# Génération de code séquentiel pour Lustre

Marc Pouzet

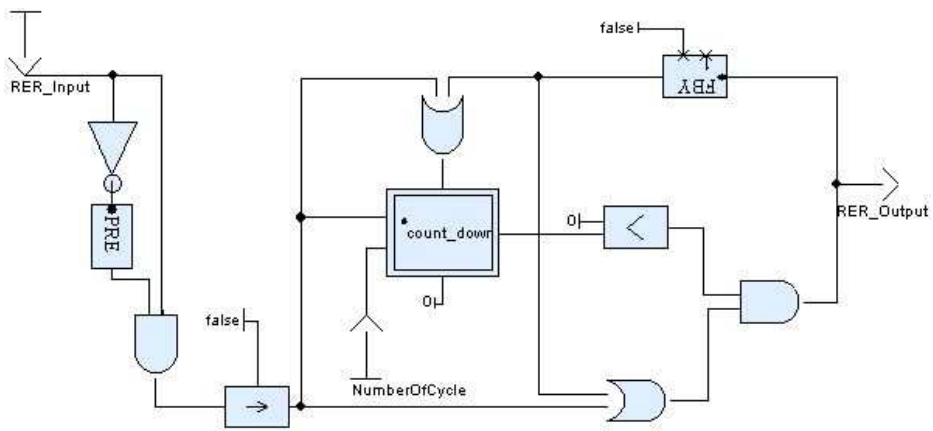
Marc.Pouzet@ens.fr

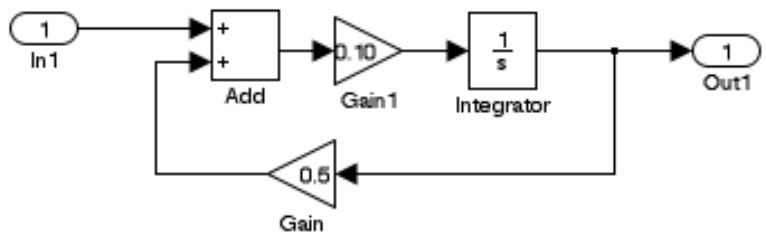
Calcul parallèle et réactif
M1
Octobre 2024

# The problem

- **Input:** a parallel data-flow network made of synchronous operators. E.g., **Lustre**, **Scade**, **Simulink**
- **Output:** a sequential procedure (e.g., C, Java) to compute one step of the network: static scheduling

## Examples: **Scade** and **Simulink**



# This is part of a more general question

How to "compile the parallelism", i.e., generate seq. code which:

- preserves the parallel semantics,
- treats all programs with no ad-hoc restriction.

## Why sequentializing a parallel program?

- Often far more efficient that the parallel version.
- Get a time predictable implementation (real-time system).
- At the moment, tools for analysing the Worst Case Execution Time (WCET) work well for sequential code only.

This is not contradictory with the question of generating parallel code.

Both questions are interesting.

# The basic intuition

Implement $f : Stream(T) \to Stream(T')$ as a pair $(s_0, f_t)$:

- an initial state $s_0 : S$;
- a sequential step function: $\langle f_t : S \times T \to T' \times S \rangle$

An equation $y = f(x)$ can be computed sequentially such that:

$$\forall n \in \mathbb{N}, y_n, s_{n+1} = f_t(s_n, x_n)$$

In practice, the internal state is modified in place.

Equivalently, decompose the step function in two:

- an initial state: $s_0 : S$
- a value function: $f_v : S \times T \to T'$
- a transition function ("commit"): $f_c : S \times T \to S'$

$$\forall n \in \mathbb{N}, y_n = f_v(s_n, x_n) \wedge s_{n+1} = f_c(s_n, x_n)$$

# Several inputs/several outputs

If $f : Stream(T_1) \times ... \times Stream(T_n) \rightarrow Stream(T'_1) \times ... \times Stream(T'_m)$

A simple solution does the same:xs

- an initial state: $s_0 : S$
- a value function: $f_v : S \times T_1 \times ... T_n \rightarrow T'_1 \times ... \times T'_k$

but this solution may forbid correct feedback loops, e.g.:

$$(y, z) = copy(t, y)$$

where $copy(x, y) = (x, y)$

Indeed, if *copy* is inlined, we get the two equations:

$$y = t \text{ and } z = y$$

which can be compiled into two assignments.

# Two classical implementations

- Periodic sampling

```
s := s0;
every clock tick
 read_input e;
 let o, s' = ft s e in
 s := s';
 write_output o
end
```

- Event driven

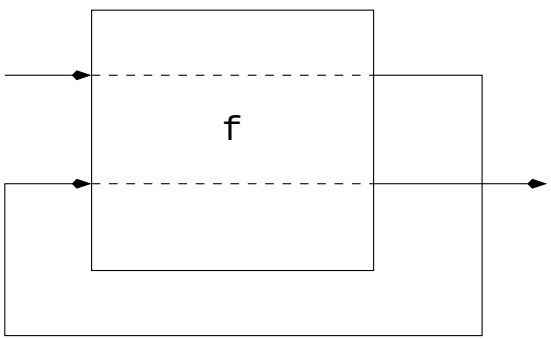
```
s := s0;
everytime e is present
 let o, s' = ft s e in
 s := s';
 write_output o
end
```

Check that the inter-arrival time between events or clock ticks in greater than the WCET of the read/compute/write.

# Modular Static Scheduling

Sequentializing parallel code cannot be done once for all, independently from the context.

## Example [1]




```
node copy(a, b:bool) returns (c, d:bool);
  let
     c = a; (* 1 *)
     d = b; (* 2 *)
  tel;

node loop(t:bool) returns (z:bool);
  var y: bool;
  let (y, z) = copy(t, y);
  tel;
```

`loop(t)` whould run perfectly in a parallel implementation.

[1] The example is due to Georges Gonthier.

# Modular Static Scheduling

```
void step_copy_one(int *a, int *b, int *c, int *d) {
  *c = *a;
  *d = *b;}

void step_copy_two(int *a, int *b, int *c, int *d) {
  *d = *b;
  *c = *a;}

void loop_step(int *t, int *z) {
  int y;
  step_copy_one(t, &y, &y, z);}
```

Only the first implementation of copy can be used.

Generating a single transition function is not sufficient.

Two main approaches has been followed in synchronous compilers.

# The two main approaches to code generation

## Maximal Static Expansion ("white boxing")

- Function calls are statically (inlined).
- The way it is done in the **Lustre** compiler (VERIMAG).
- Efficient enumeration techniques can be applied to generate finite state automata [Raymond PhD. Thesis[Ray91b], Halbwachs et al. [HRR91]].
- But code size can be prohibitive.
- This technique is illustrated in the second part of these notes.

## Single Loop Code Generation ("black boxing")

- A single code repeated infinitely.
- Modular: one node produces one step function.
- Makes tracability of the compiler simpler.
- Imposes stronger causality constraints: every loop must cross a delay.
- The approach of **Scade** KCG.

# An intermediate solution ("grey boxing")

- Instead of producing a single step function per node, produce several together with a partial order.
- They must be called in an order compatible with this partial order.
- E.g., for copy, produces two, one that computes `c := a`, one that computes `d := b`. [2]
- This is called the Modular Static Scheduling problem.
- Identified in 1988 by Pascal Raymond who proposed a first algorithm. [Ray88]
- The Optimal Modular Static Scheduling problem is when the number of step functions is minimal.
- Several solutions has been proposed, notably [LST09, PR09, PR10].

In this class, we focus on the simpler single loop code generation problem.

[2] Surely, for such a small function, inlining is preferable!

# Single loop code generation

# Historical note

- The technique of single loop code generation dates back to the PhD. thesis of John Plaice [Pla88].
- Enumeration techniques were introduced to generate far better code in the form of finite state machines:
  - Forward technique by Plaice [Pla88];
  - backward technique by Raymond [Ray91a]
- Modular techniques that avoid full inlining were known and implemented in the industrial compiler of Lustre made by the Verilog company (1995 and after).
- Its formalisation and proof was given in a more general setting in [CP98].
- The "clock directed modular code generation" was introduced in 1996 in the Lucid Synchrone compiler [Pou06]. It was implemented in an internal compiler prototype for Lustre (called ReLuC), at Verilog in 2000.
- It was presented in [BCHP08].
- It is implemented in the industrial compiler of Scade 6 and used since 2008.

# A reference compiler for a synchronous data-flow kernel [3]

MiniLS: a minimalistic clocked data-flow language as input.

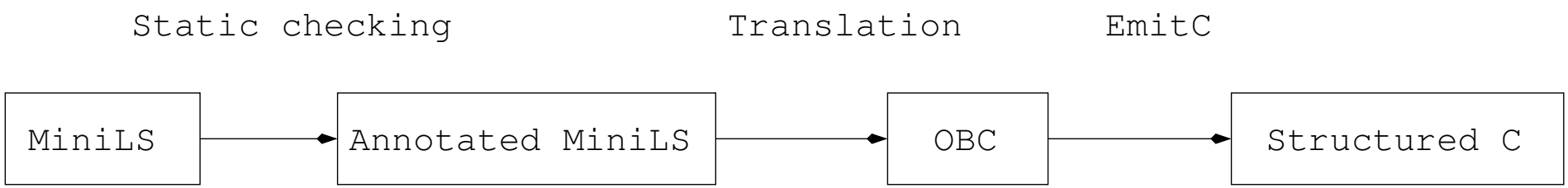
- used as some sort of "typed assembly language".
- General enough to be used as a target language for **Lustre**.
- and a target for control structures like hierarchial automata.

## Objective

- Single loop code generation.
- Compilation into an intermediate "object based" language that represent transition functions.
- Then, translated into imperative code (e.g., structured C, OCaml, Java).

[3] The notes are adapted from [BCHP08]

# Organization of the Compiler



# The source language

Expressions:

$$\begin{array}{rcl} a & ::= & v \mid x \mid v \texttt{ fby } a \\ & & \mid op\,(a, ..., a) \\ & & \mid a \texttt{ when } C(x) \\ & & \mid \texttt{merge } x\ (C \to a)\ ...\ (C \to a) \end{array}$$

Equations:

$$D \quad ::= \quad x = a \mid (x, ..., x) = f\,(a, ..., a) \texttt{ every } a \mid D \texttt{ and } D$$

Function definitions, constants:

$$\begin{array}{rcl} d & ::= & \texttt{node } f(p) = p \texttt{ with var } p \texttt{ in } D \\ p & ::= & x : bt; ...; x : bt \\ v & ::= & C \mid i \end{array}$$

# When/Merge

| $h$ | `true` | `false` | `true` | `false` | ... |
|---|---|---|---|---|---|
| $x$ | $x_0$ | $x_1$ | $x_2$ | $x_3$ | ... |
| $y$ | $y_0$ | $y_1$ | $y_2$ | $y_3$ | ... |
| $v$ `fby` $x$ | $v$ | $x_0$ | $x_1$ | $x_2$ | ... |
| $x + y$ | $x_0 + y_0$ | $x_1 + y_1$ | $x_2 + y_2$ | $x_3 + y_3$ | ... |
| $z = x$ `when true`$(h)$ | $x_0$ | | $x_2$ | | ... |
| $t = y$ `when false`$(h)$ | | $y_1$ | | $y_3$ | ... |
| `merge` $h$ $(\texttt{true} \to z)$ $(\texttt{false} \to t)$ | $x_0$ | $y_1$ | $x_2$ | $y_3$ | ... |

- $v$ `fby` $x$ is the unit delay initialized with $v$.
- the `merge` constructs combines two complementary sequences

# Example (counter)

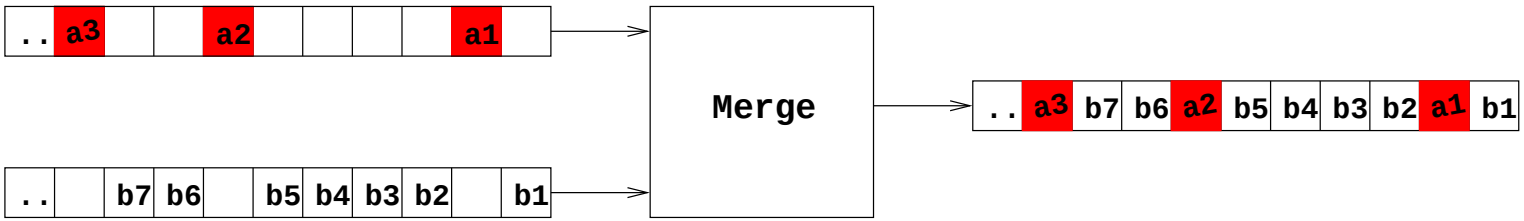
"Counts the number of ocurrences of tick".

```
node counting (tick:bool) = (o:int) with
var v: int in
    o = (0 fby o) + v
and v = if tick then 1 else 0
```

| tick | true | true | true | false | false | true | true | ... |
|---|---|---|---|---|---|---|---|---|
| o | 1 | 2 | 3 | 3 | 3 | 4 | 5 | ... |
| v | 1 | 1 | 1 | 0 | 0 | 1 | 1 | ... |

# The n-ary merge operator

- The `merge c x y` operator combines two complementary flows (flows on complementary clocks) to produce a faster one




Example: `merge c (a when c) (b whenot c)`

Generalization:

- generalized to $n$ inputs of an enumerated type $t$ with:

$$t = C_1 \mid \ldots \mid C_n$$

- the sampling $e$ `when` $c$ is now written $e$ `when true`$(c)$, i.e.,

$$bool = \texttt{true} \mid \texttt{false}$$

# Reseting a behavior

How to make the node `counting` "resetable", that is, whenever `r` is true, all its internal registers are reset to an initial value?

One has to reprogram `counting` into:

```
node counting_r (r:bool; tick:bool) = (o:int) with
  var v: int in
    o = (if r then 0 else 0 fby o) + v
    and v = if tick then 1 else 0
```

- There is no modular reset in Lustre. Making a component "resetable" is painful and error prone.
- The above encoding would lead to very bad sequential code.
- Two good reasons to make it a primitive in the language.

## Specific notation:

$$f\,(a_1, ..., a_n)\ \texttt{every}\ c$$

all the registers used in the definition of node $f$ are reset when the boolean condition $c$ is true

# Derived Operators

## Mux/conditional:

$$\texttt{if } x \texttt{ then } e_2 \texttt{ else } e_3 \quad = \quad \texttt{merge } x \; (\texttt{true} \rightarrow e_2 \texttt{ when true}(x)) \; (\texttt{false} \rightarrow e_3 \texttt{ when false}(x))$$

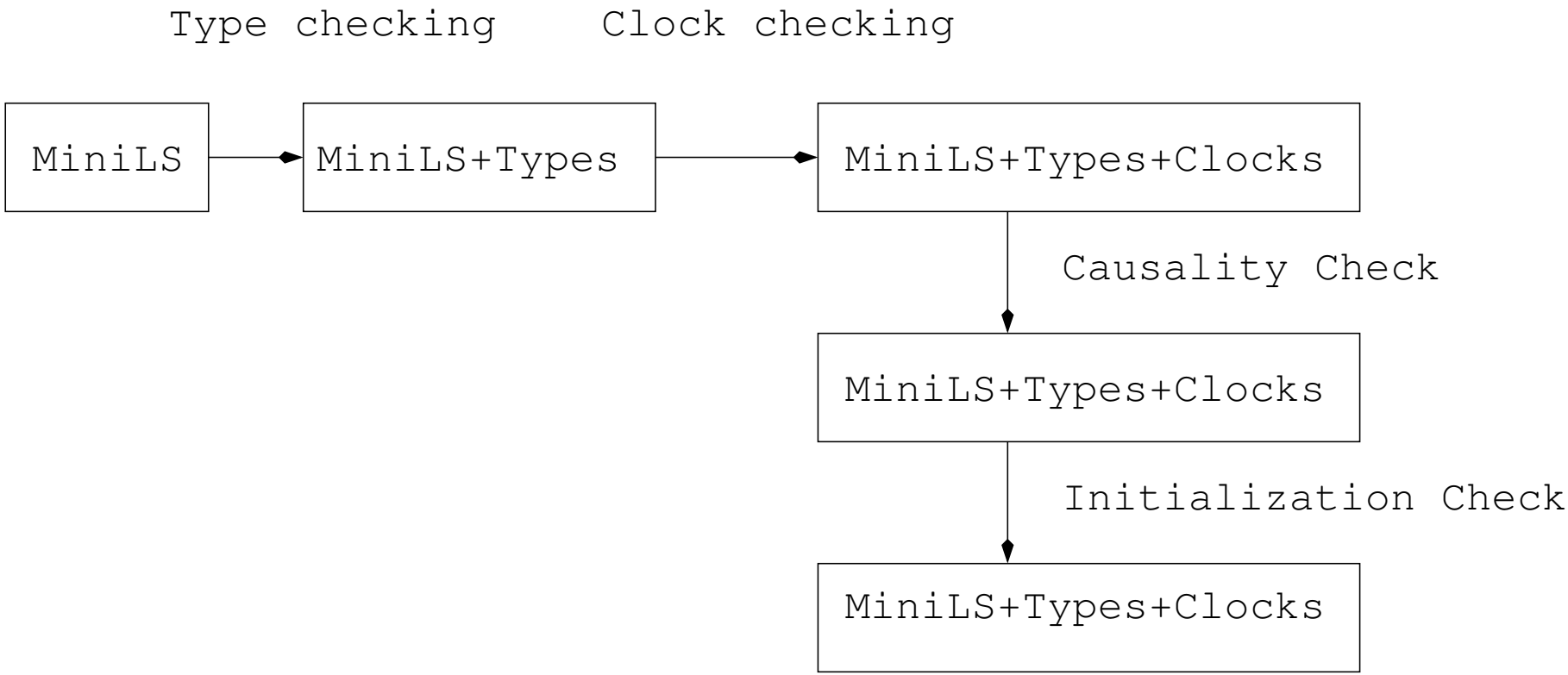
## Initialization and un-initialized delay:

$$y = e_1 \texttt{ -> } e_2 \quad = \quad y = \texttt{if } \mathit{init} \texttt{ then } e_1 \texttt{else } e_2 \text{ and } \mathit{init} = \texttt{true fby false}$$

$$\texttt{pre}\,(e) \quad = \quad \mathit{nil} \texttt{ fby } e \text{ where } \mathit{nil} \text{ is a value of the type of } e$$

Either add them to the language kernel or express them into it.

# Static Checking



# Type and Clock checking

An intermediate language where every expression is annotated with a type expression ($bt$) and a clock expression ($ck$).

$$
\begin{array}{lcl}
a & ::= & e^{ck}_{bt} \\
e & ::= & v \mid x \mid v \texttt{ fby } a \mid a \texttt{ when } C(x) \mid op(a, ..., a) \\
 & & \mid \texttt{merge } x\ (C \rightarrow a)\ ...\ (C \rightarrow a) \\
D & ::= & x = a \mid f(a, ..., a) \texttt{ every } a \mid D \texttt{ and } D \\
d & ::= & \texttt{node } f(p) = p \texttt{ with var } p \texttt{ in } D
\end{array}
$$

## Clock expression:

- The clock $clock(s)$ of a sequence $s$ is a boolean sequence such that $clock(s) = true$ iff $s$ is present
- An expression $e$ is annotated a boolean formula $ck$

$$ck ::= \texttt{base} \mid ck \texttt{ on } C(x)$$

- `base` is the "base" clock of the node; it is always true.
- $ck$ on $C(x)$ is true when $ck$ is true and $x = C$.

# Clock Verification

Type and clock checking are performed in order.

- Clock environment: $\mathcal{H} ::= [ck_1/x_1; ...; ck_n/x_n]$
- $\mathcal{H} \vdash a : ck$ means that $a$ is well annotated in $\mathcal{H}$ with clock type $ck$.

$$\text{(Annot)}\ \frac{\mathcal{H} \vdash e : ck}{\mathcal{H} \vdash e^{ck}_{bt} : ck} \qquad \text{(Op)}\ \frac{\mathcal{H} \vdash a_1 : ck \ ... \ \mathcal{H} \vdash a_n : ck}{\mathcal{H} \vdash op\,(a_1, ..., a_n) : ck} \qquad \text{(Const)}\ \mathcal{H} \vdash v : \texttt{base}$$

$$\text{(Var)}\ \mathcal{H}, x : ck \vdash x : ck \qquad \text{(Fby)}\ \frac{\mathcal{H} \vdash a : ck}{\mathcal{H} \vdash v\ \texttt{fby}\ a : ck} \qquad \text{(When)}\ \frac{\mathcal{H} \vdash a : ck \qquad \mathcal{H} \vdash x : ck}{\mathcal{H} \vdash a\ \texttt{when}\ C(x) : ck\ \texttt{on}\ C(x)}$$

$$\text{(Merge)}\ \frac{\mathcal{H} \vdash x : ck \quad \mathcal{H} \vdash a_1 : ck\ \texttt{on}\ C_1(x) \ ... \ \mathcal{H} \vdash a_n : ck\ \texttt{on}\ C_n(x)}{\mathcal{H} \vdash \texttt{merge}\ x\ (C_1 \rightarrow a_1) \ ... \ (C_n \rightarrow a_n) : ck}$$

# Clock Checking

We consider the simple case where all input/outputs of a node have the same clock [4]

$$\frac{\forall i \in [1..k]\ \mathcal{H} \vdash x_i : ck \qquad \forall i \in [1..n]\ \mathcal{H} \vdash a_i : ck \qquad \mathcal{H} \vdash c : ck}{\mathcal{H} \vdash (x_1, ..., x_k) = f\,(a_1, ..., a_n)\ \texttt{every}\ c : ck}\ (\textsc{Call})$$

$$\frac{\vdash_{\texttt{base}} p : \mathcal{H}_p \qquad \vdash_{\texttt{base}} q : \mathcal{H}_q \qquad \vdash r : \mathcal{H}_r \qquad \mathcal{H}_p, \mathcal{H}_q, \mathcal{H}_r \vdash D}{\vdash \texttt{node}\ f(p)(q) = \texttt{var}\ r\ \texttt{in}\ D}\ (\textsc{Node})$$

$$\vdash x_1 : bt_1, ..., x_n : bt_n : [x_1 : ck_1; ...; x_n : ck_n]\ (\textsc{Pat})$$

$$\vdash_{\texttt{base}} x_1 : t_1, ..., x_n : t_n : [x_1 : \texttt{base}; ...; x_n : \texttt{base}]\ (\textsc{Param})$$

[4] Lustre is more powerful as it allows for input/outputs to be on different clocks provided the first input is on the base clock.

# Clock checking

$$\frac{\mathcal{H} \vdash x : ck \qquad \mathcal{H} \vdash a : ck}{H \vdash x = a} \text{ (Eq)}$$

$$\frac{\mathcal{H} \vdash D_1 \qquad \mathcal{H} \vdash D_2}{\mathcal{H} \vdash D_1 \text{ and } D_2} \text{ (And)}$$

This system is very limited.

The language kernel has no data-structures, no type polymorphism.

All inputs and outputs must be on the same clock.

# Extensions

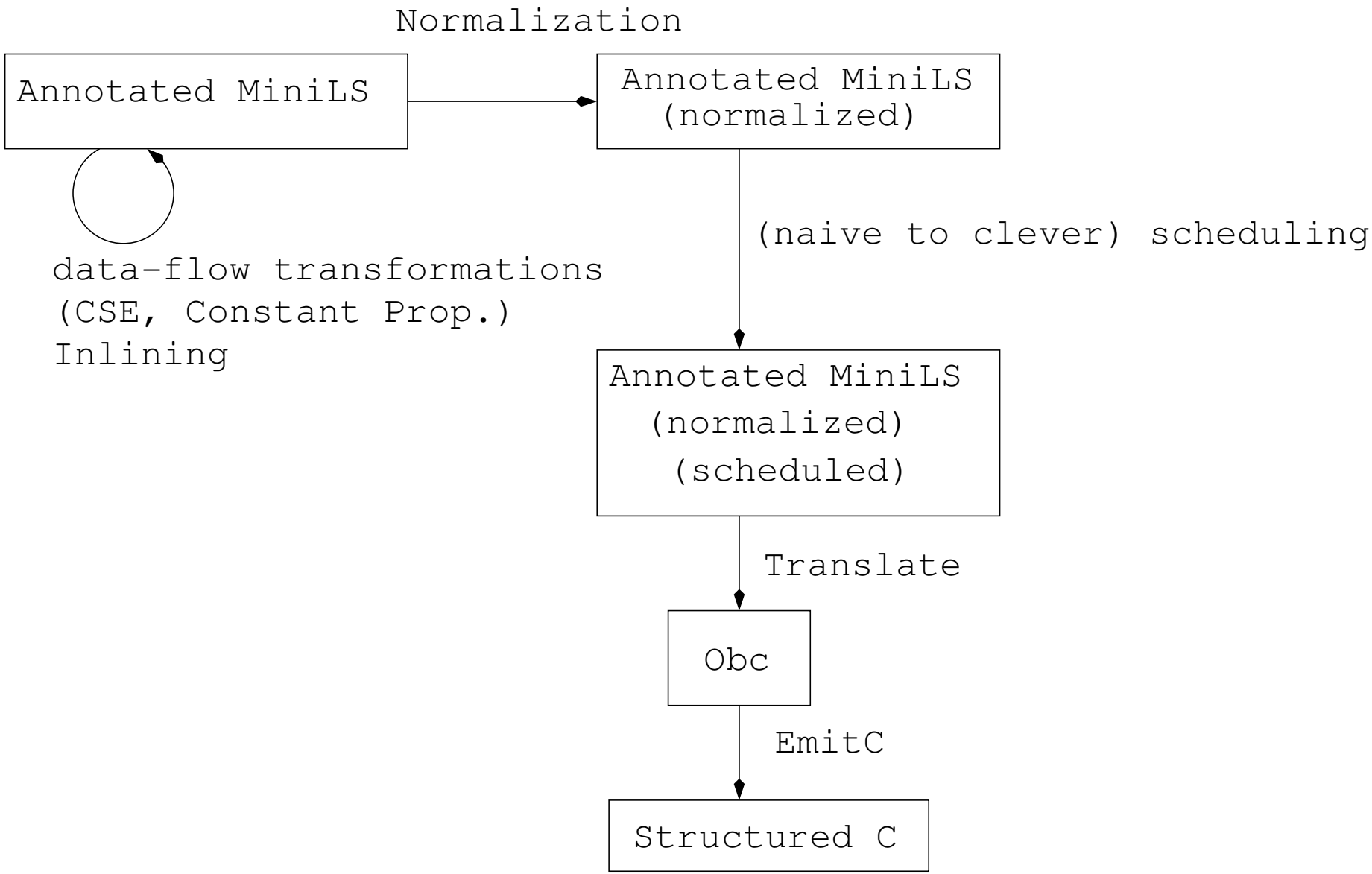
Some examples of equations and functions that can be defined in **Scade 6**.

```
x = (1, 2)
x = {a = 1; b = 2}
x = if c then (1,2) else (3, 4)

node hold(i: 't; clock h: bool; a: 't when h) returns (o:'t)
   o = merge(h; a; (i -> pre o) when not h);
```

- Enrich the type language and clock type language with tuples and polymorphism.
- The clock calculus of Lustre can be defined as a dependent type system [CP96, BH01]
- A simpler one reminiscent of the ML-type system [CP03].
- Programs from the enrichied language can be translated into the basic language.
- Yet, it is sufficient to illustrate how clocks are used to generate code.

# Translation into sequential code



# Putting Equations in Normal Form

- Prepare equations before the translation.
- Identify state variables *vs* temporaries.
- Rewrite equations such that delays and function applications do not appear in nested expressions.

## Normal Form:

$$
\begin{array}{lcl}
a & ::= & e^{ck}_{bt} \\
e & ::= & a \texttt{ when } C(x) \mid op\,(a, ..., a) \mid x \mid v \\
ce & ::= & \texttt{merge } x\ (C \rightarrow ca)\ ...\ (C \rightarrow ca) \mid e \\
ca & ::= & ce^{ck}_{bt} \\
eq & ::= & x = ca \mid x = v \texttt{ fby } a \\
 & & \quad \mid (x, ..., x) = f\,(a, ..., a) \texttt{ every } a \\
D & ::= & eq \mid eq \texttt{ and } D
\end{array}
$$

Notation: $[eq_1; ...; eq_n]$ or $(eq_i)_{i \in [1...n]}$ for $eq_1 \texttt{ and } ... eq_n$. $eq_1.eq_2$ for the concatenation.

# Checking the correctness of the normalization

The normalization is a simple rewritting. Its correctness can be validated independently, a posteriori.

- Given a list of equations, a (non trusted) normalization function returns a list of normalized equations and a substitution:

$$normalize([eq_1; ...; eq_n]) \stackrel{def}{=} [eq'_1; ...; eq'_m], [e_1/x_1, ..., e_k/x_k]$$

- Check that:

$$sub([eq'_1; ...; eq'_m])[e_1/x_1, ..., e_l/x_k] = [eq_1; ...; eq_n] = eqs$$

  with:

$$sub(eqs')[e_1/x_1, ..., e_l/x_k] = sub(sub(eqs')[e_1/x_1])[e_2/x_2, ..., e_k/x_k]$$

- prove that the substitution preserves the semantics, that is, if $e_j : bt_j$, $j \in [1..k]$, $eqs$ and $\texttt{var}\ (x_j : bt_j)_{j\in[1..k]}\ \texttt{in}\ (x_j = e_j)_{j\in[1..k]}.eqs'$ are semantically equivalent.
- If the substitution succeeds, then the translation preserves the semantics.

# Data-structures (tuples, records)

Typing/clocking constraints and normalisation can be extended.

## Types and clocks

$$ty ::= ty \times ... \times ty \mid bt \qquad ct ::= ct \times ... \times ct \mid ck$$

For records, consider that the component must be on the same clock $ck$.

## Normalisation

Rewrites equations into more elementary ones.

E.g.,: rewrite `(x1, x2) = (1, 2)` into `x1 = 1 and x2 = 2`

`(x1, x2) = if c then (1, 2) else (3, 4)` into
`x1 = if c then 1 else 3` and `x2 = if c then 2 else 4`.

The same approach that separates the rewriting from its validation can be followed.

# Syntactic Dependences and Scheduling

After the normalization, equations are scheduled according to data-dependences.

- A (non trusted) scheduling function:

$$\text{schedule: eq list} \rightarrow \text{eq list}$$

- Define what is a well scheduled sequence of equations:
  - An equation $x = ca$ must appear before any read of $x$ (data-dependence)
  - An equation $x = v$ `fby` $a$ must appear after any read of $x$ (anti-dependence)
- Prove that the semantics of a set of equations does not depend on the relative order between them.
- Check that the set of equations is well scheduled.

By separating the scheduling function from the verification that the result is well scheduled, it is easier to implement a clever scheduling function.

# Well Scheduled Equations

Notation [] is an empty sequence of equations; $L = [eq_1; ...; eq_n]$ is a set of $n$ equations; $eq.L$ is a shortcut for $[eq; eq_1; ...; eq_n]$.

$Left\,(ca)$ returns the list of variables that are read in $ca$.

The predicate $SCH([eq_1; ...; eq_n]) : r, w, mem$ means:

- the sequence $[eq_1; ...; eq_n]$ is well scheduled;
- it reads variables in $r$, write variables in $w$ and memories in $mem$.

$$\frac{x \notin Left\,(ca)}{SCH(x = ca) : Left\,(ca), \{x\}, \emptyset} \qquad SCH(x = (v \texttt{ fby } a)^{ck}_{bt}) : Left\,(a), \emptyset, \{x\}$$

$$\frac{r = \cup_{0 \leq i \leq n} Left\,(a_i) \cup Left\,(c) \qquad \{y_1, ..., y_m\} \cap r = \emptyset}{SCH(\vec{y} = f\,(\vec{a}) \texttt{ every } c) : r, \{y_1, ..., y_m\}, \emptyset}$$

$$\frac{SCH(eq) : r, w, mem \quad SCH(L) : r', w', mem' \quad w' \cap r = \emptyset \quad mem \cap r' = \emptyset}{SCH(eq.L) : r \cup r', w \cup w', mem \cup mem'}$$

$$SCH([]) : \emptyset, \emptyset, \emptyset$$

# Example (the counting node)

Once the type and clock checking and annotation are done, we get:

$$
\begin{aligned}
&\texttt{node}\ \mathit{counting}\,(\mathit{tick} : \mathit{bool}, \mathit{top} : \mathit{bool})\ \texttt{with}\,(o : \mathit{int})\ \texttt{in}\,(v : \mathit{int})\\
&\qquad o \quad = \quad (\texttt{merge}\ \mathit{top}\ (\texttt{true} \to (v^{b}\ \texttt{when}\ \texttt{true}(\mathit{top}))^{ck_1})\\
&\qquad\qquad\qquad\qquad (\texttt{false} \to (((0\ \texttt{fby}\ o^{b})^{b} + v^{b})^{b}\ \texttt{when}\ \texttt{false}(\mathit{top}))^{c}\\
&\texttt{and}\ v \quad = \quad (\texttt{merge}\ \mathit{tick}\ (\texttt{true} \to (1^{b}\ \texttt{when}\ \texttt{true}(\mathit{tick}))^{ck_3})\\
&\qquad\qquad\qquad\qquad (\texttt{false} \to (0^{b}\ \texttt{when}\ \texttt{false}(\mathit{tick}))^{ck_4}))^{b}
\end{aligned}
$$

$ck_1 = b\ \texttt{on}\ \texttt{true}(\mathit{top})$
$ck_2 = b\ \texttt{on}\ \texttt{false}(\mathit{top})$
$ck_3 = b\ \texttt{on}\ \texttt{true}(\mathit{tick})$
$ck_4 = b\ \texttt{on}\ \texttt{false}(\mathit{tick})$

We write $b$ as a short-cut for `base`.

# Example (the counting node)

After the normalization, it becomes:

$$
\begin{array}{rcl}
\texttt{node}\ \mathit{counting}\,(\mathit{tick} : \mathit{bool}, \mathit{top} : \mathit{bool})\ \texttt{with}\,(o : \mathit{int})\ \texttt{in}\,(v : \mathit{int}) & & \\
o & = & (\texttt{merge}\ \mathit{top}\ (\texttt{true} \to (v^b\ \texttt{when}\ \texttt{true}(\mathit{top}))^{ck_1}) \\
 & & \qquad (\texttt{false} \to ((t^b + v^b)^b\ \texttt{when}\ \texttt{false}(\mathit{top}))^{ck_2}))^b \\
\texttt{and}\ t & = & (0\ \texttt{fby}\ o^b)^b \\
\texttt{and}\ v & = & (\texttt{merge}\ \mathit{tick}\ (\texttt{true} \to (1^b\ \texttt{when}\ \texttt{true}(\mathit{tick}))^{ck_3}) \\
 & & \qquad (\texttt{false} \to (0^b\ \texttt{when}\ \texttt{false}(\mathit{tick}))^{ck_4}))^b
\end{array}
$$

where :

$ck_1 = b\ \texttt{on}\ \texttt{true}(\mathit{top})$

$ck_2 = b\ \texttt{on}\ \texttt{false}(\mathit{top})$

$ck_3 = b\ \texttt{on}\ \texttt{true}(\mathit{tick})$

$ck_4 = b\ \texttt{on}\ \texttt{false}(\mathit{tick})$

# Example (the counting node)

After the scheduling, it becomes:

$$
\begin{array}{rcl}
\texttt{node}\ \mathit{counting}\,(\mathit{tick} : \mathit{bool}, \mathit{top} : \mathit{bool})\ \texttt{with}\,(o : \mathit{int})\ \texttt{in}\,(v : \mathit{int}) & & \\
v & = & (\texttt{merge}\ \mathit{tick}\ (\texttt{true} \to (1^{b}\ \texttt{when}\ \texttt{true}(\mathit{tick}))^{ck_3}) \\
 & & \quad (\texttt{false} \to (0^{b}\ \texttt{when}\ \texttt{false}(\mathit{tick}))^{ck_4}))^{b} \\
\texttt{and}\ o & = & (\texttt{merge}\ \mathit{top}\ (\texttt{true} \to (v^{b}\ \texttt{when}\ \texttt{true}(\mathit{top}))^{ck_1}) \\
 & & \quad (\texttt{false} \to ((t^{b} + v^{b})^{b}\ \texttt{when}\ \texttt{false}(\mathit{top}))^{ck_2}))^{b} \\
\texttt{and}\ t & = & (0\ \texttt{fby}\ o^{b})^{b}
\end{array}
$$

$$
\begin{aligned}
ck_1 &= b\ \texttt{on}\ \texttt{true}(\mathit{top}) \\
ck_2 &= b\ \texttt{on}\ \texttt{false}(\mathit{top}) \\
ck_3 &= b\ \texttt{on}\ \texttt{true}(\mathit{tick}) \\
ck_4 &= b\ \texttt{on}\ \texttt{false}(\mathit{tick})
\end{aligned}
$$

# Static scheduling and copy variables

The normalisation process may have to introduce extra copy variables; otherwise, equations may not be statically schedulable. E.g.,:

$$\begin{array}{rcl} x & = & 0\ \texttt{fby}\ y \\ \texttt{and}\ \ y & = & 1\ \texttt{fby}\ x \end{array} \qquad \begin{array}{rcl} cx & = & x \\ \texttt{and}\ \ x & = & 0\ \texttt{fby}\ y \\ \texttt{and}\ \ y & = & 1\ \texttt{fby}\ cx \end{array}$$

- Both are in normal form but the left sequence is not schedulable.
- Either do a clever normalisation a priori;
- or systematically add a copy for the registers to get an equation of the form: `m = v fby x`

Property: if the equation defining `x` is scheduled after any read of `m`, then `x` and `m` can be stored in the same location.

By characterising well scheduled equations only, we give the liberty to the compiler to consider several possible implementations of the scheduling/normalization functions.

# Translation to Sequential Code

We introduce an intermediate target imperative language in which annotated normalized data-flow programs are compiled.

## What do we need?

- represent transition functions in an imperative style
- a simple memory model: static allocation of memory; no aliasing.
- such that the translation into C code is simple.

## Intuition

A synchronous function $f$ defines a "class" with

- a set of state variables and a set of instance variables;
- a set of methods that read/write these state variables.

The memory model is a tree and there is no aliasing between states. The method of a class can only modify its own states ("instance variables").

# A Simplification

For MiniLS, we only need to produce a class with two methods `step` and `reset`:

- Given the current inputs, the method `step` produces the current outputs and modifies in place its internal state.
- A method `reset` initialize/reset its internal state.

Yet, the general case with several methods is useful. E.g.,:

- provides set/get methods to have direct access to inputs and output in order to reduce the number of copies;
- to implement the modular static scheduling problem;
- to do program specialisation, e.g., implement a special faster method for particular values of the inputs.
- to implement the language extended with ODEs (see related course).

# The Obc Intermediate Language

$$
\begin{array}{lll}
md & ::= & \texttt{let}\, x = c \mid md; md \\
 & & \texttt{let}\, f = \texttt{class}\langle M,\ I,\ (method_i(p_i) = q_i\ \texttt{where}\ S_i)_{i\in[1..n]}\rangle \\
p, q & ::= & x : bt, ..., x : bt \\
M & ::= & [x : bt[= v]; \ldots; x : bt[= v]] \\
I & ::= & [o : f; \ldots; o : f]
\end{array}
$$

$$
\begin{array}{lll}
c & ::= & v \mid lv \mid \ \mid op(c, \ldots, c) \mid o.method(c, \ldots, c) \\
S & ::= & () \mid lv \leftarrow e \mid S\ ;\ S \mid \texttt{var}\ x : bt\ \texttt{in}\ S \mid \texttt{if}\ c\ \texttt{then}\ S\ \texttt{else}\ S \\
 & & \quad \mid \texttt{case}\ (x)\ \{C : S; ...; C : S\} \\
R, L & ::= & S; \ldots; S \\
lv & ::= & x \mid \texttt{state}\,(x) \\
method & ::= & step \mid reset \mid ...
\end{array}
$$

# Principles of the translation

- Hierarchical memory model which corresponds to the call graph: one instance variable per function call;
- Control-structure (invariant): an equation annotated with clock $ck$ is executed when $ck$ is true.
- A guarded equations $x = e^{ck}$ translates into a control-structure. E.g., the equation:

$$x = (y + 2)^{\texttt{base on } C_1(x_1) \texttt{ on } C_2(x_2)}$$

  is translated into a piece of control-structure:

$$\texttt{case}\ (x_1)\ \{C_1 : \texttt{case}\ (x_2)\ \{C_2 : x = y + 2\}\}$$

- The translation is made by a linear traversal of the sequence of normalized/scheduled equations.

- local generation of a control-structure from a clock:

$$\begin{array}{lcl} \mathit{Control}(\mathtt{base}, S) & = & S \\ \mathit{Control}(ck \ \mathtt{on}\ C(x), S) & = & \mathit{Control}(ck, \mathtt{case}\ (x)\ \{C : S\}) \end{array}$$

- merge them locally

$$\begin{array}{l} \mathit{Join}(\mathtt{case}\ (x)\ \{C_1 : S_1; ...; C_n : S_n\}, \mathtt{case}\ (x)\ \{C_1 : S'_1; ...; C_n : S'_n\}) \\ \quad = \mathtt{case}\ (x)\ \{C_1 : \mathit{Join}(S_1, S'_1); ...; C_n : \mathit{Join}(S_n, S'_n)\} \\ \mathit{Join}(S_1, S_2) = S_1; S_2 \end{array}$$

## Control-optimization:

- The scheduling function must put equations with the same clock or one that is a sub-clock of the other close to each others
- $ck$ on $C(x)$ is a subclock of $ck'$ if $ck$ is a subclock of $ck'$ or $ck = ck'$.

# Example

```
class counting =
  memory t1 : int = 0;

  reset () =  state(t1) := 0;

  step(tick:bool,top:bool) returns (o:int) =
    v:int, t2:int in
    case (tick) {
      | True: v := 1;
      | False: v := 0; };
    case(top) {
      | True: o := v;
      | False: o := state(t1) + v; };
    t2 := o;
    state(t1) := t2;
```

# Example (modularity)

- A node definition is compiled separately, once for all.
- A node that calls an other node need a memory to store its internal state.

## Example:

```
node sum(x:int) returns (o:int) with
  o = 0 fby o + x;

node condact(c:bool;input:int) returns (o:int) with
var o':int in
  o = merge c (true -> o')
              (false -> (0 fby o) when false(c)) and
  o' = sum(input when true(c))
```

# Target code:

```
class condact =
  memory x_2 : int = 0
  instances x_4 : sum

  reset() =
    x_4.reset();
    state x_2 := 0;

  step(c : bool; input : int) returns (o : int)
    var o' : int in
    case (c) {
      case true :
        o' := x_4.step(input);
        o := o';
      case false :
        o := state(x_2);
    };
    state x_2 := o; }
```

# Démo
# Velus, Heptagon [5] and Scade 6

`https://velus.inria.fr/tryvelus/index.html`

[5] Available at `https://gitlab.inria.fr/synchrone/heptagon`, with source code in OCaml.

# Notations

- If $p = [x_1 : bt_1; ...; x_n : bt_n]$ and $p_2 = [x'_1 : bt'_1; ...; x'_k : bt'_k]$ then $p_1 + p_2 = [x_1 : bt_1; ...; x_n : bt_n; x'_1 : bt'_1; ...; x'_k : bt'_k]$ provided $x_i \neq x'_j$ for all $i$, $j$ such that $1 \leq i \leq n$, $1 \leq j \leq k$.
- $[\,]$ denotes the empty list of variable declarations.
- $m_1$ and $m_2$ denotes environments for memories.
- $j_1$ and $j_2$ denotes environments for instances.
- $m_1 + m_2$ for the composition of two substitutions on memory names and $j_1 + j_2$ on object instances.
- $S \cdot L$ is a list of instructions whose head is $S$ and tail is $L$. [] is the empty list and $[S_1; ...; S_n] = S1 \cdot (... \cdot S_n \cdot [])$.

# Translation functions

Applied on normalised/scheduled expressions and equations.

- $TE_m\,(a)$ translates an expression $a$.
- $TCA_m\,(y, ca)$ translates an expression $ca$ with result stored in $y$.
- $TEq_{\langle m,S,j,L\rangle}\,(eq)$ defines the translation of an equation:
  - $m$ is a memory environment;
  - $S$ is executed when reset;
  - $j$ is the instance environment;
  - $L$ is a sequence of instructions
- $TEList_m\,[a_1; ...; a_n]$ translates a list of expressions.
- $TEqList([eq_1; ...; eq_n])$ translates a list of equations.

## Expressions

$$
\begin{array}{lcl}
TE_m\,(e^{ck}_{bt}) & = & TE_m\,(e) \\
TE_m\,(v) & = & v \\
TE_{m+[x:bt=v]}\,(x) & = & \texttt{state}\,(x) \\
TE_m\,(x) & = & x \text{ otherwise} \\
TE_m\,(a \texttt{ when } C(x)) & = & TE_m\,(a) \\
TE_m\,(op(a_1, ..., a_n)) & = & let\,[c_1; ...; c_n] = TEList_m\,[a_1; ...; a_n]\,in \\
 & & op(c_1, ..., c_n)
\end{array}
$$

## Controled expressions

$$
\begin{array}{lcl}
TCA_m\,(y, \texttt{merge } x\,(C \vec{\rightarrow} ca)^{ck}_{bt}) & = & \texttt{case}\,(x)\,\{C : \vec{TCA}_m\,(y, ca)\} \\
TCA_m\,(y, a) & = & y := TE_m\,(a) \text{ otherwise}
\end{array}
$$

## Equations

$$
\begin{array}{lcl}
TEList_m\,[a_1; \ldots; a_n] & = & [TE_m\,(a_1); \ldots; TE_m\,(a_n)] \\
TEqList(eq) & = & TEq_{\langle [],\mathtt{skip},[],[]\rangle}\,(eq) \\
TEqList([eq_1; \ldots; eq_n]) & = & TEq_{TEqList([eq_2;\ldots;eq_n])}\,(eq_1)
\end{array}
$$

## Instantaneous equations, synchronous register

$$TEq_{\langle m,S,j,L\rangle}\,(x = e_{bt}^{ck}) = \langle m, j, S, (Control(ck, TCA_m\,(x, e_{bt}^{ck}))) \cdot L\rangle$$

$$
\begin{array}{l}
TEq_{\langle m,S,j,L\rangle}\,(y = (v\ \mathtt{fby}\ a)_{bt}^{ck}) = \\
\quad let\ m' = m + [y : bt = v]\ in \\
\quad let\ c = TE_{m'}\,(a)\ in \\
\quad \langle m', \mathtt{state}\,(y)\ :=\ v; S, j, (Control(ck, \mathtt{state}\,(y)\ :=\ c)) \cdot L\rangle
\end{array}
$$

## Application of a node

$$
\begin{aligned}
&TEq_{\langle m,S,j,L\rangle}\left((x_1,...,x_k) = f\,(a_1,...,a_n)\ \texttt{every}\ e_0{}^{ck}_{bt}\right) = \\
&\quad let\ c_0 = TE_m\,(e_0{}^{ck}_{bt})\ in \\
&\quad let\ [c_1,...,c_n] = TEList_m\,[a_1;...;a_n]\ in \\
&\quad \langle m, o.\texttt{reset}; S, [o : f] + j \\
&\quad (Control(ck, \texttt{case}\ (c_0)\ \{(\texttt{true} : o.\texttt{reset})\}))\cdot \\
&\quad (Control(ck, (x_1,...,x_k) = o.\texttt{step}\,(c_1,...,c_n)))\cdot L\rangle \text{ where } o \notin Dom(j)
\end{aligned}
$$

# Translation of a node definition

$$
\begin{array}{l}
TP\,(\texttt{node}\,f\,(p)\,\texttt{returns}\,(q)\,\texttt{var}\,r\,\texttt{in}\,eq_1\,\texttt{and}\,\ldots eq_n) = \\
\quad \mathit{let}\ \langle m, S, j, L\rangle = \mathit{TEqList}([eq_1; \ldots; eq_n])\ \mathit{in} \\
\quad \texttt{class}\ f = \ \langle \texttt{memory} = m; \\
\qquad\qquad\qquad\qquad \texttt{instances} = j; \\
\qquad\qquad\qquad\qquad \texttt{reset} = S; \\
\qquad\qquad\qquad\qquad \texttt{step}(p)\ \texttt{returns}(q)\ \texttt{var}\ r\ \texttt{in}\ \mathit{JoinList}(L)\rangle \\
\text{where}\ \mathit{SCH}([eq_1; \ldots; eq_n])
\end{array}
$$

We write $\mathit{SCH}([eq_1; \ldots; eq_n])$ when there exists $r, w, m$ such that $\mathit{SCH}([eq_1; \ldots; eq_n]) : r, w, m$.

# From Obc to a target language

The translation from Obc to C is straightforward. Yet, the proof of correctness is not [BBD+17].

## Principle

- For a class $f$, define a C structure that stores the internal state of $f$.
- Every method $m$ of $f$ becomes a function $f_m$ with an extra input *self* that point to its internal state.
- When the "step" method has several outputs, returns a C structure on the stack or, better:
- define a C structure to store the output; the method takes an extra pointer to it.
- or add extra methods, with no output, to get the value of each output. They are implemented in C by direct access to the state of the callee.

# Démo
# Velus, Heptagon [6] and Scade 6

`https://velus.inria.fr/tryvelus/index.html`

[6] Available at `https://gitlab.inria.fr/synchrone/heptagon`, with source code in OCaml.

# Optimizations

- Some optimizations can be done by the compiler of the target language. E.g., it is useless to optimize reuse between local (stack) variables.
- But this depends on the quality of the compiler of the target language.
- Some classical optimizations (CSE, copy and const. prop., inlining) can be applied directly on the data-flow representation.
- It is always useful to reduce the number of state variables and the liveness between reads and writes.
- Optimize the control structure. E.g., gather `if/then/else`.
- These two optimizations depend on the scheduling heuristic.

# Optimization that a C compiler cannot do easily

- Avoid copies: `x` and `pre x` can be shared when all equations reading `pre x` can be scheduled before the equation `x = ...` E.g.:
  `x = pre x + 1` can be compiled into `x := x + 1`
- Automata minimization (generalization of CSE). E.g.,
  `y = 0 fby y + 1 and z = 0 fby z + 1` as
  `y = 0 fby y + 1 and z = y`.
- share memories between two pieces of code never active in parallel and when going from one to the other is reset on entry.
- for array iterators, perform loop fusion and generate in place modification for functional updates. [GGPP12]

Those optimisations are implemented in the Heptagon compiler.

# Control Optimization

## Share/reduce the number of control-structures

- some piece of code is only executed at the very first instant or only when some condition is true. E.g.,

```
if state(i) then x = 0; /* initialization code */
...
if state(i) { y = 1;} /* initialization code */
...
if c then x = m -> pre_1 + 1;
    /* step when clock c_1 is true */
...
if c then m -> pre_1 = x;
    /* set the memory when clock c_1 is true */
```

- minimize the number of `if/then/else` to open. For that, define a scheduling function which gather equations activated on the same clock (cf. *Join*(., .)).

Still, the generation is not that efficient.

# Compilation into Automata

# Compilation into Automata

Generating a single step functions means that some conditions that are surely false will be executed at every step. E.g., consider the way an initialization `o = x -> y` is compiled. In Obc.

```
if state(init_1) then o := x else o := y;
...;
state(init_1) := false;
```

Generates an "optimal" control structure which only execute the necessary code at every instant.

This is the idea of compilation into automata introduced by Halbwachs and Plaice (Lustre V2).

It was improved to generate a minimal automaton (Lustre V3) by Ratel et Raymond in 1991 [HRR91].

It is implemented in the academic Lustre compiler.

# An example (in Lustre syntax)

```
node counter(tick,top:bool) returns (cpt:int)
 var i:int;
 let cpt = 0 -> if pre top then i
                else if tick then pre cpt + 1
                else pre cpt;
     i = if tick then 1 else 0;
 tel;
```

After normalization and scheduling, we get:

```
node counter(tick,top:bool) returns (cpt:int)
 var i:int;
 let i = if tick then 1 else 0;
     cpt = if init then 0
           else if ptop then i
           else if tick then pcpt + 1
           else ptop;
     ptop = pre top;
     pcpt = pre cpt;
     init = true fby false
 tel;
```

# Single loop code

```
if tick then i := 1 else i := 0;
if state(init) = 0 then cpt := 0
else if state(ptop) then cpt := i
else if tick then cpt := state(pcpt) + 1
     else cpt := state(pcpt)
state(init) := false;
state(ptop) := top;
state(pcpt) := cpt
```

`top` and `state(ptop)` can be stored at the same location; the same with `cpt` and `state(pcpt)`. Then, the last two assignment can be removed (see remark on slide 36).

# Example

Initial state: $S_1 = [true/init]$

The code that computes the output is:

```
if tick then i := 1 else i := 0;
cpt := 0;
state(pcpt) := cpt
```

- It can be simplified into `state(pcpt) := 0;`
- The code that computes the next state is:

```
if top then state(ns) := S2 else state(ns) := S3;
```

State $S_2$: $S_2 = [false/init, true/ptop]$

```
if tick then i := 1 else i := 0;
state(pcpt) := i;
if top then state(ns) := S2; else state(ns) := S3;
```

State $S_3$: $S_3 = [false/init, false/ptop]$

```
if tick then
  { i := 1;
    state(pcpt) := state(pcpt) + 1; }
else
    i := 0;
if top then state(ns) := S2
else state(ns) := S3;
```

# The final automaton

```
case state(ns){
  S1: state(pcpt) := 0;
      if top state(ns) := S2; else state(ns) := S3;
  S2: if tick then i := 1 else i := 0;
      state(pcpt) := i;
      if top then state(ns) := S2 else state(ns) := S3;
  S3: if tick then state(pcpt) := state(pcpt) + 1;
      if top state(ns) := S2 else state(ns) := S3;
}
```

# Conclusion

- far better code but the size has increased
- assertions (i.e., `assert` $P$ in **Lustre**) can be taken into account during the enumeration

## Problems

- combinatorial explosion
- in **Lustre**, the control-structure is hidden and encoded with booleans (think of a one-hot encoding of an automaton)
- which boolean variables should we consider? There is no *good* programming rules to avoid this explosion
- limit to a predefined set of boolean variables (e.g., clocks).

## Solutions?

- automata minimization done a posteriori.
- direct generation of a minimal automaton (called "compilation on demand", [Halbwachs, Ratel, Raymond, PLILP 91])

# An example (Halbwachs et al [HRR91])

```
node Example(i: bool) return (n: int);
var x,y,z : bool;
let
   n = 0 -> if (pre x) then 0 else (pre n) + 1;
   x = false -> not (pre x) and z;
   y = i -> if (pre x) then (pre y) and i
            else (pre z) or i;
   z = true -> if (pre x) then (pre z)
               else ((pre y) and (pre z)) or i);
tel
```

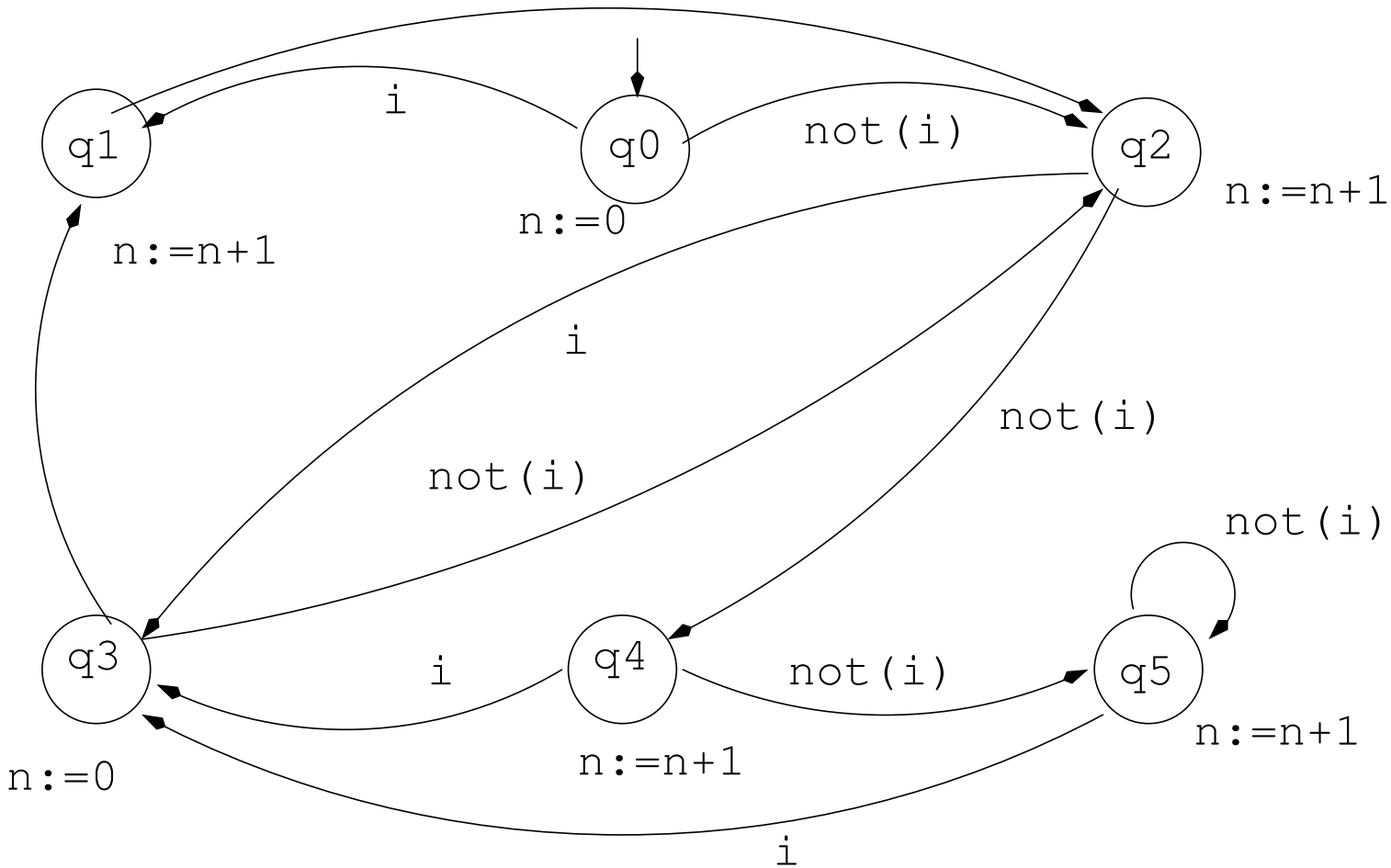
4 state variables (`->`, `pre x`, `pre y` and `pre z`)

# Example

## Example:

- $q_0 : (init, pre\_x, pre\_y, pre\_z) = (1, nil, nil, nil)$
  Action: `n:=0`
  $next_{init}(q_0, i) = 0$
  $next_{pre\_x}(q_0, i) = 0$
  $next_{pre\_y}(q_0, i) = i$
  $next_{pre\_z}(q_0, i) = 0$
  `if (`$i$`) { state =` $q_1$`;} else { state =` $q_2$`;}`
  with $q_1 = (0, 0, 1, 1)$ et $q_2 = (0, 0, 0, 1)$
- $q_1 = (0, 0, 1, 1)$
  Action: `n:=n+1`
  $next_{init}(q_1, i) = 0$
  $next_{pre\_x}(q_1, i) = 1$
  $next_{pre\_y}(q_1, i) = 1$
  $next_{pre\_z}(q_1, i) = 1$
  `state =` $q_3$
- $q_2 = (0, 0, 0, 1)$
  Action: `n:=n+1, if (`$i$`) { state =` $q_3$`;} else { state =` $q_4$`;}`

- $q_3 = (0, 1, 1, 1)$
  Action: `n:=0, if (`$i$`) {state = `$q_1$`;} else { state = `$q_2$`;}`
- $q_4 = (0, 0, 1, 0)$
  Action: `n:=n+1, if (`$i$`) { state = `$q_3$`;} else { state = `$q_5$`;}`
- $q_5 = (0, 0, 0, 0)$
  Action: `n:=n+1, if (`$i$`) { state = `$q_3$`;} else { state = `$q_5$`;}`



# Compilation into automata "on demand"

- the automaton is not minimal: $q_0$ and $q_3$ are equivalent; $q_4$, $q_5$ and $q_2$ are equivalent
- we can minimize *a posteriori* (**Lustre** V2) but still an explosion of the number of states in the intermediate automaton

Solution: directly generate a minimal automaton. In practice, the code is still too big, but:

- this technique say something very interesting when the main node has a single boolean variable
- what is the minimal automaton for a node with a single boolean variable which is always true? The trivial automaton `true`!
- this corresponds exactly to proving a safety (invariant) property by a Model Checking technique.

# Conclusion

- The compilation into automata is possible but it is not modular and tend to generate enormous code.
- The compiler of **Scade** generate single-loop code.
- The clock-directed translation method is a good compromise (simple et reasonnably efficient code).
- If the input language has automata, like **Scade 6**, the result of compilation into automata could be expressed as a source-to-source transformation.
- Would-it simplify the proof of its correctness?

# References I


Timothy Bourke, Lélio Brun, Pierre-Évariste Dagand, Xavier Leroy, Marc Pouzet, and Lionel Rieg.

A Formally Verified Compiler for Lustre.

In *International Conference on Programming Language, Design and Implementation (PLDI)*, Barcelona, Spain, June 19-21 2017. ACM.

Darek Biernacki, Jean-Louis Colaco, Grégoire Hamon, and Marc Pouzet.

Clock-directed Modular Code Generation of Synchronous Data-flow Languages.

In *ACM International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, Tucson, Arizona, June 2008.

Sylvain Boulmé and Grégoire Hamon.

Certifying Synchrony for Free.

In *International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR)*, volume 2250, La Havana, Cuba, December 2001. Lecture Notes in Artificial Intelligence, Springer-Verlag.

Short version of *A clocked denotational semantics for Lucid-Synchrone in Coq*, available as a Technical Report (LIP6), at `www.di.ens.fr/~pouzet/bib/bib.html`.

Paul Caspi and Marc Pouzet.

Synchronous Kahn Networks.

In *ACM SIGPLAN International Conference on Functional Programming (ICFP)*, Philadelphia, Pensylvania, May 1996.

Paul Caspi and Marc Pouzet.

A Co-iterative Characterization of Synchronous Stream Functions.

In *Coalgebraic Methods in Computer Science (CMCS'98)*, Electronic Notes in Theoretical Computer Science, March 1998.

Extended version available as a VERIMAG tech. report no. 97–07 at `www.di.ens.fr/~pouzet/bib/bib.html`.

Jean-Louis Colaço and Marc Pouzet.

Clocks as First Class Abstract Types.

In *Third International Conference on Embedded Software (EMSOFT'03)*, Philadelphia, Pennsylvania, USA, october 2003.

# References II


Léonard Gérard, Adrien Guatto, Cédric Pasteur, and Marc Pouzet.

A Modular Memory Optimization for Synchronous Data-Flow Languages. Application to Arrays in a Lustre Compiler.
In *Languages, Compilers and Tools for Embedded Systems (LCTES'12)*, Beijing, June 12-13 2012. ACM.
Best paper award.

N. Halbwachs, P. Raymond, and C. Ratel.

Generating efficient code from data-flow programs.
In *Third International Symposium on Programming Language Implementation and Logic Programming*, Passau (Germany), August 1991.

R. Lublinerman, C. Szegedy, and S. Tripakis.

Modular Code Generation from Synchronous Block Diagrams — Modularity vs. Code Size.
In *ACM Principles of Programming Languages (POPL)*, 2009.

John Plaice.

*Sémantique et Compilation de LUSTRE, un langage à fiux de données temps-réel*.
PhD thesis, INP Grenoble, 1988.

Marc Pouzet.

*Lucid Synchrone, version 3. Tutorial and reference manual*.
Université Paris-Sud, LRI, April 2006.
Distribution available at: `https://www.di.ens.fr/~pouzet/lucid-synchrone/`.

Marc Pouzet and Pascal Raymond.

Modular Static Scheduling of Synchronous Data-flow Networks: An efficient symbolic representation.
In *ACM International Conference on Embedded Software (EMSOFT'09)*, Grenoble, France, October 2009.

# References III


Marc Pouzet and Pascal Raymond.

Modular Static Scheduling of Synchronous Data-flow Networks: An efficient symbolic representation.
*Journal of Design Automation for Embedded Systems*, 3(14):165–192, 2010.
Special issue of selected papers from `http://esweek09.inrialpes.fr/`Embedded System Week. Extended version of [PR09].

Pascal Raymond.

Compilation séparée de programmes Lustre.
Technical report, Projet SPECTRE, IMAG, July 1988.

Pascal Raymond.

*Compilation efficace d'un langage déclaratif synchrone : Le générateur de code Lustre-V3*.
Thèse, INPG, Grenoble, France, november 1991.

Pascal Raymond.

*Compilation efficace d'un langage déclaratif synchrone: le générateur de code Lustre-v3*.
PhD thesis, Institut National Polytechnique de Grenoble, 1991.