Even more proofs about Lustre in Coq
(work in progress: existence of a semantics)

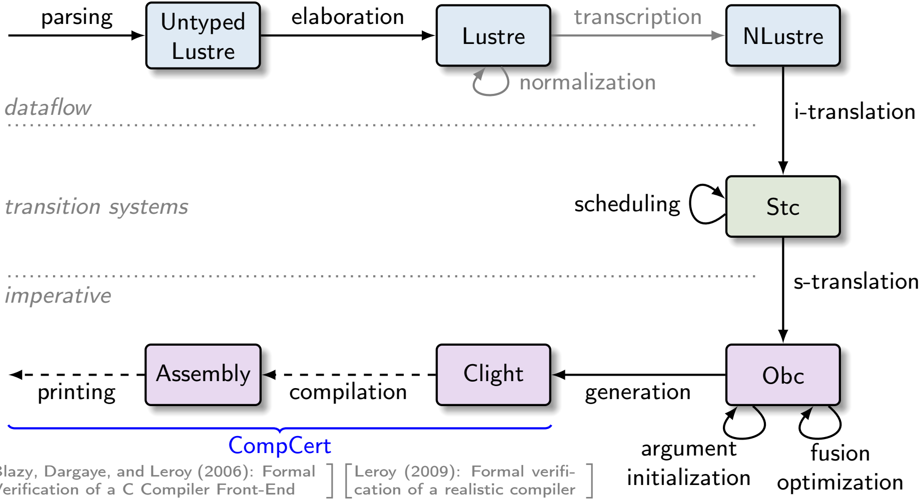
Timothy Bourke

Inria Paris — PARKAS Team
École normale supérieure, PSL University

27 November 2019, Synchron, Aussois

The Vélus Lustre Compiler

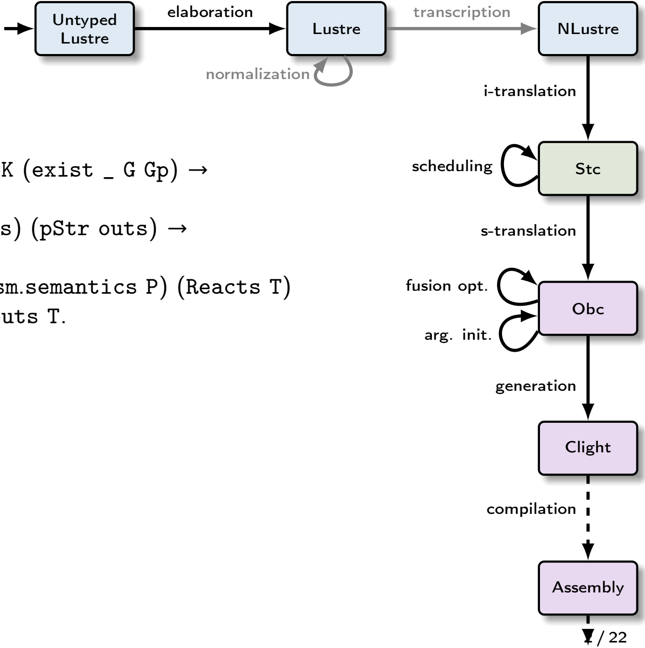
[Jourdan, Pottier, and Leroy (2012): Validating LR(1) parsers]



[Blazy, Dargaye, and Leroy (2006): Formal Verification of a C Compiler Front-End]

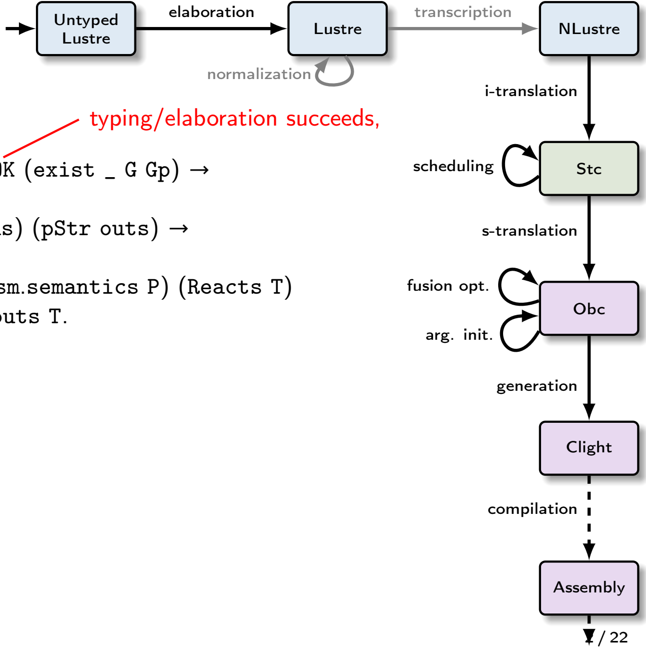
[Leroy (2009): Formal verification of a realistic compiler]

Main Theorem



```
Theorem correctness:
∀ D G Gp P main ins outs,
  elab_declarations D = OK (exist _ G Gp) →
  wt_ins G main ins →
  sem_node G main (pStr ins) (pStr outs) →
  compile D main = OK P →
  ∃ T, program_behaves (Asm.semantics P) (Reacts T)
   ∧ bisim_IO G main ins outs T.
```

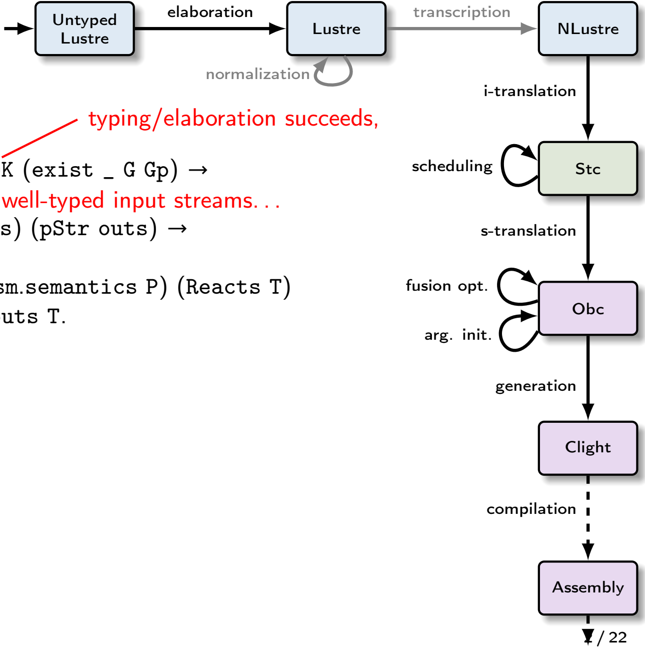
Main Theorem



```
Theorem correctness:
∀ D G Gp P main ins outs,
  elab_declarations D = OK (exist _ G Gp) →
  wt_ins G main ins →
  sem_node G main (pStr ins) (pStr outs) →
  compile D main = OK P →
  ∃ T, program_behaves (Asm.semantics P) (Reacts T)
   ∧ bisim_IO G main ins outs T.
```

typing/elaboration succeeds,

Main Theorem



Untyped Lustre

elaboration

Lustre

normalization

transcription

NLustre

i-translation

Stc

scheduling

s-translation

Obc

fusion opt.

arg. init.

generation

Clight

compilation

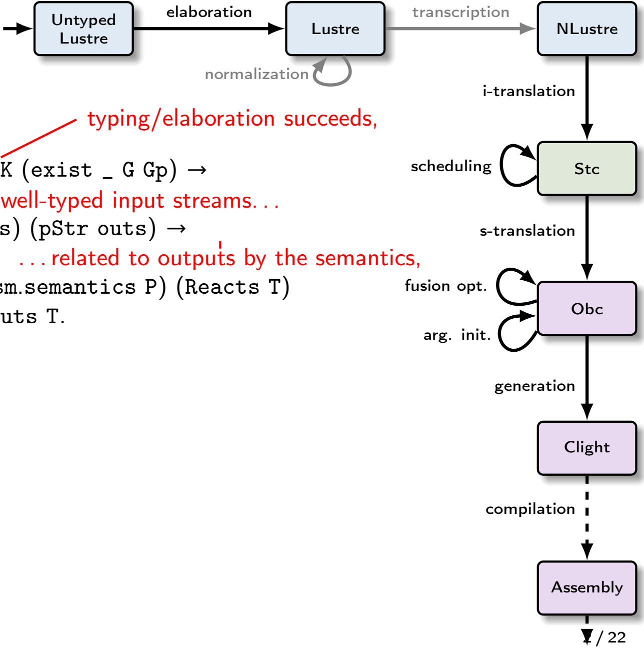
Assembly

```
Theorem correctness:
∀ D G Gp P main ins outs,
  elab_declarations D = OK (exist _ G Gp) →
  wt_ins G main ins →
  sem_node G main (pStr ins) (pStr outs) →
  compile D main = OK P →
  ∃ T, program_behaves (Asm.semantics P) (Reacts T)
   ∧ bisim_IO G main ins outs T.
```

typing/elaboration succeeds,

} ∀ well-typed input streams. . .

Main Theorem



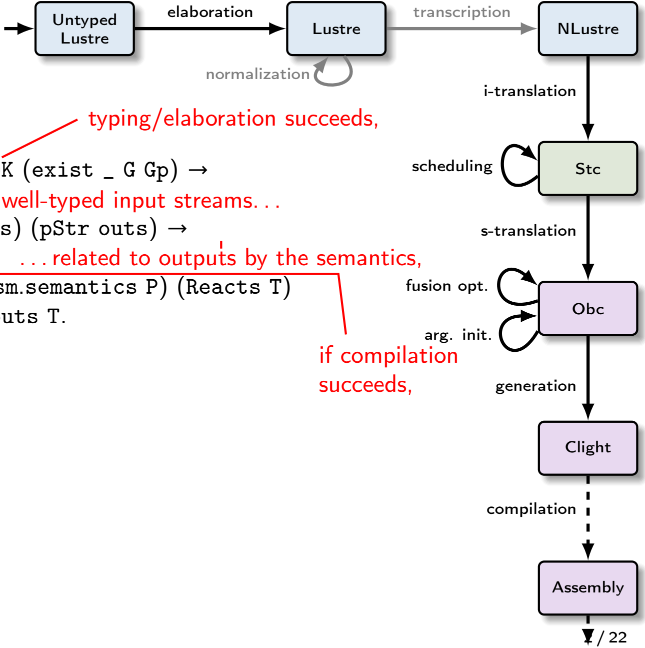
```
Theorem correctness:
∀ D G Gp P main ins outs,
  elab_declarations D = OK (exist _ G Gp) →
  wt_ins G main ins →
  sem_node G main (pStr ins) (pStr outs) →
  compile D main = OK P →
  ∃ T, program_behaves (Asm.semantics P) (Reacts T)
   ∧ bisim_IO G main ins outs T.
```

typing/elaboration succeeds,

} ∀ well-typed input streams. . .

. . . related to outputs by the semantics,

Main Theorem



```
Theorem correctness:
∀ D G Gp P main ins outs,
  elab_declarations D = OK (exist _ G Gp) →
  wt_ins G main ins →
  sem_node G main (pStr ins) (pStr outs) →
  compile D main = OK P →
  ∃ T, program_behaves (Asm.semantics P) (Reacts T)
   ∧ bisim_IO G main ins outs T.
```

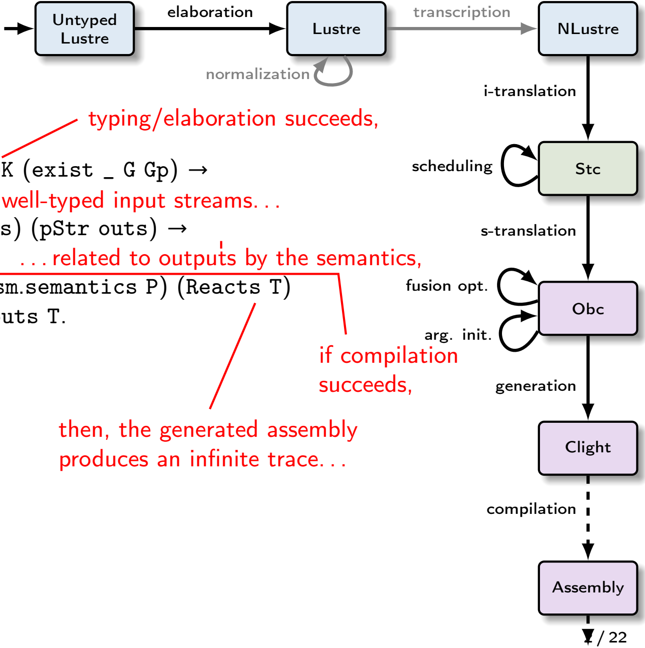
typing/elaboration succeeds,

} ∀ well-typed input streams. . .

. . . related to outputs by the semantics,

if compilation succeeds,

Main Theorem



```
Theorem correctness:
∀ D G Gp P main ins outs,
  elab_declarations D = OK (exist _ G Gp) →
  wt_ins G main ins →
  sem_node G main (pStr ins) (pStr outs) →
  compile D main = OK P →
  ∃ T, program_behaves (Asm.semantics P) (Reacts T)
   ∧ bisim_IO G main ins outs T.
```

typing/elaboration succeeds,

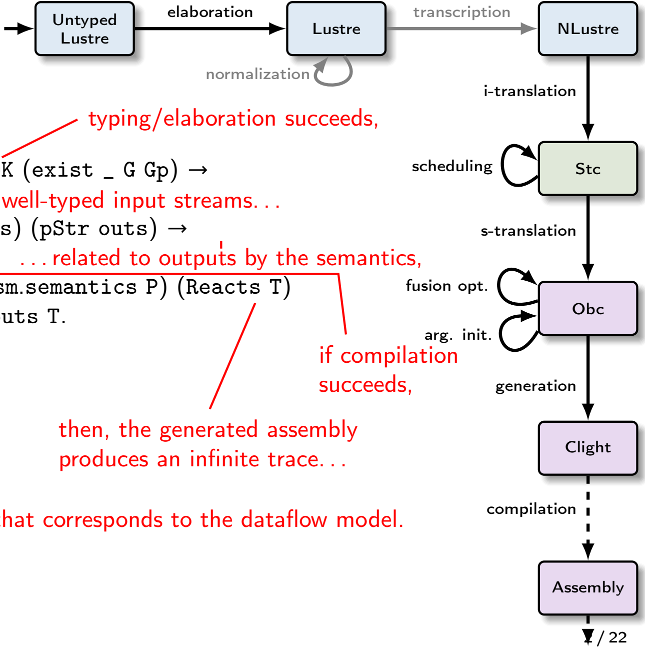
} ∀ well-typed input streams. . .

. . . related to outputs by the semantics,

if compilation succeeds,

then, the generated assembly produces an infinite trace. . .

Main Theorem



```
Theorem correctness:
∀ D G Gp P main ins outs,
  elab_declarations D = OK (exist _ G Gp) →
  wt_ins G main ins →
  sem_node G main (pStr ins) (pStr outs) →
  compile D main = OK P →
  ∃ T, program_behaves (Asm.semantics P) (Reacts T)
   ∧ bisim_IO G main ins outs T.
```

typing/elaboration succeeds,

} ∀ well-typed input streams. . .

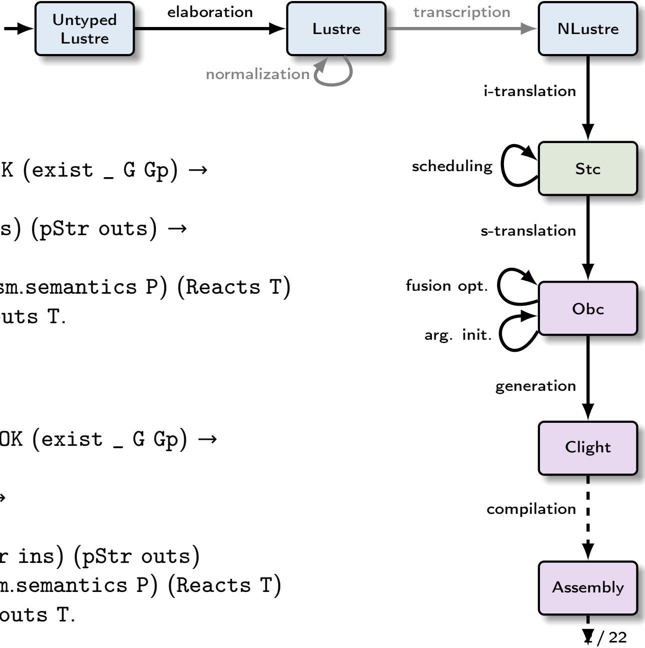
. . . related to outputs by the semantics,

if compilation succeeds,

then, the generated assembly produces an infinite trace. . .

. . . that corresponds to the dataflow model.

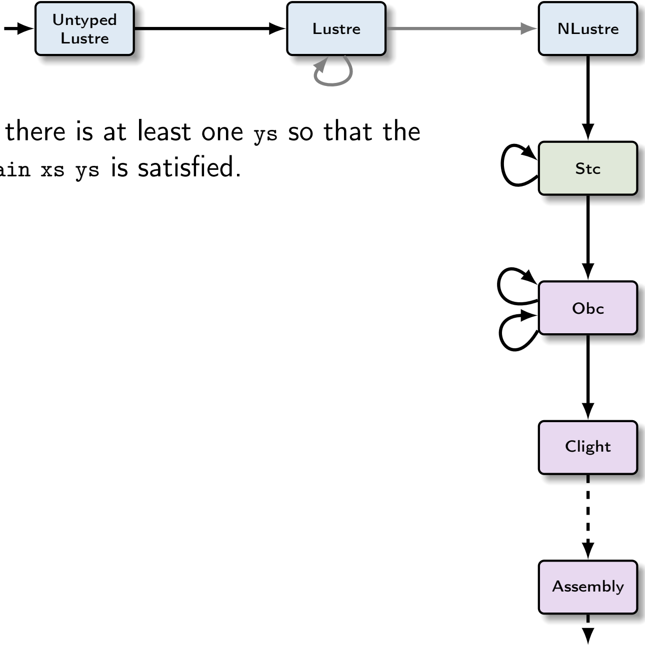
Main Theorem



```
Theorem correctness:
∀ D G Gp P main ins outs,
  elab_declarations D = OK (exist _ G Gp) →
  wt_ins G main ins →
  sem_node G main (pStr ins) (pStr outs) →
  compile D main = OK P →
  ∃ T, program_behaves (Asm.semantics P) (Reacts T)
   ∧ bisim_IO G main ins outs T.
```

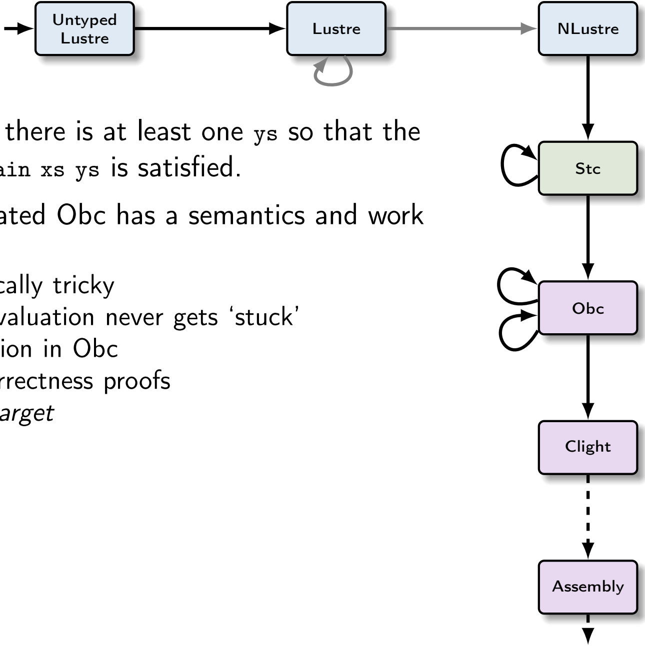
```
Theorem correctness:
∀ D G Gp P main ins,
  elab_declarations D = OK (exist _ G Gp) →
  wt_ins G main ins →
  compile D main = OK P →
  ∃ outs T,
    sem_node G main (pStr ins) (pStr outs)
  ∧ program_behaves (Asm.semantics P) (Reacts T)
  ∧ bisim_IO G main ins outs T.
```

Providing a Witness



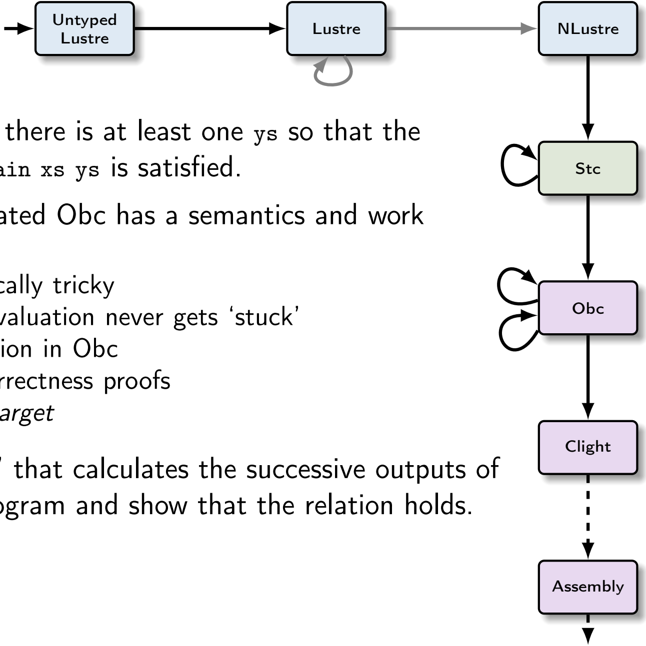
- For a given `xs`, show there is at least one `ys` so that the relation `sem_node G main xs ys` is satisfied.

Providing a Witness



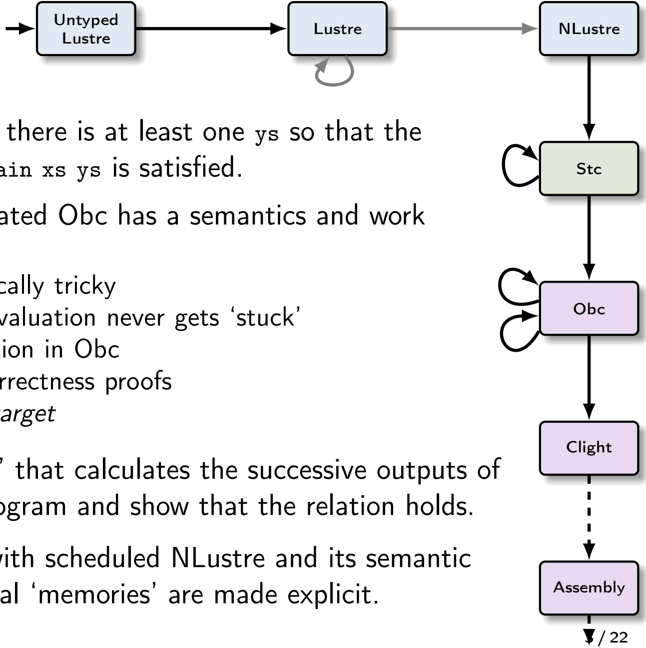
- For a given `xs`, show there is at least one `ys` so that the relation `sem_node G main xs ys` is satisfied.
- Show that the generated Obc has a semantics and work backward?
  - Tempting but technically tricky
  - Need to show that evaluation never gets ‘stuck’
  - No clocking information in Obc
  - Show ‘backwards’ correctness proofs
    generate(*source*) = *target*

Providing a Witness



- For a given `xs`, show there is at least one `ys` so that the relation `sem_node G main xs ys` is satisfied.
- Show that the generated Obc has a semantics and work backward?
  - Tempting but technically tricky
  - Need to show that evaluation never gets ‘stuck’
  - No clocking information in Obc
  - Show ‘backwards’ correctness proofs
    $generate(source) = target$
- Write an ‘interpreter’ that calculates the successive outputs of the source Lustre program and show that the relation holds.

Providing a Witness



- For a given `xs`, show there is at least one `ys` so that the relation `sem_node G main xs ys` is satisfied.
- Show that the generated Obc has a semantics and work backward?
  - Tempting but technically tricky
  - Need to show that evaluation never gets ‘stuck’
  - No clocking information in Obc
  - Show ‘backwards’ correctness proofs
    generate(*source*) = *target*
- Write an ‘interpreter’ that calculates the successive outputs of the source Lustre program and show that the relation holds.
- Start simple: work with scheduled NLustre and its semantic relation where internal ‘memories’ are made explicit.

N-Lustre syntax

Expressions

e	$::=$	x	variables
	$\mid$	k	constants
	$\mid$	$\diamond\ e$	unary operators
	$\mid$	$e \oplus e$	binary operators
	$\mid$	e when $(x = k)$	sampling
ce	$::=$	merge $x\ ce_t\ ce_f$	binary merge
	$\mid$	if e then ce_t else ce_f	multiplexors
	$\mid$	e	simple expressions

Equations

$eq ::= x =_{ck}\ ce$

$\mid\ x =_{ck}\ k_0$ fby e

$\mid\ x =_{ck}\ f(e, \dots, e)$

$\mid\ x =_{ck}\ ($restart f every $r)(e, \dots, e)$

Nodes

node $f\ (x : \tau)$ returns $(x : \tau)$

var $x : \tau, \dots, x : \tau$

let $eq; \cdots; eq$ tel

Clocks

$ck ::=$ base $\mid$ ck on $(x = k)$

N-Lustre node declarations (in Coq)

```
Record node : Type :=
  mk_node {
      name   : ident;
      inp    : list (ident ∗ (type ∗ clock));
      out    : list (ident ∗ (type ∗ clock));
      vars   : list (ident ∗ (type ∗ clock));
      eqs    : list equation;

      inpgt0  : 0 < length inp;
      outgt0  : 0 < length out;

      defd   : Permutation (vars_defined eqs)
                           (map fst (vars ++ out));

      vout   : ∀ out, In out (map fst out) →
                     ¬ In out (vars_defined (filter is_fby eqs));

      nodup  : NoDupMembers (inp ++ vars ++ out);

      good   : Forall ValidId (inp ++ vars ++ out) ∧ valid name
    }.
```

Memory semantics of NLustre

```
Inductive msem_equation: stream bool → history → memories → equation → Prop :=
with msem_node: ident → stream (list value) → memories → stream (list value) → Prop :=
```

sem_caexp b H ck e es
sem_var H x es
msem_equation b H M ($x =_{ck} e$)
⋮

sem_aexp b H ck e es
mfby x $[\![c_0]\!]$ es M xs
sem_var H x xs
msem_equation b H M ($x =_{ck} c_0$ fby e)

b = clock_of xss
find_node f G = Some n
sem_vars H (map fst n.inp) xss
sem_vars H (map fst n.out) yss
sem_clocked_vars b H (idck n.inp)
Forall (msem_equation b H M) n.eqs
memory_closed M n.eqs
msem_node f xss M yss

```
Definition mfby x $c_0$ (xs: stream value) (M: memories) (ys: stream value) : Prop :=
  find_val x (M 0) = Some $c_0$
  ∧ ∀ n, ∃ m, find_val x (M n) = m
            ∧ match xs n with
              | ‹›  ⇒ find_val x (M (n + 1)) = Some m ∧ ys n = ‹›
              | ‹v› ⇒ find_val x (M (n + 1)) = Some v ∧ ys n = ‹m›
              end
```

Interpreting expressions

```
Variables (base : bool) (R : env).

Fixpoint interp_exp_instant (e: exp) : option value :=
  match e with
  | c ⇒ Some (if base then ‹⟦c⟧› else ‹›)

  | x ⇒ interp_var_instant x

  | e when (x = b) ⇒
      match interp_var_instant x, interp_exp_instant e with
      | Some ‹xv›, Some ‹ev› ⇒ option_map (fun b'⇒ if b' ==b b then ‹ev› else ‹›) (val_to_bool xv)
      | Some ‹›,   Some ‹› ⇒ Some ‹›
      | _, _ ⇒ None
      end

  | ◇ e ⇒
      match interp_exp_instant e with
      | Some ‹v› ⇒ option_map present (sem_unop op v (typeof e))
      | vo ⇒ vo
      end

  | e1 ⊕ e2 ⇒
      match interp_exp_instant e1, interp_exp_instant e2 with
      | Some ‹v1›, Some ‹v2› ⇒ option_map present (sem_binop op v1 (typeof e1) v2 (typeof e2))
      | Some ‹›,   Some ‹› ⇒ Some ‹›
      | _, _ ⇒ None
      end
 end.
```

Interpreting expressions

```
Fixpoint interp_cexp_instant (e: cexp) : option value :=
  match e with
  | merge x t f ⇒
    match interp_var_instant x with

    | Some ‹xv› ⇒
      match val_to_bool xv, interp_cexp_instant t, interp_cexp_instant f with
      | Some true,  Some ‹tv›, Some ‹›   ⇒ Some ‹tv›
      | Some false, Some ‹›,   Some ‹fv› ⇒ Some ‹fv›
      | _, _, _ ⇒ None
      end

    | Some ‹› ⇒
      match interp_cexp_instant t, interp_cexp_instant f with
      | Some ‹›, Some ‹› ⇒ Some ‹›
      | _, _ ⇒ None
      end

    | None ⇒ None
    end

  | if b then t else f ⇒
    match interp_exp_instant b, interp_cexp_instant t, interp_cexp_instant f with

    | Some ‹bv›, Some ‹tv›, Some ‹fv› ⇒
      option_map (fun (b : bool) ⇒ if b then ‹tv› else ‹fv›) (val_to_bool bv)

    | Some ‹›,   Some ‹›,   Some ‹› ⇒ Some ‹›
    | _, _, _ ⇒ None
    end

  | e ⇒ interp_exp_instant e
  end.
```

Interpreting nodes

```
Fixpoint interp_node (G : global) (f : ident) (vs : list value) (base : bool)
                     (M : memory val) : option (env * (memory val * list value)) :=
match G with
| n :: G' ⇒
  let interp_node' f vs base M := map snd (interp_node G' f vs base M) in
  if n.name ==b f then
    do env ← updates_env (Env.empty _) (map fst n.inp) vs;

    do (env', M') ← ofold_right
                          (interp_equation interp_node' (init_memory G') base M)
                          (Some (env, empty_memory _)) n.eqs;

    do _ ← if forallb (check_clock base env') n.inp then Some tt else None;

    do M'' ← ofold_right (next_from_equation base M env') (Some M') n.eqs;

    do rvs ← omap (fun x ⇒ Env.find x env') (map fst n.out);
    Some (env', (M'', rvs))

  else interp_node G' f vs base M
| []     ⇒ None
end.
```

Interpreting nodes

```
Definition interp_equation
    (interp_node : ident → list value → bool → memory val →
                   option (memory val ∗ list value))
    (init_node : ident → option (memory val))
    (base : bool)
    (M : memory val)
    (eq : equation) ((R, M') : env ∗ memory val) : option (env ∗ memory val) :=

  match eq with
  | x =ck ce ⇒
    do v ← interp_caexp_instant base R ck ce;
    Some (Env.add x v R, M')

  | x =ck c0 fby e ⇒
    do v ← find_val x M;
    do c ← interp_clock_instant base R ck;
    Some (Env.add x (if c then ‹v› else ‹›) R, M')

  | xs =ck (restart f every r)(es) ⇒
    ...
  end.
```

Interpreting nodes

```
Definition next_from_equation (base : bool) (M : memory val) (R : env)
    (eq : equation) (M' : memory val) : option (memory val) :=

  match eq with
  | x =ck ce ⇒ Some M'

  | x =ck c0 fby e ⇒
      do v ← interp_aexp_instant base R ck e;
      match v with
      | ‹c› ⇒ Some (add_val x c M')
      | ‹›  ⇒ map (fun c ⇒ add_val x c M') (find_val x M)
      end

  | xs =ck (restart f every r)(es) ⇒ Some M'
  end.
```

Interpreter

- Ugly; mechanical; ad hoc
- Calculates a single cycle and the state for the next cycle.

Interpreter

- Ugly; mechanical; ad hoc
- Calculates a single cycle and the state for the next cycle.
- Causality of `fbys`: calculate in two passes.
  - More liberal than the compilation scheme.

```
x = false fby (not y)
y = false fby x
```

Interpreter

- Ugly; mechanical; ad hoc
- Calculates a single cycle and the state for the next cycle.
- Causality of `fbys`: calculate in two passes.
  - More liberal than the compilation scheme.

```
x = false fby (not y)
y = false fby x
```

- Causality of node instances: calculate all inputs before interpreting recursively to produce all outputs (and next state).
  - Matches compilation scheme and 'well scheduled' predicate
  - Otherwise more restrictive than necessary

```
node f(x, y) returns (u, v);
let
  u = x;
  v = y;
tel
...
s, t = f(3, s)
```

One Instant Memory semantics of NLustre

```
Inductive MI.msem_equation: bool → bool → env → memory val → memory val → equation → Prop :=
with MI.msem_node: ident → list value → memory val → memory val → list value → Prop :=
```

find_val x M = Some m
sem_clock_inst b R ck bi
sem_var_inst R x (if bi then ‹m› else ‹›)
msem_equation *F* b R M (x $=_{ck}$ c_0 fby e)
⋮

sem_aexp_inst b R ck e ev
sem_var_inst R x xv
mfby x ev M M' xv
msem_equation *T* b R M M' (x $=_{ck}$ c_0 fby e)

k = clock_of_inst xvs
find_node f G = Some n
sem_vars_inst R (map fst n.inp) xvs
sem_vars_inst R (map fst n.out) yvs
sem_clocked_vars_inst b R (idck n.inp)
Forall (msem_equation *T* b R M M') n.eqs
memory_closed M n.eqs
memory_closed M' n.eqs
msem_node f xss M M' yvs

```
Definition mfby (x: ident) (xs: value) (M M': memory val) (ys: value) : Prop :=
  ∃ m, find_val x (M n) = m
     ∧ match xs with
       | ‹›   ⇒ find_val x M' = Some m ∧ ys = ‹›
       | ‹v›  ⇒ find_val x M' = Some v ∧ ys = ‹m›
       end
```

Reasoning modulo options and memories

Relating optional values (with P. Jeanmaire)

```
Context {A : Type} (R : relation A).

Inductive orel : relation (option A) :=
| Oreln : orel None None
| Orels : ∀ sx sy, R sx sy → orel (Some sx) (Some sy).
```

Reasoning modulo options and memories

Relating optional values (with P. Jeanmaire)

```
Context {A : Type} (R : relation A).

Inductive orel : relation (option A) :=
| Oreln : orel None None
| Orels : ∀ sx sy, R sx sy → orel (Some sx) (Some sy).
```

Memories

```
Inductive memory (V: Type) := { values: Env.t V;
                                 instances: Env.t (memory V) }.

Inductive equal_memory {V: Type} : memory V → memory V → Prop :=
  equal_memory_intro: ∀ m m',
    Env.Equiv eq (values m) (values m') →
    Env.Equiv equal_memory (instances m) (instances m') →
    equal_memory m m'.

Lemma Equiv_orel {R : relation A} :
  ∀ S T, Env.Equiv R S T ↔ (∀ x, (orel R) (Env.find x S) (Env.find x T)).
```

Proof Sketch

1. Assume semantic relation, show the interpreter calculates the same result

Proof Sketch

1. Assume semantic relation, show the interpreter calculates the same result

```
Ordered_nodes G →
Forall Causal G →
wt_global G →
wc_global G →
MI.msem_node G f xs M M' ys →
orel (equal_memory * eq)
     (option_map snd (interp_node G f xs (clock_of_inst xs) M))
     (Some (M', ys))
```

Proof Sketch

1. Assume semantic relation, show the interpreter calculates the same result

```
Ordered_nodes G →
Forall Causal G →
wt_global G →
wc_global G →
MI.msem_node G f xs M M' ys →
orel (equal_memory * eq)
     (option_map snd (interp_node G f xs (clock_of_inst xs) M))
     (Some (M',  ys))
```

2. Assume interpreter success, show the result satisfies the semantic relation

Proof Sketch

1. Assume semantic relation, show the interpreter calculates the same result

```
Ordered_nodes G →
Forall Causal G →
wt_global G →
wc_global G →
MI.msem_node G f xs M M' ys →
orel (equal_memory * eq)
     (option_map snd (interp_node G f xs (clock_of_inst xs) M))
     (Some (M', ys))
```

2. Assume interpreter success, show the result satisfies the semantic relation

```
Ordered_nodes G →
wc_global G →
memory_closed_rec G f M →
option_map snd (interp_node G f xs (clock_of_inst xs) M) = Some (M', ys) →
MI.msem_node G f xs M M' ys
```

Proof Sketch

1. Assume semantic relation, show the interpreter calculates the same result

```
Ordered_nodes G →
Forall Causal G →
wt_global G →
wc_global G →
MI.msem_node G f xs M M' ys →
orel (equal_memory * eq)
     (option_map snd (interp_node G f xs (clock_of_inst xs) M))
     (Some (M',  ys))
```

2. Assume interpreter success, show the result satisfies the semantic relation

```
Ordered_nodes G →
wc_global G →
memory_closed_rec G f M →
option_map snd (interp_node G f xs (clock_of_inst xs) M) = Some (M', ys) →
MI.msem_node G f xs M M' ys
```

3. Assume well-typed, well-clocked, causal, . . . , show interpreter success

```
...  → interp_node G f xs (clock_of_inst xs) M ≠ None
```

Intermediate results on equations

```
Fixpoint interp_node (G : global) (f : ident) (vs : list value) (base : bool) (M : memory val)
  : option (env * (memory val * list value)) :=

  match G with
  | n :: G' ⇒
    let interp_node' f vs base M := option_map snd (interp_node G' f vs base M) in
    if n.name ==b f then

      do env ← updates_env (Env.empty _) (map fst n.inp) vs;

      do (env', M') ← ofold_right (interp_equation interp_node' (init_memory G') base M)
                                                (Some (env, empty_memory _)) n.eqs;

      do _ ← if forallb (check_clock base env') n.inp then Some tt else None;

      do M'' ← ofold_right (next_from_equation base M env') (Some M') n.eqs;

      do rvs ← omap (fun x ⇒ Env.find x env') (map fst n.out);

      Some (env', (M'', rvs))

    else interp_node G' f vs base M

  | []      ⇒ None
  end.
```

Intermediate results on equations

```
Fixpoint interp_node (G : global) (f : ident) (vs : list value) (base : bool) (M : memory val)
  : option (env * (memory val * list value)) :=

  match G with
  | n :: G' ⇒
    let interp_node' f vs base M := option_map snd (interp_node G' f vs base M) in
    if n.name ==b f then

      do env ← updates_env (Env.empty _) (map fst n.inp) vs;

      do (env', M') ← ofold_right (interp_equation interp_node' (init_memory G') base M)
                                                   (Some (env, empty_memory _)) n.eqs;

      do _ ← if forallb (check_clock base env') n.inp then Some tt else None;

      do M'' ← ofold_right (next_from_equation base M env') (Some M') n.eqs;

      do rvs ← omap (fun x ⇒ Env.find x env') (map fst n.out);

      Some (env', (M'', rvs))

    else interp_node G' f vs base M

  | []     ⇒ None
  end.

...
(∀ f xs M M' rvs, memory_closed_rec G f M →
     orel (equal_memory * eq) (f_node f xs (clock_of_inst xs) M) (Some (M', rvs)) →
    MI.msem_node G f xs M M' rvs) →
ofold_right (interp_equation f_node (init_memory G) b M) (Some (H, N)) eqs = Some (H', M') →
Forall (MI.msem_equation G false b H' M M') eqs
∧ ...
```

Intermediate results on equations

```
Fixpoint interp_node (G : global) (f : ident) (vs : list value) (base : bool) (M : memory val)
  : option (env * (memory val * list value)) :=

  match G with
  | n :: G' ⇒
    let interp_node' f vs base M := option_map snd (interp_node G' f vs base M) in
    if n.name ==b f then

      do env ← updates_env (Env.empty _) (map fst n.inp) vs;

      do (env', M') ← ofold_right (interp_equation interp_node' (init_memory G') base M)
                                  (Some (env, empty_memory _)) n.eqs;

      do _ ← if forallb (check_clock base env') n.inp then Some tt else None;

      do M'' ← ofold_right (next_from_equation base M env') (Some M') n.eqs;

      do rvs ← omap (fun x ⇒ Env.find x env') (map fst n.out);

      Some (env', (M'', rvs))

    else interp_node G' f vs base M

  | []     ⇒ None
  end.

...
Forall (MI.msem_equation G false b H M N) eqs →
ofold_right (next_from_equation b M H) (Some N) eqs = Some M' →
Forall (MI.msem_equation G true b H M M') eqs
∧ ...
```

Lifting to streams

1. Lifting equations

```
...
(∀ f M xss yss,
    MI.init_node G f (M 0) →
    (∀ i, MI.msem_node G f (xss i) (M i) (M (i + 1)) (yss i)) →
    msem_node G f xss M yss) →

MI.init_equation G (M 0) eq →
(∀ i, MI.msem_equation G true (bck i) (H i) (M i) (M (i + 1)) eq) →
msem_equation G bck H M eq
```

Lifting to streams

1. Lifting equations
2. Lifting nodes

```
Fixpoint interp_state' (i : nat) : option (memory val) :=
  match i with
  | 0 ⇒ init_memory G f
  | S j ⇒ do M ← interp_state' j;
          do (M',  ys) ← option_map snd (interp_node G f (xss j)
                                           (clock_of_inst (xss j)) M);
          Some M'
  end.

Definition interp' (i : nat) : option (list value) :=
  do M ← interp_state' i;
  do (M',  ys) ← option_map snd
                   (interp_node G f (xss i) (clock_of_inst (xss i)) M);
  Some ys.

Definition interp (i : nat) : list value :=
  odef [‹›] (interp' i).
```

Lifting to streams

1. Lifting equations
2. Lifting nodes

```
...
MI.init_node G f (M 0) →
(∀ i, MI.msem_node G f (xss i) (M i) (M (i + 1)) (yss i)) →
msem_node G f xss M yss
```

Lifting to streams

1. Lifting equations
2. Lifting nodes

```
...
MI.init_node G f (M 0) →
(∀ i, MI.msem_node G f (xss i) (M i) (M (i + 1)) (yss i)) →
msem_node G f xss M yss
```

```
...
(∀ i, interp' G f xss i ≠ None) →
msem_node G f xss (interp_state G f xss) (interp G f xss).
```

Proof Sketch

1. Assume semantic relation, show the interpreter calculates the same result
2. Assume interpreter success, show the result satisfies the semantic relation
3. Assume well-typed, well-clocked, causal, . . . , show interpreter success

```
Ordered_nodes G →
wt_global G →
wc_global G →
Forall Causal G →
operator applications are defined →
interp_node G f xs (clock_of_inst xs) M ≠ None
```

- Introduce an abstract interface for values, types, and operators.
  - Define N-Lustre and Obc syntax and semantics against this interface.
  - Likewise for the N-Lustre to Obc translation and proof.
- Instantiate with definitions for the Obc to Clight translation and proof.

```
Module Type OPERATORS.

 Parameter val   : Type.
 Parameter type  : Type.
 Parameter const : Type.
```

- Introduce an abstract interface for values, types, and operators.
  - Define N-Lustre and Obc syntax and semantics against this interface.
  - Likewise for the N-Lustre to Obc translation and proof.
- Instantiate with definitions for the Obc to Clight translation and proof.

```
Module Type OPERATORS.

 Parameter val   : Type.
 Parameter type  : Type.
 Parameter const : Type.

 (* Boolean values *)
 Parameter bool_type : type.

 Parameter true_val  : val.
 Parameter false_val : val.
```

- Introduce an abstract interface for values, types, and operators.
  - Define N-Lustre and Obc syntax and semantics against this interface.
  - Likewise for the N-Lustre to Obc translation and proof.
- Instantiate with definitions for the Obc to Clight translation and proof.

```
Module Type OPERATORS.

 Parameter val   : Type.
 Parameter type  : Type.
 Parameter const : Type.

 (* Boolean values *)
 Parameter bool_type : type.

 Parameter true_val  : val.
 Parameter false_val : val.

 (* Constants *)
 Parameter type_const : const → type.
 Parameter sem_const  : const → val.
```

- Introduce an abstract interface for values, types, and operators.
    - Define N-Lustre and Obc syntax and semantics against this interface.
    - Likewise for the N-Lustre to Obc translation and proof.
- Instantiate with definitions for the Obc to Clight translation and proof.

```
Module Type OPERATORS.

 Parameter val   : Type.
 Parameter type  : Type.
 Parameter const : Type.

 (* Boolean values *)
 Parameter bool_type : type.

 Parameter true_val  : val.
 Parameter false_val : val.

 (* Constants *)
 Parameter type_const : const → type.
 Parameter sem_const  : const → val.


 (* Operators *)
 Parameter unop  : Type.
 Parameter binop : Type.

 Parameter sem_unop  :
   unop → val → type → option val.

 Parameter sem_binop :
   binop → val → type → val → type
      → option val.

 Parameter type_unop  :
   unop → type → option type.

 Parameter type_binop :
   binop → type → type → option type.

 (* ... *)

End OPERATORS.
```

- Introduce an abstract interface for values, types, and operators.
  - Define N-Lustre and Obc syntax and semantics against this interface.
  - Likewise for the N-Lustre to Obc translation and proof.

- Instantiate with definitions for the Obc to Clight translation and proof.

```
Module Type OPERATORS.

 Parameter val   : Type.
 Parameter type  : Type.
 Parameter const : Type.

 (* Boolean values *)
 Parameter bool_type : type.

 Parameter true_val  : val.
 Parameter false_val : val.

 (* Constants *)
 Parameter type_const : const → type.
 Parameter sem_const  : const → val.


 (* Operators *)
 Parameter unop  : Type.
 Parameter binop : Type.

 Parameter sem_unop  :
   unop → val → type → option val.

 Parameter sem_binop :
   binop → val → type → val → type
      → option val.

 Parameter type_unop  :
   unop → type → option type.

 Parameter type_binop :
   binop → type → type → option type.

 (* ... *)

End OPERATORS.
```

```
Module Export Op <: OPERATORS.

 Definition val: Type := Values.val.
```

```
Inductive val: Type :=
  | Vundef  : val
  | Vint    : int → val
  | Vlong   : int64 → val
  | Vfloat  : float → val
  | Vsingle : float32 → val
  | Vptr    : block → int → val.
```

```
Module Type OPERATORS.

 Parameter val   : Type.
 Parameter type  : Type.
 Parameter const : Type.

 (* Boolean values *)
 Parameter bool_type : type.

 Parameter true_val  : val.
 Parameter false_val : val.

 (* Constants *)
 Parameter type_const : const → type.
 Parameter sem_const  : const → val.


 (* Operators *)
 Parameter unop  : Type.
 Parameter binop : Type.

 Parameter sem_unop  :
   unop → val → type → option val.

 Parameter sem_binop :
   binop → val → type → val → type
      → option val.

 Parameter type_unop  :
   unop → type → option type.

 Parameter type_binop :
   binop → type → type → option type.

 (* ... *)

End OPERATORS.
```

```
Module Export Op <: OPERATORS.

 Definition val: Type := Values.val.

 Inductive type : Type :=
 | Tint   : intsize → signedness → type
 | Tlong  : signedness → type
 | Tfloat : floatsize → type.
```

```
Inductive signedness : Type :=
  | Signed   : signedness
  | Unsigned : signedness.

Inductive intsize : Type :=
  | I8    : intsize    (* char  *)
  | I16   : intsize    (* short *)
  | I32   : intsize    (* int   *)
  | IBool : intsize.   (* bool  *)

Inductive floatsize : Type :=
  | F32 : floatsize    (* float  *)
  | F64 : floatsize.   (* double *)
```

```
Module Type OPERATORS.

 Parameter val   : Type.
 Parameter type  : Type.
 Parameter const : Type.

 (* Boolean values *)
 Parameter bool_type : type.

 Parameter true_val  : val.
 Parameter false_val : val.

 (* Constants *)
 Parameter type_const : const → type.
 Parameter sem_const  : const → val.


 (* Operators *)
 Parameter unop  : Type.
 Parameter binop : Type.

 Parameter sem_unop  :
   unop → val → type → option val.

 Parameter sem_binop :
   binop → val → type → val → type
      → option val.

 Parameter type_unop  :
   unop → type → option type.

 Parameter type_binop :
   binop → type → type → option type.

 (* ... *)

End OPERATORS.
```

```
Module Export Op <: OPERATORS.

 Definition val: Type := Values.val.

 Inductive type : Type :=
 | Tint   : intsize → signedness → type
 | Tlong  : signedness → type
 | Tfloat : floatsize → type.

 Inductive const : Type :=
 | Cint    : int → intsize → signedness → const
 | Clong   : int64 → signedness → const
 | Cfloat  : float → const
 | Csingle : float32 → const.
```

```
Module Type OPERATORS.

 Parameter val   : Type.
 Parameter type  : Type.
 Parameter const : Type.

 (* Boolean values *)
 Parameter bool_type : type.

 Parameter true_val  : val.
 Parameter false_val : val.

 (* Constants *)
 Parameter type_const : const → type.
 Parameter sem_const  : const → val.


 (* Operators *)
 Parameter unop  : Type.
 Parameter binop : Type.

 Parameter sem_unop  :
   unop → val → type → option val.

 Parameter sem_binop :
   binop → val → type → val → type
      → option val.

 Parameter type_unop  :
   unop → type → option type.

 Parameter type_binop :
   binop → type → type → option type.

 (* ... *)

End OPERATORS.
```

```
Module Export Op <: OPERATORS.

 Definition val: Type := Values.val.

 Inductive type : Type :=
 | Tint   : intsize → signedness → type
 | Tlong  : signedness → type
 | Tfloat : floatsize → type.

 Inductive const : Type :=
 | Cint    : int → intsize → signedness → const
 | Clong   : int64 → signedness → const
 | Cfloat  : float → const
 | Csingle : float32 → const.

 Definition true_val  := Vtrue.  (* Vint Int.one  *)
 Definition false_val := Vfalse. (* Vint Int.zero *)

 Definition bool_type : type := Tint IBool Signed.
```

```
Module Type OPERATORS.

 Parameter val   : Type.
 Parameter type  : Type.
 Parameter const : Type.

 (* Boolean values *)
 Parameter bool_type : type.

 Parameter true_val  : val.
 Parameter false_val : val.

 (* Constants *)
 Parameter type_const : const → type.
 Parameter sem_const  : const → val.


 (* Operators *)
 Parameter unop  : Type.
 Parameter binop : Type.

 Parameter sem_unop  :
   unop → val → type → option val.

 Parameter sem_binop :
   binop → val → type → val → type
      → option val.

 Parameter type_unop  :
   unop → type → option type.

 Parameter type_binop :
   binop → type → type → option type.

 (* ... *)

End OPERATORS.
```

```
Module Export Op <: OPERATORS.

 Definition val: Type := Values.val.

 Inductive type : Type :=
 | Tint   : intsize → signedness → type
 | Tlong  : signedness → type
 | Tfloat : floatsize → type.

 Inductive const : Type :=
 | Cint    : int → intsize → signedness → const
 | Clong   : int64 → signedness → const
 | Cfloat  : float → const
 | Csingle : float32 → const.

 Definition true_val  := Vtrue.  (* Vint Int.one  *)
 Definition false_val := Vfalse. (* Vint Int.zero *)

 Definition bool_type : type := Tint IBool Signed.

 Inductive unop : Type :=
 | UnaryOp: Cop.unary_operation → unop
 | CastOp:  type → unop.

 Definition sem_unop (uop: unop) (v: val) (ty: type) : option val
 := match uop with
    | UnaryOp op ⇒ sem_unary_operation op v (cltype ty) Mem.empty
    | CastOp ty' ⇒ sem_cast v (cltype ty) (cltype ty') Mem.empty
    end.

 Definition binop := Cop.binary_operation.

 Definition sem_binop (op: binop) (v1: val) (ty1: type)
                            (v2: val) (ty2: type) : option val :=
   Cop.sem_binary_operation empty_composite_env
     op v1 (cltype ty1) v2 (cltype ty2) Memory.Mem.empty.
 (* ... *)
End Op.
```

```
Definition sem_binary_operation
    (cenv: composite_env)
    (op: binary_operation)
    (v1: val) (t1: type) (v2: val) (t2:type)
    (m: mem): option val :=
  match op with
  | Oadd ⇒ sem_add cenv v1 t1 v2 t2 m
  | Osub ⇒ sem_sub cenv v1 t1 v2 t2 m
  | Omul ⇒ sem_mul v1 t1 v2 t2 m
  | Omod ⇒ sem_mod v1 t1 v2 t2 m
  | Odiv ⇒ sem_div v1 t1 v2 t2 m
  | Oand ⇒ sem_and v1 t1 v2 t2 m
  | Oor  ⇒ sem_or v1 t1 v2 t2 m
  | Oxor  ⇒ sem_xor v1 t1 v2 t2 m
  | Oshl ⇒ sem_shl v1 t1 v2 t2
  | Oshr  ⇒ sem_shr v1 t1 v2 t2
  | Oeq ⇒ sem_cmp Ceq v1 t1 v2 t2 m
  | One ⇒ sem_cmp Cne v1 t1 v2 t2 m
  | Olt ⇒ sem_cmp Clt v1 t1 v2 t2 m
  | Ogt ⇒ sem_cmp Cgt v1 t1 v2 t2 m
  | Ole ⇒ sem_cmp Cle v1 t1 v2 t2 m
  | Oge ⇒ sem_cmp Cge v1 t1 v2 t2 m
  end.
```

```
Definition sem_mul (v1:val) (t1:type) (v2: val) (t2:type) (m:mem) : option val :=
  sem_binarith

    (fun sg n1 n2 ⇒ Some (Vint    (Int.mul     n1 n2)))

    (fun sg n1 n2 ⇒ Some (Vlong   (Int64.mul   n1 n2)))

    (fun n1 n2 ⇒    Some (Vfloat  (Float.mul   n1 n2)))

    (fun n1 n2 ⇒    Some (Vsingle (Float32.mul n1 n2)))

    v1 t1 v2 t2 m.
```

```
Definition sem_binarith
    (sem_int: signedness → int → int → option val)
    (sem_long: signedness → int64 → int64 → option val)
    (sem_float: float → float → option val)
    (sem_single: float32 → float32 → option val)
    (v1: val) (t1: type) (v2: val) (t2: type) (m: mem): option val :=

  let c := classify_binarith t1 t2 in
  let t := binarith_type c in
  match sem_cast v1 t1 t m with
  | None ⇒ None
  | Some v1' ⇒

  match sem_cast v2 t2 t m with
  | None ⇒ None
  | Some v2' ⇒

  match c with
  | bin_case_i sg ⇒
      match v1', v2' with
      | Vint n1, Vint n2 ⇒ sem_int sg n1 n2
      | _,  _ ⇒ None
      end

  | bin_case_f ⇒
      match v1', v2' with
      | Vfloat n1, Vfloat n2 ⇒ sem_float n1 n2
      | _,  _ ⇒ None
      end

  | bin_case_s ⇒ ...
  | bin_case_l sg ⇒ ...
  | bin_default ⇒ None
  end end end.
```

```
Definition sem_div (v1: val) (t1: type) (v2:  val) (t2: type) (m: mem) : option val :=
  sem_binarith

    (fun sg n1 n2 ⇒
      match sg with
      | Signed ⇒
          if (Int.eq n2 Int.zero)
          || (Int.eq n1 (Int.repr Int.min_signed)) && (Int.eq n2 Int.mone)
          then None
          else Some (Vint (Int.divs n1 n2))
      | Unsigned ⇒
          if (Int.eq n2 Int.zero)
          then None
          else Some (Vint (Int.divu n1 n2))
      end)

    (fun sg n1 n2 ⇒
      match sg with
      | Signed ⇒
          if (Int64.eq n2 Int64.zero)
          || (Int64.eq n1 (Int64.repr Int64.min_signed)) && (Int64.eq n2 Int64.mone)
          then None else Some (Vlong (Int64.divs n1 n2))
      | Unsigned ⇒
          if Int64.eq n2 Int64.zero
          then None
          else Some (Vlong (Int64.divu n1 n2))
      end)

    (fun n1 n2 ⇒ Some (Vfloat  (Float.div   n1 n2)))

    (fun n1 n2 ⇒ Some (Vsingle (Float32.div n1 n2)))

    v1 t1 v2 t2 m.
```

Reasoning about operator arguments

- Some programs are always ok:

```
node f (x : int) returns (y : int);
let
  y = x / 2;
tel
```

Reasoning about operator arguments

- Some programs are always ok:

```
node f (x : int) returns (y : int);
let
  y = x / 2;
tel
```

- Others require preconditions on inputs:

```
node f (x : int) returns (y : int);
let
  y = 2 / x;
tel
```

Reasoning about operator arguments

- Some programs are always ok:

```
node f (x : int) returns (y : int);
let
  y = x / 2;
tel
```

- Others require preconditions on inputs:

```
node f (x : int) returns (y : int);
let
  y = 2 / x;
tel
```

- Others require invariants

```
node f (x : int) returns (y : int);
var w : int
let
  y = x / w;
  w = 1 fby (if w >= 100 then 1 else w + 1);
tel
```

Reasoning about operator arguments—future work?

- Need a semantics to reason about program behaviour.
- But, we're trying to show the existence of a semantics!

Reasoning about operator arguments—future work?

- Need a semantics to reason about program behaviour.
- But, we're trying to show the existence of a semantics!

- Solution? A new semantic model with explicit errors.

```
Inductive value :=
| ‹›
| error
| ‹c : val›.
```

[Boulmé and Hamon (2001): Certifying Synchrony for Free]

Reasoning about operator arguments—future work?

- Need a semantics to reason about program behaviour.
- But, we're trying to show the existence of a semantics!

- Solution? A new semantic model with explicit errors.

```
Inductive value :=
| ‹›
| error
| ‹c : val›.
```

[Boulmé and Hamon (2001): Certifying Synchrony for Free]

- Propagate `error` through expressions, equations, nodes.
- Propagate into future?

Reasoning about operator arguments—future work?

- Need a semantics to reason about program behaviour.
- But, we're trying to show the existence of a semantics!

- Solution? A new semantic model with explicit errors.

```
Inductive value :=
| ‹›
| error
| ‹c : val›.
```

[Boulmé and Hamon (2001): Certifying Synchrony for Free]

- Propagate `error` through expressions, equations, nodes.
- Propagate into future?

- Obligation to prove that `error` never occurs by static analysis or proof.
- Switch to `error`-free model for other proofs and compilation correctness.

Reasoning about operator arguments—future work?

- Need a semantics to reason about program behaviour.
- But, we're trying to show the existence of a semantics!

- Solution? A new semantic model with explicit errors.

```
Inductive value :=
| ‹›
| error
| ‹c : val›.
```

[Boulmé and Hamon (2001): Certifying Synchrony for Free]

- Propagate `error` through expressions, equations, nodes.
- Propagate into future?

- Obligation to prove that `error` never occurs by static analysis or proof.
- Switch to `error`-free model for other proofs and compilation correctness.

Interpreting (non-normalized) Lustre—future work?

- Interpret source-level Lustre programs?

```
node f(x : int) returns (y : int);
var w : int;
let
  (w, y) = (x, w);
tel

node g(x : int) returns (y : int);
let
  y = (0 fby (1 fby (x + y))) + (0 fby y);
tel
```

Interpreting (non-normalized) Lustre—future work?

- Interpret source-level Lustre programs?

```
node f(x : int) returns (y : int);
var w : int;
let
  (w, y) = (x, w);
tel

node g(x : int) returns (y : int);
let
  y = (0 fby (1 fby (x + y))) + (0 fby y);
tel
```

- Denotational semantics in Coq?

  [Paulin-Mohring (2009): A constructive denotational semantics for Kahn networks in Coq]

Interpreting (non-normalized) Lustre—future work?

- Interpret source-level Lustre programs?

```
node f(x : int) returns (y : int);
var w : int;
let
  (w, y) = (x, w);
tel

node g(x : int) returns (y : int);
let
  y = (0 fby (1 fby (x + y))) + (0 fby y);
tel
```

- Denotational semantics in Coq?

  [Paulin-Mohring (2009): A constructive denotational semantics for Kahn networks in Coq]

- Interpreter with bottom?

  [Edwards and Lee (2003): The semantics and execution of a synchronous block-diagram language]

Conclusion

Almost done: Interpreter for NLustre

- Direct approach based on normalized and scheduled Lustre
- Reason by rewriting equivalences
- It will work but it's not glorious
- imperative $\rightsquigarrow$ dataflow constraints $\rightsquigarrow$ imperative

Conclusion

Almost done: Interpreter for NLustre

- Direct approach based on normalized and scheduled Lustre
- Reason by rewriting equivalences
- It will work but it's not glorious
- imperative $\rightsquigarrow$ dataflow constraints $\rightsquigarrow$ imperative

Future work

- Reason about undefined operators
- Treat Lustre (prior to normalization and scheduling)
- Find a canonical interpreter

Vélus: website and source code

- Website: https://velus.inria.fr
- Source: https://github.com/INRIA/velus (non-commercial license)

ECRTS 2020: Euromicro Conf. on Real-Time Systems

- Modena, 7 July – 10 July 2020
- Deadline: 6 February 2020

EMSOFT 2020: Int. Conf. on Embedded Software

- ESWEEK in Shanghai, 11 October – 16 October 2020
- Abstract: 3 April 2020

TETRAMAX: Bilateral TTX calls

- H2020 funded initiative for transferring research to SMEs
- Funding for Technology Transfer Experiments:
  1. Provider: Academic partner in one EU country
  2. Receiver: SME/mid-cap in another EU country
  3. Transfer: a novel HW or SW technology
- Apply by 31 December 2019
- https://www.tetramax.eu/ttx/calls

References I

- Blazy, S., Z. Dargaye, and X. Leroy (Aug. 2006). "Formal Verification of a C Compiler Front-End". In: *Proc. 14th Int. Symp. Formal Methods (FM 2006)*. Vol. 4085. LNCS. Hamilton, Canada: Springer, pp. 460–475.
- Boulmé, S. and G. Hamon (Dec. 2001). "Certifying Synchrony for Free". In: *Proc. 8th Int. Conf. on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR 2001)*. Ed. by R. Nieuwenhuis and A. Voronkov. Vol. 2250. LNCS. Havana, Cuba: Springer, pp. 495–506.
- Edwards, S. A. and E. A. Lee (2003). "The semantics and execution of a synchronous block-diagram language". In: *Science of Computer Programming* 48, pp. 21–42.
- Jourdan, J.-H., F. Pottier, and X. Leroy (Mar. 2012). "Validating LR(1) parsers". In: *21st European Symposium on Programming (ESOP 2012), held as part of European Joint Conferences on Theory and Practice of Software (ETAPS 2012)*. Ed. by H. Seidl. Vol. 7211. LNCS. Tallinn, Estonia: Springer, pp. 397–416.
- Leroy, X. (2009). "Formal verification of a realistic compiler". In: *Comms. ACM* 52.7, pp. 107–115.

References II

- McCoy, F. (1885). *Natural history of Victoria: Prodromus of the Zoology of Victoria*. Frog images.
- Paulin-Mohring, C. (2009). "A constructive denotational semantics for Kahn networks in Coq". In: *From Semantics to Computer Science: Essays in Honour of Gilles Kahn*. Ed. by Y. Bertot, G. Huet, J.-J. Lévy, and G. Plotkin. Cambridge, UK: CUP, pp. 383–413.