Towards a denotational semantics for Vélus

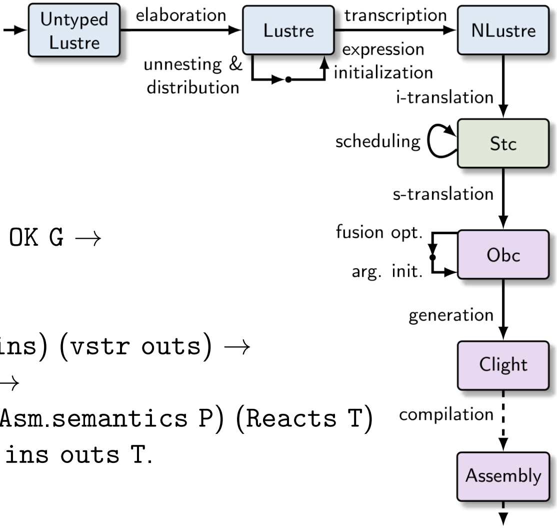
Timothy Bourke **Paul Jeanmaire** Marc Pouzet

ENS, Inria, équipe Parkas

Séminaire Sesterce, Orléans

Vélus, a correct compiler for Lustre

Formally verified




```
Theorem behavior_asm :
  ∀ D G P main ins outs,
    elab_declarations D = OK G →
    wt_ins G main ins →
    wc_ins G main ins →
    sem_node G main (vstr ins) (vstr outs) →
    compile D main = OK P →
    ∃ T, program_behaves (Asm.semantics P) (Reacts T)
         ∧ bisim_io G main ins outs T.
```

Relational semantics

sem_node G f : list (Stream svalue) $\rightarrow$ list (Stream svalue) $\rightarrow \mathbb{P}$

Formal rules

$$\frac{bs = \text{base-of } xs \quad G(f) = n \quad H_*(n.\mathbf{in}) = xs \quad H_*(n.\mathbf{out}) = ys \qquad \text{respects-clock } bs\ H\ n.\mathbf{in} \quad \forall eq \in n.\mathbf{eqs},\ G, bs, H \vdash eq}{G \vdash f(xs) \Downarrow ys}$$

$$\frac{bs, H \vdash e :: ck \Downarrow H_*(x)}{G, bs, H \vdash x =_{ck} e} \qquad \frac{bs, H \vdash \boldsymbol{e} \Downarrow vs \quad bs, H \vdash ck \Downarrow \text{base-of } vs \quad G \vdash f(vs) \Downarrow H_*(\boldsymbol{x})}{G, bs, H \vdash \boldsymbol{x} =_{ck} f\ (\boldsymbol{e})}$$

$$\frac{bs, H \vdash e :: ck \Downarrow vs \qquad H_*(x) \approx \text{fby } ([\![c]\!], vs)}{G, bs, H \vdash x =_{ck} c \ \texttt{fby}\ e} \qquad \frac{bs, H \vdash \boldsymbol{e} \Downarrow vs \quad \text{bools-of } H_*(y)\ rs \quad bs, H \vdash ck \Downarrow \text{base-of } vs \qquad \forall k,\ G \vdash f\left(\text{mask}^k_{rs}\ vs\right) \Downarrow \text{mask}^k_{rs}\ H_*(\boldsymbol{x})}{G, bs, H \vdash \boldsymbol{x} =_{ck} \left(\texttt{restart}\ f\ \texttt{every}\ y^{ck_y}\right)(\boldsymbol{e})}$$

Fig. 6. Dataflow semantics: mutually inductive semantics of equations and nodes

Theorem (determinism)

$$\forall G, f, xs, ys, ys', \ \text{is\_causal } G \implies$$

$$\text{sem\_node } G\ f\ xs\ ys \wedge \text{sem\_node } G\ f\ xs\ ys' \implies ys \equiv ys'$$

The synchronous denotational (**SD**) model

$$
\begin{array}{ll}
i^{\#}[\epsilon] & = \epsilon \\
i^{\#}[true.cl] & = v.i^{\#}[cl] \\
i^{\#}[false.cl] & = abs.i^{\#}[cl] \\
\\
op^{\#}(s_1, s_2) & = \epsilon \text{ if } s_1 = \epsilon \text{ or } s_2 = \epsilon \\
op^{\#}(abs.s_1, abs.s_2) & = abs.op^{\#}(s_1, s_2) \\
op^{\#}(v_1.s_1, v_2.s_2) & = (v_1\, op\, v_2).op^{\#}(s_1, s_2) \\
\\
\texttt{fby}^{\#}(\epsilon, ys) & = \epsilon \\
\texttt{fby}^{\#}(abs.xs, abs.ys) & = abs.\texttt{fby}^{\#}(xs, ys) \\
\texttt{fby}^{\#}(x.xs, y.ys) & = x.\texttt{fby1}^{\#}(y, xs, ys) \\
\texttt{fby1}^{\#}(v, \epsilon, ys) & = \epsilon \\
\texttt{fby1}^{\#}(v, abs.xs, abs.ys) & = abs.\texttt{fby1}^{\#}(v, xs, ys) \\
\texttt{fby1}^{\#}(v, w.xs, s.ys) & = v.\texttt{fby1}^{\#}(s, xs, ys) \\
\\
\texttt{when}^{\#}(s_1, s_2) & = \epsilon \text{ if } s_1 = \epsilon \text{ or } s_2 = \epsilon \\
\texttt{when}^{\#}(abs.xs, abs.cs) & = abs.\texttt{when}^{\#}(xs, cs) \\
\texttt{when}^{\#}(x.xs, true.cs) & = x.\texttt{when}^{\#}(xs, cs) \\
\texttt{when}^{\#}(x.xs, false.cs) & = abs.\texttt{when}^{\#}(xs, cs) \\
\\
\texttt{merge}^{\#}(s_1, s_2, s_3) & = \epsilon \text{ if } s_1 = \epsilon \text{ or } s_2 = \epsilon \text{ or } s_3 = \epsilon \\
\texttt{merge}^{\#}(abs.cs, abs.xs, abs.ys) & = abs.\texttt{merge}^{\#}(cs, xs, ys) \\
\texttt{merge}^{\#}(true.cs, x.xs, abs.ys) & = x.\texttt{merge}^{\#}(cs, xs, ys) \\
\texttt{merge}^{\#}(false.cs, abs.xs, y.ys) & = y.\texttt{merge}^{\#}(cs, xs, ys)
\end{array}
$$

Fig. 2. Synchronous data-flow semantics

Colaço & Pouzet, *Clocks as first class abstract types*, EMSOFT'03

- Set of streams: $D^* \cup D^\omega$
- Elements: *abs* or *v*
- CPO with prefix order
- $\perp$ (or ϵ) is the empty stream

$$x = 0 \rightarrow \text{pre } y;$$
$$y = x + 1;$$

x	012⊥34567891
y	123⊥45678910

Coq implementation

```
CoFixpoint filter (f : A → 𝔹) (s : Str A) : Str A := (* ? *)
```

Thanks to a generic CPO library[1]

- Fixpoint operator: `∀ (D:cpo), FIXP : (D -C→ D) -C→ D.`
- Associated proof principle:

  `∀ F P, admissible P → P ⊥ → (∀ x, P x → P (F x)) → P (FIXP F).`

- Guardedness by using a transparent "not-yet" element

```
CoInductive Str A :=
| Eps : Str → Str
| Con : A → Str → Str.
```

```
CoFixpoint Str_bot :=
  (* empty stream *)
  Eps Str_bot.
```

- Prefix order modulo Eps, example:

$$\bot = \epsilon\,\epsilon\,\epsilon\,\ldots \leq x\,\epsilon\,\epsilon\,\ldots$$

$$\epsilon\,\epsilon\,x\,y\,\epsilon\,\epsilon\,\ldots \leq x\,\epsilon\,\epsilon\,\epsilon\,\epsilon\,\epsilon\,y\,\ldots$$

---

[1] Christine Paulin-Mohring, *A constructive denotational semantics for Kahn networks in Coq*, From semantics to CS, 2007

Coq implementation

Language semantics

Datatype: Str sampl

```
Inductive error : Type :=
| error_Ty  (* data error *)
| error_Cl  (* scheduling error *)
| error_Op. (* arithmetic error *)
```

```
Inductive sampl : Type :=
| abs
| pres (a : CompCert.value)
| err  (e : error).
```

Denotation functions

```
Definition denot_exp (e : exp) :
(* nodes *)  (* ins *)                (* env *)
  F_prod —C→ S_prod —C→ Str 𝔹 —C→ S_prod —C→ Str sampl.

Definition denot_equation (equ : equation) :
  F_prod —C→ Str 𝔹 —C→ S_prod —C→ S_prod.

Definition denot_node n envG envI :=
  FIXP (denot_equation n.(equ) envG envI (union envI)).

Definition denot_global G := FIXP (λ envG f ⇒ denot_node (G f) envG).
```

Witness of the relation?

```
Conjecture existence :
  ∀ G f xs,
    sem_node G f xs (denot_global G f xs).
```

Problem n°1: finite streams

```
node f (c : bool) returns (n : int)
let
  n = if c then 0 else n + 1;
tel
```

```
Lemma denot_inf :
  ∀ G xs,
    Forall node_causal (nodes G) →
    all_infinite xs →
    ∀ f, all_infinite (denot_global G f xs).
```

Witness of the relation?

```
Conjecture existence :
  ∀ G f xs,
    Forall node_causal (nodes G) →
    sem_node G f xs (denot_global G f xs).
```

Problem n°2: typing errors

```
node f (c : bool) returns (n : int)
let
  n = if c = 3 then 0 else 1;
tel
```

Witness of the relation?

```
Conjecture existence :
  ∀ G f xs,
    Forall wt_node (nodes G) →
    Forall node_causal (nodes G) →
    sem_node G f xs (denot_global G f xs).
```

Problem n°3: synchronization errors

```
node f (c : bool) returns (n : int)
let
  n = 0 fby n + (1 when c);
tel
```

Witness of the relation?

```
Conjecture existence :
  ∀ G f xs,
    Forall wt_node (nodes G) →
    Forall wc_node (nodes G) →
    Forall node_causal (nodes G) →
    sem_node G f xs (denot_global G f xs).
```

Problem n°4: operator failure/arithmetic runtime errors

```
node f (c : bool) returns (n : int)
let
  n = 4 / 0;
tel
```

```
node f (c : bool) returns (n : int)
let
  n = 4 / g(c);  -- node g (c : bool) returns (x : int)
tel
```

[see op_correct.v]

Witness of the relation

```
Theorem existence :
  ∀ G f xs,
    Forall wt_node (nodes G) →
    Forall wc_node (nodes G) →
    Forall node_causal (nodes G) →
    let envG := denot_global G in
    op_correct G envG →
    sem_node G f xs (envG f xs).
```

Conclusion
(perspectives)

How to satisfy `op_correct`?

- Static analysis
- Reasoning techniques on dataflow programs
- Pre/post-conditions, node contracts. . . ?

Thank you! Questions?