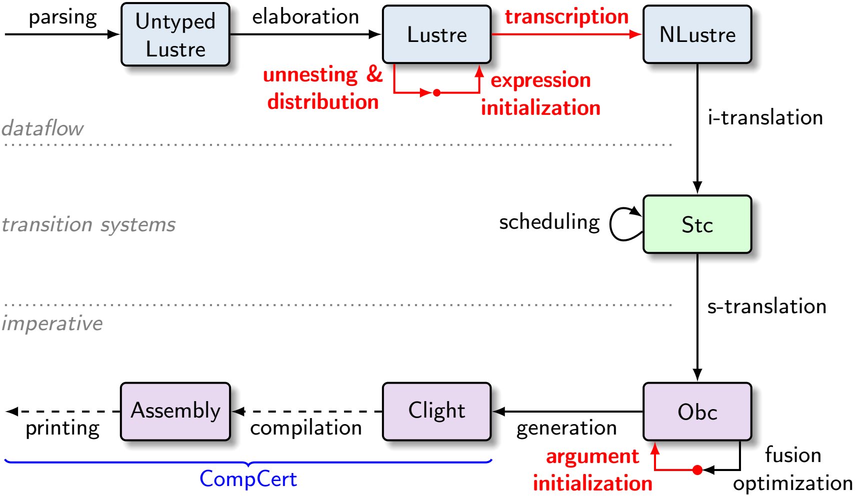


- Widely used in safety-critical applications: Aerospace, Defense, Rail Transportation, Heavy Equipment, Energy, Nuclear. . .
- Scade 6: Qualified compiler for a Lustre-like language
- Our work: Verified compilation in an Interactive Theorem Prover (Coq)

Block-Diagram Languages for Embedded Systems



- 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)



parsing
Untyped Lustre
elaboration
Lustre
transcription
NLustre
unnesting & distribution
expression initialization
dataflow
i-translation
transition systems
scheduling
Stc
s-translation
imperative
Assembly
Clight
Obc
printing
compilation
generation
argument initialization
fusion optimization
CompCert

Lustre: example

"**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
```

Lustre: example

"**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
```

res	F	F	F	T	F	F	F	F	F	⋯
n	6	6	6	6	6	6	6	6	6	⋯

Lustre: example

"**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
```

res	F	F	F	T	F	F	F	F	F	⋯
n	6	6	6	6	6	6	6	6	6	⋯
cpt	6	5	4	6	5	4	3	2	1	⋯

Lustre: example

"**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
```

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

res	F	F	F	T	F	F	F	F	F	···
n	6	6	6	6	6	6	6	6	6	···
norm1$1	6	5	4	3	5	4	3	2	1	···
cpt	6	5	4	6	5	4	3	2	1	···

Lustre: example

"**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
```

```
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
```

res	F	F	F	T	F	F	F	T	F	···
n	6	6	6	6	6	6	6	6	6	···
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

$$\lfloor \mathsf{c} \rfloor = ([\mathsf{c}], [\,])$$
$$\lfloor \mathsf{x} \rfloor = ([\mathsf{x}], [\,])$$

Unnesting & Distribution function

$$
\begin{aligned}
\lfloor \mathsf{c} \rfloor &= ([\mathsf{c}], [\,]) \\
\lfloor \mathsf{x} \rfloor &= ([\mathsf{x}], [\,]) \\
\lfloor e_1 \oplus e_2 \rfloor &= ([e_1'], \mathsf{eqs}_1') \leftarrow \lfloor e_1 \rfloor \\
&\quad\ ([e_2'], \mathsf{eqs}_2') \leftarrow \lfloor e_2 \rfloor \\
&\quad\ ([e_1' \oplus e_2'], \mathsf{eqs}_1' \ \cup\ \mathsf{eqs}_2')
\end{aligned}
$$

Unnesting & Distribution function

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

```
(x, y) = if res
         then (0, 0)
         else ((0, 0) fby (x + 1, y − 1));
```

```
t1 = 0 fby (x + 1);
t2 = 0 fby (y − 1);
x = if res then 0 else t1;
y = if res then 0 else t2;
```

Unnesting & Distribution function

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

```
(x, y) = if res
         then (0, 0)
         else ((0, 0) fby (x + 1, y − 1));
```

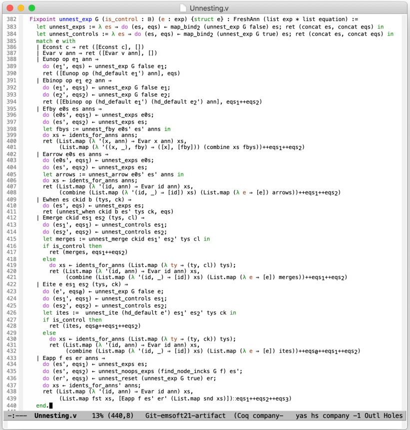
```
t1 = 0 fby (x + 1);
t2 = 0 fby (y − 1);
x = if res then 0 else t1;
y = if res then 0 else t2;
```

Unnesting & Distribution in the Coq Proof Assistant

Unnesting.v

```
Fixpoint unnest_exp G (is_control : 𝔹) (e : exp) {struct e} : FreshAnn (list exp * list equation) :=
  let unnest_exps := λ es ⇒ do (es, eqs) ← map_bind2 (unnest_exp G false) es; ret (concat es, concat eqs) in
  let unnest_controls := λ es ⇒ do (es, eqs) ← map_bind2 (unnest_exp G true) es; ret (concat es, concat eqs) in
  match e with
   | Econst c ⇒ ret ([Econst c], [])
   | Evar v ann ⇒ ret ([Evar v ann], [])
   | Eunop op e1 ann ⇒
     do (e1', eqs) ← unnest_exp G false e1;
     ret ([Eunop op (hd_default e1') ann], eqs)
   | Ebinop op e1 e2 ann ⇒
     do (e1', eqs1) ← unnest_exp G false e1;
     do (e2', eqs2) ← unnest_exp G false e2;
     ret ([Ebinop op (hd_default e1') (hd_default e2') ann], eqs1++eqs2)
   | Efby e0s es anns ⇒
     do (e0s', eqs1) ← unnest_exps e0s;
     do (es', eqs2) ← unnest_exps es;
     let fbys := unnest_fby e0s' es' anns in
     do xs ← idents_for_anns anns;
     ret (List.map (λ '(x, ann) ⇒ Evar x ann) xs,
          (List.map (λ '((x, _), fby) ⇒ ([x], [fby])) (combine xs fbys))++eqs1++eqs2)
   | Earrow e0s es anns ⇒
     do (e0s', eqs1) ← unnest_exps e0s;
     do (es', eqs2) ← unnest_exps es;
     let arrows := unnest_arrow e0s' es' anns in
     do xs ← idents_for_anns anns;
     ret (List.map (λ '(id, ann) ⇒ Evar id ann) xs,
          (combine (List.map (λ '(id, _) ⇒ [id]) xs) (List.map (λ e ⇒ [e]) arrows))++eqs1++eqs2)
   | Ewhen es ckid b (tys, ck) ⇒
     do (es', eqs) ← unnest_exps es;
     ret (unnest_when ckid b es' tys ck, eqs)
   | Emerge ckid es1 es2 (tys, cl) ⇒
     do (es1', eqs1) ← unnest_controls es1;
     do (es2', eqs2) ← unnest_controls es2;
     let merges := unnest_merge ckid es1' es2' tys cl in
     if is_control then
       ret (merges, eqs1++eqs2)
     else
       do xs ← idents_for_anns (List.map (λ ty ⇒ (ty, cl)) tys);
       ret (List.map (λ '(id, ann) ⇒ Evar id ann) xs,
            (combine (List.map (λ '(id, _) ⇒ [id]) xs) (List.map (λ e ⇒ [e]) merges))++eqs1++eqs2)
   | Eite e es1 es2 (tys, ck) ⇒
     do (e', eqs0) ← unnest_exp G false e;
     do (es1', eqs1) ← unnest_controls es1;
     do (es2', eqs2) ← unnest_controls es2;
     let ites :=  unnest_ite (hd_default e') es1' es2' tys ck in
     if is_control then
       ret (ites, eqs0++eqs1++eqs2)
     else
       do xs ← idents_for_anns (List.map (λ ty ⇒ (ty, ck)) tys);
       ret (List.map (λ '(id, ann) ⇒ Evar id ann) xs,
            (combine (List.map (λ '(id, _) ⇒ [id]) xs) (List.map (λ e ⇒ [e]) ites))++eqs0++eqs1++eqs2)
   | Eapp f es er anns ⇒
     do (es', eqs1) ← unnest_exps es;
     do (es', eqs2) ← unnest_noops_exps (find_node_incks G f) es';
     do (er', eqs3) ← unnest_reset (unnest_exp G true) er;
     do xs ← idents_for_anns' anns;
     ret (List.map (λ '(id, ann) ⇒ Evar id ann) xs,
          (List.map fst xs, [Eapp f es' er' (List.map snd xs)])::eqs1++eqs2++eqs3)
  end.
```

-:--- Unnesting.v   13% (440,8)   Git-emsoft21-artifact (Coq company-   yas hs company -1 Outl Holes

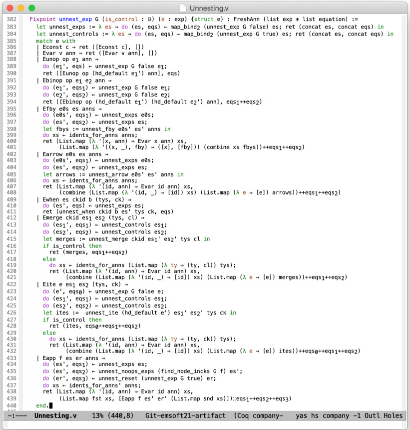


```
Fixpoint unnest_exp G (is_control : 𝔹) (e : exp) {struct e} : FreshAnn (list exp * list equation) :=
  let unnest_exps := λ es ⇒ do (es, eqs) ← map_bind2 (unnest_exp G false) es; ret (concat es, concat eqs) in
  let unnest_controls := λ es ⇒ do (es, eqs) ← map_bind2 (unnest_exp G true) es; ret (concat es, concat eqs) in
  match e with
  | Econst c ⇒ ret ([Econst c], [])
  | Evar v ann ⇒ ret ([Evar v ann], [])
  | Eunop op e1 ann ⇒
    do (e1', eqs) ← unnest_exp G false e1;
    ret ([Eunop op (hd_default e1') ann], eqs)
  | Ebinop op e1 e2 ann ⇒
    do (e1', eqs1) ← unnest_exp G false e1;
    do (e2', eqs2) ← unnest_exp G false e2;
    ret ([Ebinop op (hd_default e1') (hd_default e2') ann], eqs1++eqs2)
  | Efby e0s es anns ⇒
    do (e0s', eqs1) ← unnest_exps e0s;
    do (es', eqs2) ← unnest_exps es;
    let fbys := unnest_fby e0s' es' anns in
    do xs ← idents_for_anns anns;
    ret (List.map (λ '(x, ann) ⇒ Evar x ann) xs,
         (List.map (λ '((x, _), fby) ⇒ ([x], [fby])) (combine xs fbys))++eqs1++eqs2)
  | Earrow e0s es anns ⇒
    do (e0s', eqs1) ← unnest_exps e0s;
    do (es', eqs2) ← unnest_exps es;
    let arrows := unnest_arrow e0s' es' anns in
    do xs ← idents_for_anns anns;
    ret (List.map (λ '(id, ann) ⇒ Evar id ann) xs,
         (combine (List.map (λ '(id, _) ⇒ [id]) xs) (List.map (λ e ⇒ [e]) arrows))++eqs1++eqs2)
  | Ewhen es ckid b (tys, ck) ⇒
    do (es', eqs) ← unnest_exps es;
    ret (unnest_when ckid b es' tys ck, eqs)
  | Emerge ckid es1 es2 (tys, cl) ⇒
    do (es1', eqs1) ← unnest_controls es1;
    do (es2', eqs2) ← unnest_controls es2;
    let merges := unnest_merge ckid es1' es2' tys cl in
    if is_control then
      ret (merges, eqs1++eqs2)
    else
      do xs ← idents_for_anns (List.map (λ ty ⇒ (ty, cl)) tys);
      ret (List.map (λ '(id, ann) ⇒ Evar id ann) xs,
           (combine (List.map (λ '(id, _) ⇒ [id]) xs) (List.map (λ e ⇒ [e]) merges))++eqs1++eqs2)
  | Eite e es1 es2 (tys, ck) ⇒
    do (e', eqs0) ← unnest_exp G false e;
    do (es1', eqs1) ← unnest_controls es1;
    do (es2', eqs2) ← unnest_controls es2;
    let ites :=  unnest_ite (hd_default e') es1' es2' tys ck in
    if is_control then
      ret (ites, eqs0++eqs1++eqs2)
    else
      do xs ← idents_for_anns (List.map (λ ty ⇒ (ty, ck)) tys);
      ret (List.map (λ '(id, ann) ⇒ Evar id ann) xs,
           (combine (List.map (λ '(id, _) ⇒ [id]) xs) (List.map (λ e ⇒ [e]) ites))++eqs0++eqs1++eqs2)
  | Eapp f es er anns ⇒
    do (es', eqs1) ← unnest_exps es;
    do (es', eqs2) ← unnest_noops_exps (find_node_incks G f) es';
    do (er', eqs3) ← unnest_reset (unnest_exp G true) er;
    do xs ← idents_for_anns' anns;
    ret (List.map (λ '(id, ann) ⇒ Evar id ann) xs,
         (List.map fst xs, [Eapp f es' er' (List.map snd xs)])::eqs1++eqs2++eqs3)
  end.
-:---  Unnesting.v    13% (440,8)   Git-emsoft21-artifact  (Coq company-    yas hs company -1 Outl Holes
```

Fresh identifier generation

- In OCaml:

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



Fresh identifier generation

- In 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)

$$\boxed{st} \longrightarrow \boxed{st \uplus \{(x, b)\}}$$

```
fresh_ident b st = (x, st')
```

$$\boxed{st}$$

```
do x1 ← fresh_ident b1;
```

$$\boxed{st \uplus \{(x1, b1)\}}$$

```
do x2 ← fresh_ident b2;
```

$$\boxed{st \uplus \{(x1, b1)\} \uplus \{(x2, b2)\}}$$

```
Fixpoint unnest_exp (e : exp) : Fresh (list exp * list equation) ann
```

Stream Semantics of Lustre

`res`	F	F	F	T	F	F	F	F	F	$\cdots$
`n`	6	6	6	6	6	6	6	6	6	$\cdots$
`cpt`	6	5	4	6	5	4	3	2	1	$\cdots$

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

Stream Semantics of Lustre

res	F	F	F	T	F	F	F	F	F	...
n	6	6	6	6	6	6	6	6	6	...
cpt	6	5	4	6	5	4	3	2	1	...

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

$$\text{Svar}\ \frac{H(x) = vs}{H \vdash x \Downarrow vs}$$

```
Inductive sem_exp:
  History → exp → list Stream → Prop :=
| Svar: sem_var H x vs →
        sem_exp H (Evar x ann) [vs]        [...]
```

Stream Semantics of Lustre

res	F	F	F	T	F	F	F	F	F	...
n	6	6	6	6	6	6	6	6	6	...
cpt	6	5	4	6	5	4	3	2	1	...

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

$$\text{Svar}\ \frac{H(x) = vs}{H \vdash x \Downarrow vs}$$

```
Inductive sem_exp:
  History → exp → list Stream → Prop :=
| Svar: sem_var H x vs →
        sem_exp H (Evar x ann) [vs]       [...]
```

$$\text{Seq}\ \frac{H \vdash \mathsf{es} \Downarrow H(\mathsf{xs})}{H \vdash \mathsf{xs} = \mathsf{es}}$$

```
with sem_equation:
  History → equation → Prop :=
| Seq: Forall2 (sem_exp H) es ss →
       Forall2 (sem_var H) xs (concat ss) →
       sem_equation H (xs, es)
```

Stream Semantics of Lustre

res	F	F	F	T	F	F	F	F	F	...
n	6	6	6	6	6	6	6	6	6	...
cpt	6	5	4	6	5	4	3	2	1	...

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

$$\text{Svar}\ \frac{H(x) = vs}{H \vdash x \Downarrow vs}$$

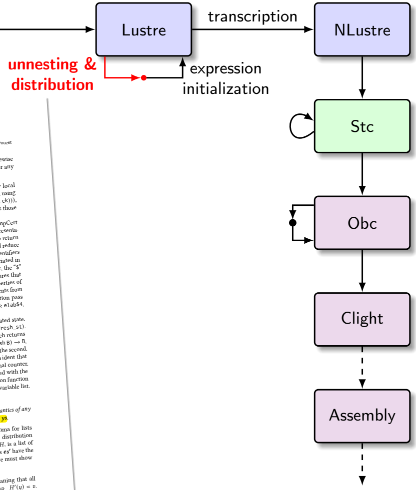
```
Inductive sem_exp:
  History → exp → list Stream → Prop :=
| Svar: sem_var H x vs →
        sem_exp H (Evar x ann) [vs]        [...]
```

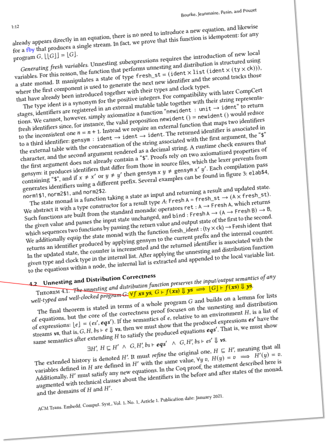
$$\text{Seq}\ \frac{H \vdash \mathrm{es} \Downarrow H(\mathrm{xs})}{H \vdash \mathrm{xs} = \mathrm{es}}$$

```
with sem_equation:
  History → equation → Prop :=
| Seq: Forall2 (sem_exp H) es ss →
       Forall2 (sem_var H) xs (concat ss) →
       sem_equation H (xs, es)
```

$$\text{Snode}\ \frac{\begin{array}{c}\mathtt{node}(G, f) \doteq n \\ H(n.\mathbf{in}) = \mathrm{xs} \qquad H(n.\mathbf{out}) = \mathrm{ys} \qquad \forall eq \in n.\mathbf{eqs},\ G, H \vdash eq\end{array}}{G \vdash f(\mathrm{xs}) \Downarrow \mathrm{ys}}$$

Unnesting & Distribution – correctness





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:* $\forall f\ \boldsymbol{xs}\ \boldsymbol{ys},\ G \vdash f(\boldsymbol{xs}) \Downarrow \boldsymbol{ys} \implies \lfloor G\rfloor \vdash f(\boldsymbol{xs}) \Downarrow \boldsymbol{ys}$.

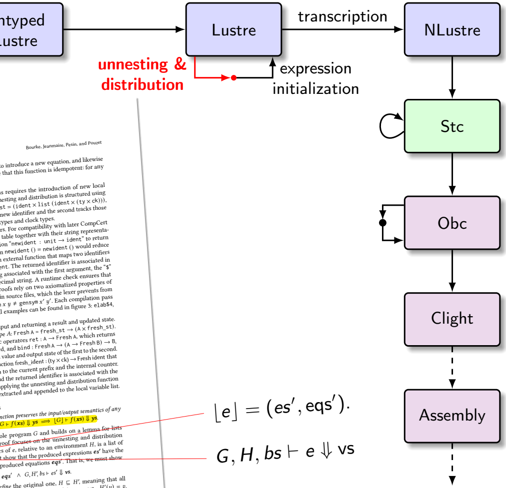
The final theorem is stated in terms of a whole program G and builds on a lemma for lists of equations, but the core of the correctness proof focuses on the unnesting and distribution of expressions: $\lfloor e\rfloor = (es', \boldsymbol{eqs}')$. If the semantics of e, relative to an environment H, is 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

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

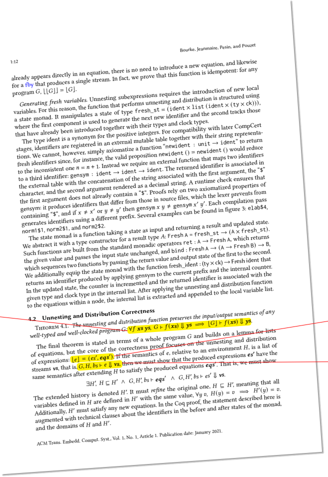
The extended history is denoted H'. It must *refine* the original one, $H \sqsubseteq H'$, meaning that all variables defined in H are defined in H' with the same value, $\forall y\ v,\ H(y) = v \implies H'(y) = v$. Additionally, H' must satisfy any new equations. In the Coq proof, the statement described 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'.

$$\forall f\ \mathbf{xs}\ \mathbf{ys},\quad G \vdash f(\mathbf{xs}) \Downarrow \mathbf{ys}$$
$$\implies \lfloor G\rfloor \vdash f(\mathbf{xs}) \Downarrow \mathbf{ys}$$

Unnesting & Distribution – correctness



$$\forall f\ \mathbf{xs}\ \mathbf{ys},\ \ G \vdash f(\mathbf{xs}) \Downarrow \mathbf{ys}$$
$$\implies\ \lfloor G \rfloor \vdash f(\mathbf{xs}) \Downarrow \mathbf{ys}$$



1:12

Bourke, Jeanmaire, Pesin, and Pouzet

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 representa-
tions. 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:* $\forall f\ \mathbf{xs}\ \mathbf{ys},\ G \vdash f(\mathbf{xs}) \Downarrow \mathbf{ys} \implies \lfloor G \rfloor \vdash f(\mathbf{xs}) \Downarrow \mathbf{ys}$.

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: $\lfloor e \rfloor = (es', \mathbf{eqs'})$. If the semantics of e, relative to an environment H, is a list of
streams $\mathbf{vs}$, that is, $G, H, bs \vdash e \Downarrow \mathbf{vs}$, then we must show that the produced expressions $\mathbf{es'}$ have the
same semantics after extending H to satisfy the produced equations $\mathbf{eqs'}$. That is, we must show

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

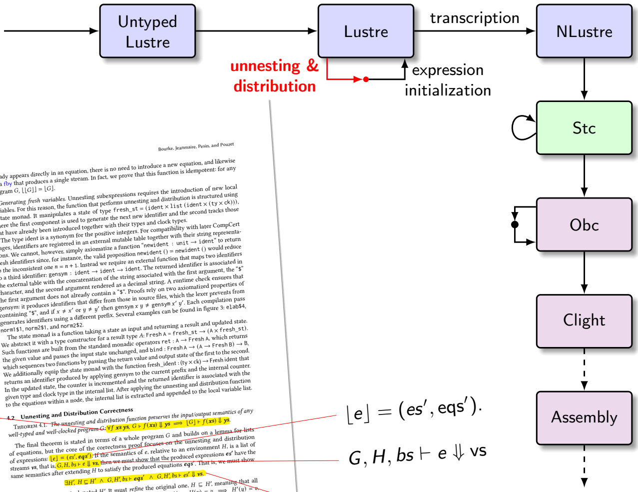
The extended history is denoted H'. It must *refine* the original one, $H \sqsubseteq H'$, meaning that all
variables defined in H are defined in H' with the same value, $\forall y\ v,\ H(y) = v \implies H'(y) = v$.
Additionally, H' must satisfy any new equations. In the Coq proof, the statement described 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'.

ACM Trans. Embedd. Comput. Syst., Vol. 1, No. 1, Article 1. Publication date: January 2021.

$$\lfloor e \rfloor = (es', \mathbf{eqs'}).$$

$$G, H, bs \vdash e \Downarrow \mathbf{vs}$$

Unnesting & Distribution – correctness



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'.

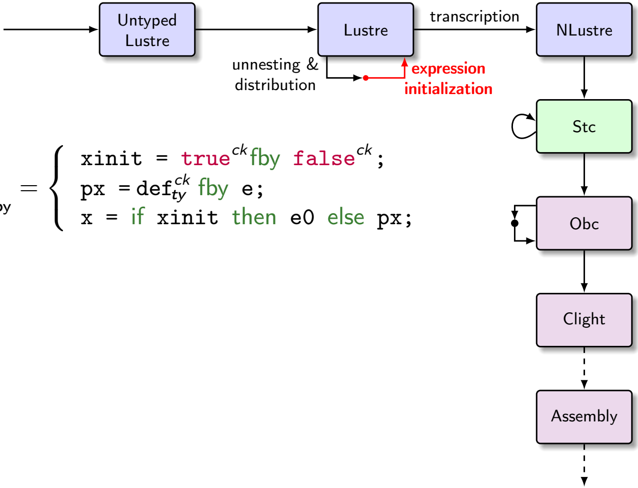
$$\forall f\ \mathbf{xs}\ \mathbf{ys},\ \ G \vdash f(\mathbf{xs}) \Downarrow \mathbf{ys}$$
$$\implies \lfloor G\rfloor \vdash f(\mathbf{xs}) \Downarrow \mathbf{ys}$$

$$\lfloor e\rfloor = (es', \mathrm{eqs}').$$

$$G, H, bs \vdash e \Downarrow \mathrm{vs}$$

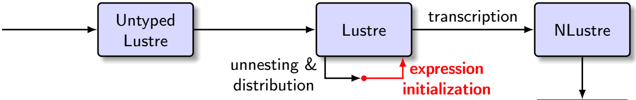
$$\exists H',\ H \sqsubseteq H' \ \wedge\ G, H', bs \vdash \mathrm{eqs}' \ \wedge\ G, H', bs \vdash es' \Downarrow \mathrm{vs}$$

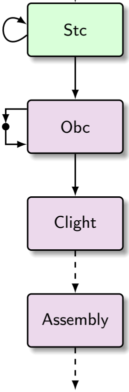
Expression Initialization



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

Expression Initialization





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

Optimization: avoid introducing several init registers

- Registers are costly in the final imperative program
- Use state monad to remember init registers introduced:
  `Fresh A (ann * bool)`
- Complicates the correctness proof with a non-local invariant

Clock system correctness

`x = 0 fby (x + 1)`	0	1	2	3	4	5	6	8	9	...
`b`	T	T	F	F	T	T	T	F	F	...
`x when b`	0	1			4	5	6			...

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

`x = 0 fby (x + 1)`	0	1	2	3	4	5	6	8	9	...
`b`	T	T	F	F	T	T	T	F	F	...
`x when b`	0	1			4	5	6			...

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.

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

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

Clock system correctness

`x = 0 fby (x + 1)`	0	1	2	3	4	5	6	8	9	...
`b`	T	T	F	F	T	T	T	F	F	...
`x when b`	0	1			4	5	6			...

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.

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

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

Clock system correctness

`x = 0 fby (x + 1)`	0	1	2	3	4	5	6	8	9	...
`b`	T	T	F	F	T	T	T	F	F	...
`x when b`	0	1			4	5	6			...

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.

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

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

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

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

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,* $H, bs \vdash y_1^{ck'_1} \Downarrow s'_1, \ldots, y_m^{ck'_m} \Downarrow s'_m$.

Clock system correctness

`x = 0 fby (x + 1)`	0	1	2	3	4	5	6	8	9	...
`b`	T	T	F	F	T	T	T	F	F	...
`x when b`	0	1			4	5	6			...

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.

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

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

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

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

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,* $H, bs \vdash y_1^{ck'_1} \Downarrow s'_1, \ldots, y_m^{ck'_m} \Downarrow s'_m$.

Clock system correctness

`x = 0 fby (x + 1)`	0	1	2	3	4	5	6	8	9	...
`b`	T	T	F	F	T	T	T	F	F	...
`x when b`	0	1			4	5	6			...

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.

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

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

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

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

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,* $H, bs \vdash y_1^{ck'_1} \Downarrow s'_1, \ldots, y_m^{ck'_m} \Downarrow s'_m$.

Clock system correctness

`x = 0 fby (x + 1)`	0	1	2	3	4	5	6	8	9	...
`b`	T	T	F	F	T	T	T	F	F	...
`x when b`	0	1			4	5	6			...

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.

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

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*

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

and semantics $f(s_1, \ldots, s_n) \Downarrow s'_1, \ldots, s'_m$, *with* $bs = \texttt{base-of}(s_1, \ldots, s_n)$, *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,* $H, bs \vdash y_1^{ck'_1} \Downarrow s'_1, \ldots, y_m^{ck'_m} \Downarrow s'_m$.

Clock system correctness – causality and proof

- 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)`

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)`

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)`

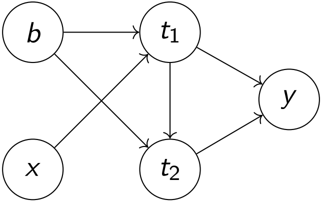
Clock system correctness – causality and proof

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

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 → 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
```

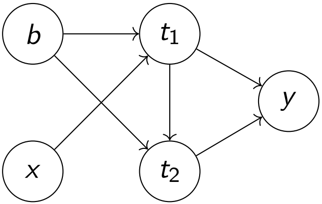


- graphs difficult to handle in a proof assistant: 1200 lines of Coq

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 → 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
```

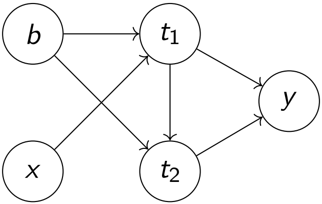


- graphs difficult to handle in a proof assistant: 1200 lines of Coq
- induction on a topological ordering of the nodes of the graph
- look only to the left of fby: the fby operator forces alignment

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 → 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
- intricate proof: around 2000 lines of Coq proof script

Node Subsampling

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

Node Subsampling

node current($\underbrace{\text{d : int; ck : bool}}_{\text{always present}}$; x : int when ck)

Node Subsampling

$\text{node current}(\underbrace{\text{d} : \texttt{int}; \text{ck} : \texttt{bool}}_{\text{always present}}; \underbrace{\text{x} : \texttt{int} \text{ when ck}}_{\text{only present when ck is}})$

Node Subsampling

$$\text{node current}(\underbrace{d : \texttt{int};\ ck : \texttt{bool}}_{\text{always present}};\ \underbrace{x : \texttt{int}\ \text{when}\ ck}_{\text{only present when ck is}})$$

Compile an instance of this node to Obc:

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

Node Subsampling

node current$(\underbrace{\text{d} : \texttt{int}; \text{ck} : \texttt{bool}}_{\text{always present}}; \underbrace{\text{x} : \texttt{int} \text{ when ck}}_{\text{only present when ck is}})$

Compile an instance of this node to Obc:

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

only defined when ck = true

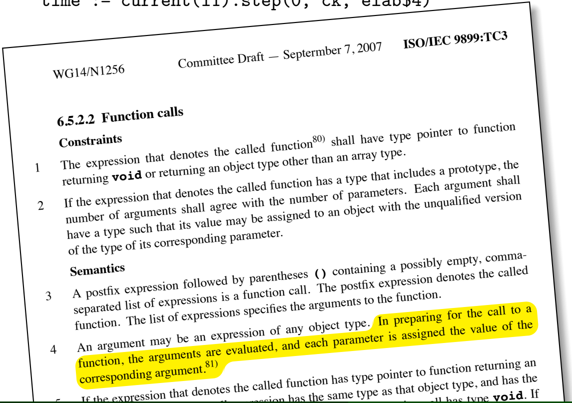
Node Subsampling

$$\text{node current}(\underbrace{d : \texttt{int}; ck : \texttt{bool}}_{\text{always present}}; \underbrace{x : \texttt{int} \text{ when } ck}_{\text{only present when ck is}})$$

Compile an instance of this node to Obc:

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

only defined when ck = true



WG14/N1256 Committee Draft — Septermber 7, 2007 **ISO/IEC 9899:TC3**

6.5.2.2 Function calls

Constraints

1 The expression that denotes the called function[80] shall have type pointer to function returning **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

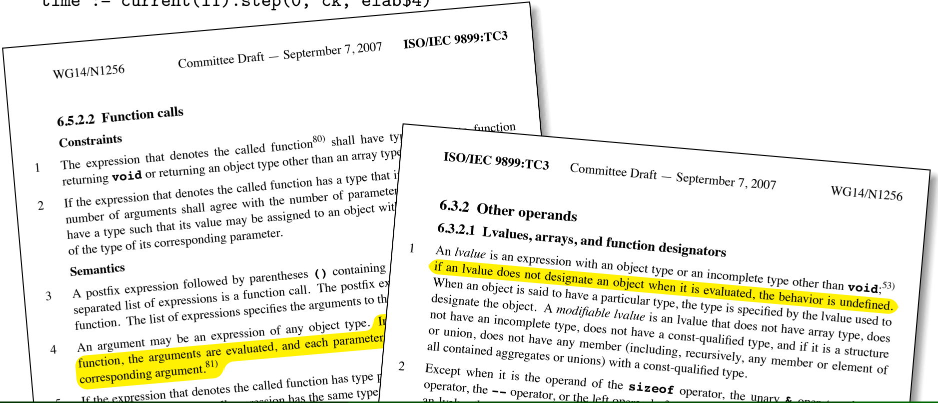
Node Subsampling

node current($\underbrace{\text{d : int; ck : bool;}}_{\text{always present}}$ $\underbrace{\text{x : int when ck}}_{\text{only present when ck is}}$)

Compile an instance of this node to Obc:

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

only defined when ck = true



WG14/N1256 Committee Draft — Septermber 7, 2007 ISO/IEC 9899:TC3

6.5.2.2 Function calls

Constraints

1 The expression that denotes the called function[80] shall have ty... returning **void** or returning an object type other than an array type...

2 If the expression that denotes the called function has a type that i... number of arguments shall agree with the number of parameter... have a type such that its value may be assigned to an object wit... of the type of its corresponding parameter.

Semantics

3 A postfix expression followed by parentheses **()** containing... separated list of expressions is a function call. The postfix ex... function. The list of expressions specifies the arguments to th...

4 An argument may be an expression of any object type. In... function, the arguments are evaluated, and each parameter... corresponding argument.[81]

ISO/IEC 9899:TC3 Committee Draft — September 7, 2007 WG14/N1256

6.3.2 Other operands

6.3.2.1 Lvalues, arrays, and function designators

1 An *lvalue* is an expression with an object type or an incomplete type other than **void**;[53] if an lvalue does not designate an object when it is evaluated, the behavior is undefined. When an object is said to have a particular type, the type is specified by the lvalue used to designate the object. A *modifiable lvalue* is an lvalue that does not have array type, does not have an incomplete type, does not have a const-qualified type, and if it is a structure or union, does not have any member (including, recursively, any member or element of all contained aggregates or unions) with a const-qualified type.

2 Except when it is the operand of the **sizeof** operator, the unary **&** oper... operator, the **--** operator, or the left opera...

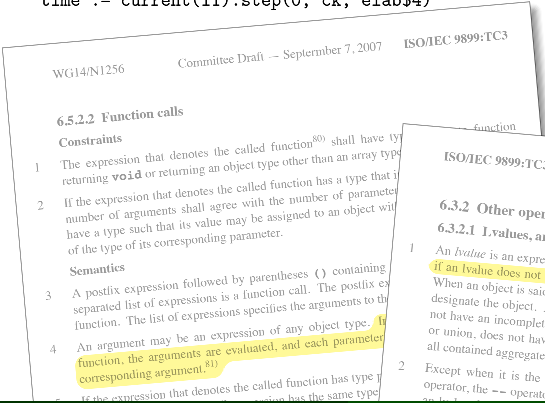
Node Subsampling

node current($\underbrace{\texttt{d : int; ck : bool}}_{\text{always present}}$; $\underbrace{\texttt{x : int when ck}}_{\text{only present when ck is}}$)

Compile an instance of this node to Obc:

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

only defined when ck = true


WG14/N1256 Committee Draft — Septermber 7, 2007 ISO/IEC 9899:TC3

6.5.2.2 Function calls

Constraints

1 The expression that denotes the called function[80] shall have ty returning **void** or returning an object type other than an array type

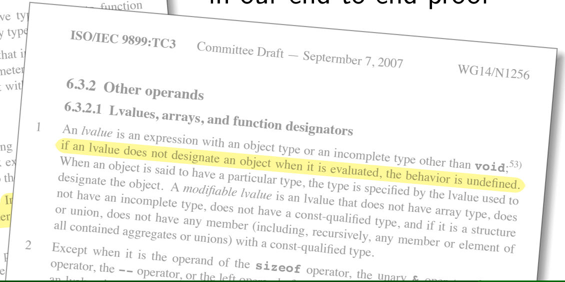
2 If the expression that denotes the called function has a type that i number of arguments shall agree with the number of parameter have a type such that its value may be assigned to an object wit of the type of its corresponding parameter.

Semantics

3 A postfix expression followed by parentheses **()** containing separated list of expressions is a function call. The postfix ex function. The list of expressions specifies the arguments to th

4 An argument may be an expression of any object type. In function, the arguments are evaluated, and each parameter corresponding argument.[81]

5 If the expression that denotes the called function has type p


ISO/IEC 9899:TC3 Committee Draft — Septermber 7, 2007 WG14/N1256

6.3.2 Other operands

6.3.2.1 Lvalues, arrays, and function designators

1 An *lvalue* is an expression with an object type or an incomplete type other than **void**;[53] if an lvalue does not designate an object when it is evaluated, the behavior is undefined. When an object is said to have a particular type, the type is specified by the lvalue used to designate the object. A *modifiable lvalue* is an lvalue that does not have array type, does not have an incomplete type, does not have a const-qualified type, and if it is a structure or union, does not have any member (including, recursively, any member or element of all contained aggregates or unions) with a const-qualified type.

2 Except when it is the operand of the **sizeof** operator, the unary **&** operator, the **--** operator, or the left operand

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

Node Subsampling

node current$(\underbrace{\text{d : int; ck : bool}}_{\text{always present}};\ \underbrace{\text{x : int when ck}}_{\text{only present when ck is}})$

Compile an instance of this node to Obc:

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

Node Subsampling

node current($\underbrace{\text{d : int; ck : bool}}_{\text{always present}}$; $\underbrace{\text{x : int when ck}}_{\text{only present when ck is}}$)

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)
```

Node Subsampling

$$\text{node current}(\underbrace{d : \mathtt{int};\ ck : \mathtt{bool}}_{\text{always present}};\ \underbrace{x : \mathtt{int}\ \text{when}\ ck}_{\text{only present when ck is}})$$

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

$\text{node current}(\underbrace{d : \texttt{int}; ck : \texttt{bool}}_{\text{always present}}; \underbrace{x : \texttt{int when } ck}_{\text{only present when ck is}})$

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:

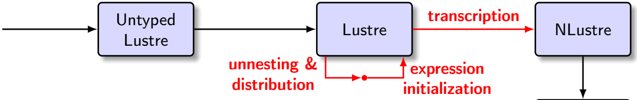
```
if (ck) {
  elab$4 := exp;
};
time := current(i1).step(0, ⟨ck⟩, elab$4)
```

2. Extra compilation pass to initialize variables:

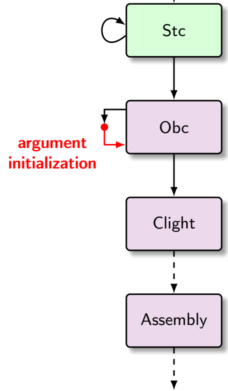
```
if (ck) {
  elab$4 := exp;
} else {
  elab$4 := 0;
};
time := current(i1).step(0, ⟨ck⟩, ⟨elab$4⟩)
```

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

Conclusion



res	F	F	F	T	F	F	F	F	F	···
n	6	6	6	6	6	6	6	6	6	···
cpt	6	5	4	6	5	4	3	2	1	···



- 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

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