Sequential Code Generation of Lustre

Marc Pouzet

École normale supérieure
Marc.Pouzet@ens.fr

MPRI
September 19, 2017

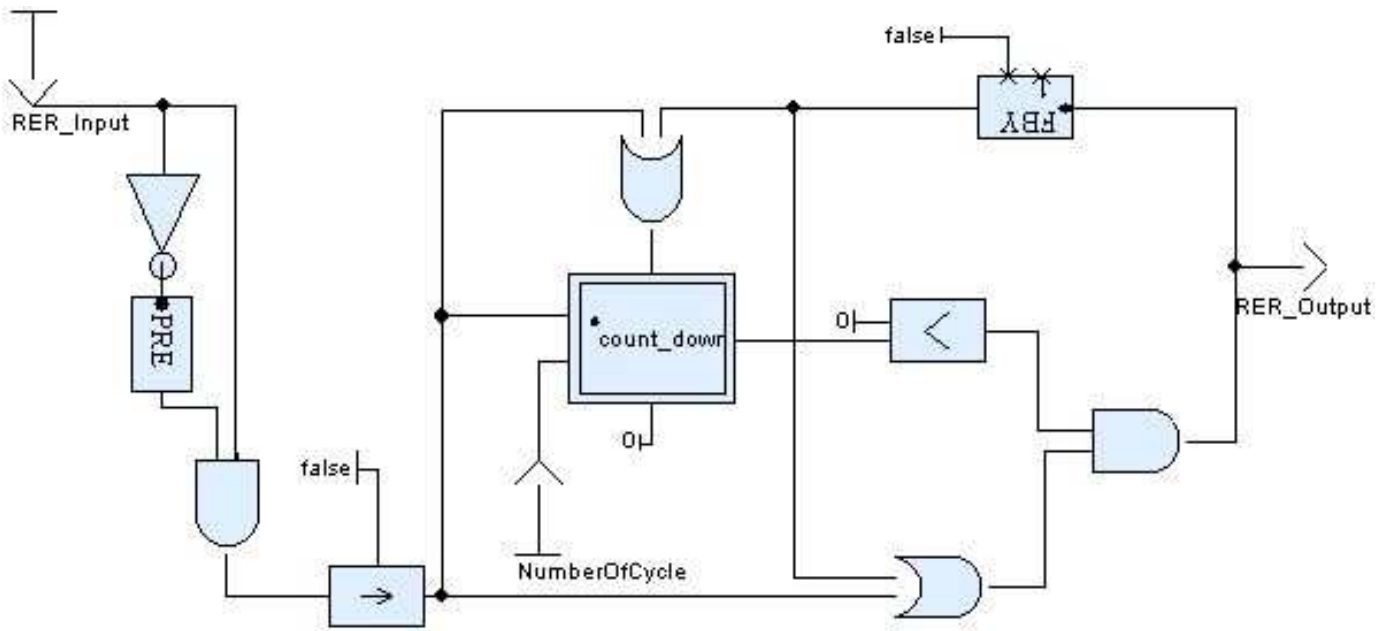
Sequential Code Generation

Input: A **parallel** data-flow networks made of synchronous operators; not necessarily **Lustre** (e.g., the discrete subset of **Simulink**)

Output: A sequential function written in an imperative language (e.g., C, Java) to compute one step of the data-flow network:

parallelism is removed by the compiler

Example: the `risingEdgeRetrigger` of **Scade**



Sequential Code Generation

A stream function $f : \mathit{Stream}(T) \to \mathit{Stream}(T')$ is compiled into a pair:

- an initial state and a transition function: $\langle s_0 : S, f_t : S \times T \to T' \times S \rangle$

A stream equation $y = f(x)$ defining $y = (y_n)_{n \in \mathbb{N}}$ is computed sequentially by $y_n, s_{n+1} = f_t(s_n, x_n)$

Synchrony ensures that a stream of type $\mathit{Stream}(T)$ is implemented by a scalar value of type T.

Remarks

- The transition function can be split in two:
  - an initial state: $s_0 : S$
  - a value function: $f_v : S \times T \to T'$
  - a state modification ("commit") function: $f_s : S \times T \to S'$
- The state is modified **in place** instead of being returned.

Two typical implementations

Periodic sampling

```
  s := s0;
  every clock tick
   read input e;
   let o, s' = ft s e in
   s := s';
   emit output o
end
```

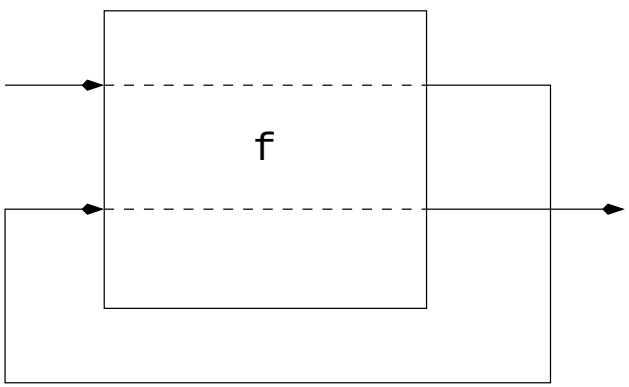
Event driven

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

Modular Code Generation

- produce a sequential function for each definition;
- compose them to obtain the final transition function;
- follow data-dependences;

Nonetheless, modular sequential code generation is not always possible, even when there is no causality loop [Gonthier, 1988].




```
node copy(a, b:bool) returns (c, d:bool)
  let
     c = a;
     d = b;
  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.

Two possible sequential code for f

Since the equations `c = f1(a)` and `d = f2(b)` are independent, they can be scheduled in any order. E.g.,

```
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. Thus, modular static scheduling generating a **single** transition function is not possible.

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); no restriction on causality loops
- the way it is done in the academic **Lustre** compiler (VERIMAG)
- efficient enumeration techniques can be applied to generate finite state automata [Raymond PhD. Thesis[**?**], Halbwachs et al. [**?**]]
- the code is very efficient but code size may explode
- it is hard to find the "good" boolean variables to enumerate to get efficient (in both time and space) code

Single Loop Code Generation ("black boxing")

- a **single code** repeated infinitely
- modular; the way it is done for **Scade**. Iimposes **stronger causality constraints**: every loop should cross an explicit delay
- this simplifies **tracability** issues
- static expansion and agressive optimisations still possible.

Modular Static Scheduling

An intermediate solution ("grey boxing")

- Instead of producing a single step function, produce several. E.g., for `copy`, two.
- This is called the **Modular Static Scheduling** problem.
- Identified in 1988 by Raymond who proposed a first algorithm [**?**]
- The **Optimal Modular Static Scheduling** problem is when the number of step functions is minimal.
- Several solutions has been proposed. See extra course.

Sequential code generation

A small reference compiler for Lustre [?]

The architecture is that of the Lucid Synchrone compiler and SCADE 6.

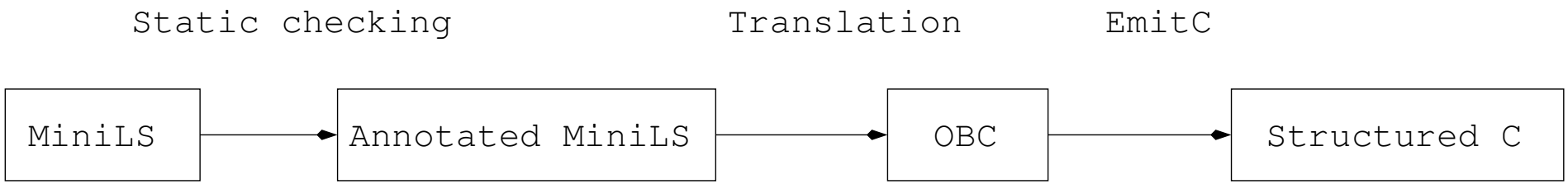
MiniLS

- A minimalistic **clocked data-flow language** as the input.
- General enough to be used as a target language for **Lustre**.
- General enough to be a target for more advanced language like SCADE 6.

Objective

- Sequential code generation with one step function per node definition.
- Write a reference compiler mainly with source-to-source transformation with everything "traced".
- Compilation into an intermediate "object based" language to represent transition functions.
- Translated then to imperative code (e.g., structured C, Java).

Organization of the Compiler



The source language

Consider a Lustre-like language. Translate it into imperative sequential code.

$$
\begin{array}{lcl}
a & ::= & v \mid x \mid v \texttt{ fby } a \\
 & & \mid op\,(a, \ldots, a) \\
 & & \mid f\,(a, \ldots, a) \texttt{ every } a \\
 & & \mid a \texttt{ when } C(x) \\
 & & \mid \texttt{merge } x\ (C \to a)\ \ldots\ (C \to a) \\
D & ::= & pat = a \mid D \texttt{ and } D \\
pat & ::= & x \mid (pat, \ldots, pat) \\
 & & \\
d & ::= & \texttt{node } f(p) = p \texttt{ with var } p \texttt{ in } D \\
p & ::= & x : bt; \ldots; x : bt \\
v & ::= & C \mid i
\end{array}
$$

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 $(\texttt{true} \rightarrow z)$ $(\texttt{false} \rightarrow t)$	x_0	y_1	x_2	y_3	...

- v `fby` x is the unit delay initialized with v ($1/z$ of Simulink)
- z is at a slower rate than x.
- the `merge` constructs combines two complementary sequences

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:

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

$$\texttt{pre}\,(e) \quad = \quad \mathit{nil} \texttt{ fby } e$$

Simplification

Replace the n-ary merge by the one of **Lucid Synchrone** [1] [**?**] that apply to a boolean condition: `merge` c e_1 e_2.

[1] `www.di.ens.fr/~pouzet/lucid-synchrone`

Example (counter)

"Counts the number of tops between two ticks".

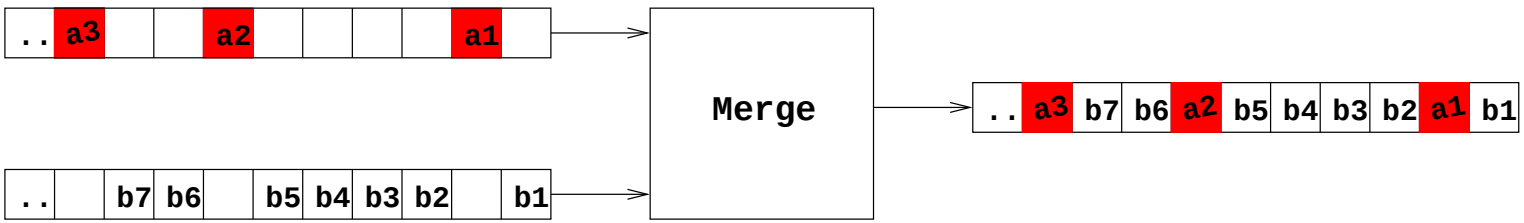
```
node counting (tick:bool; top:bool) = (o:int) with
var v: int in
    o = if tick then v else (0 fby o) + v
and v = if top then 1 else 0
```

tick	true	false	false	true	false	false	false	...
top	true	true	true	false	false	true	true	...
o	1	2	3	0	0	1	2	...
v	1	1	1	0	0	1	1	...

The n-ary merge operator

The merge operator:

- The deterministic `merge` 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:

$$t = C_1 \mid ... \mid C_n$$

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

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

Reseting a behavior

- There is no **reset** construct in Lustre: it must be manually defined

```
node counter() returns (s:int) with s = 0 fby s + 1

node resetable_counter(res:bool) returns (s:int) with
  s = if res then 0 else 0 fby s + 1;
```

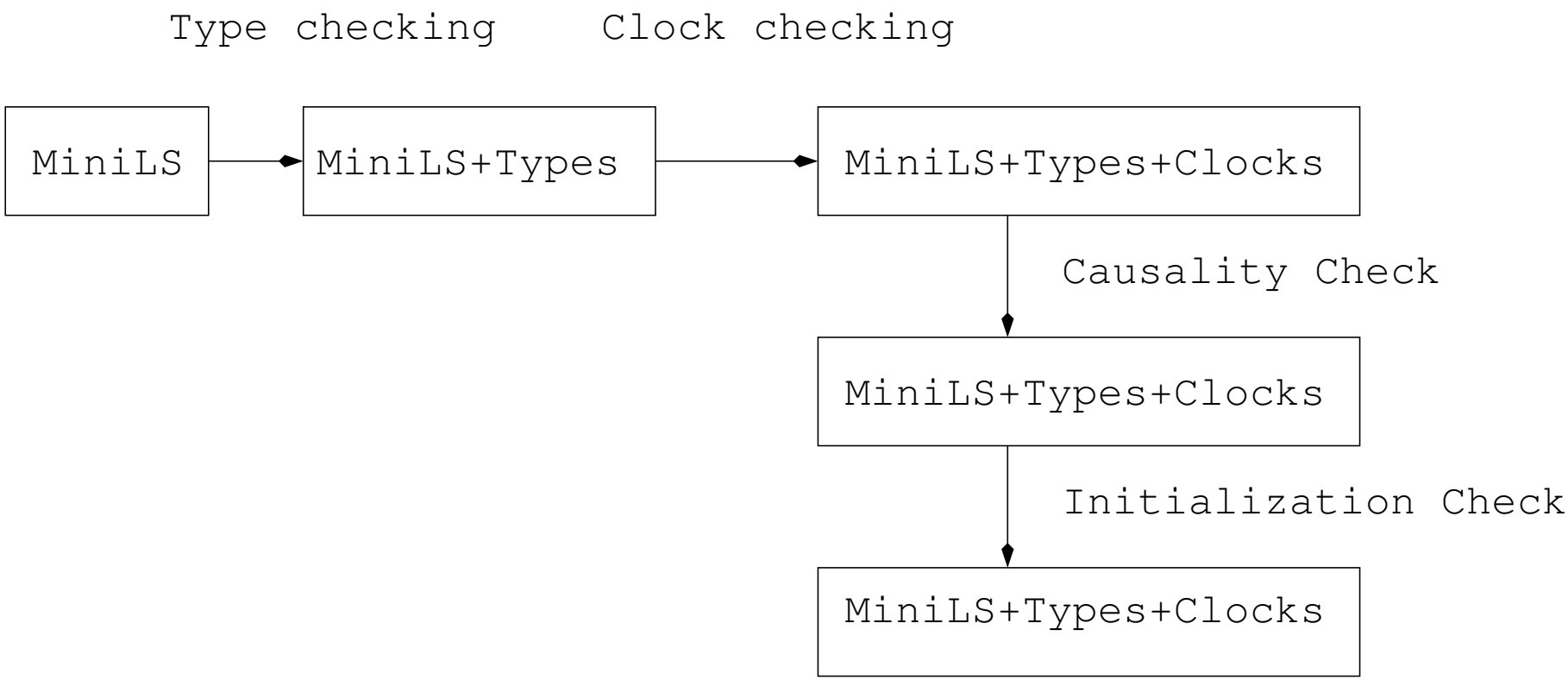
- making a component "resetable" is painful
- the generated code from the result is very bad whereas it is easy to generate code that efficienly reset a set of initialized registers.
- two good reasons to make it primitive in the language

Specific notation:

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

all the node instances used in the definition of node x are reseted when the boolean a is true

Static Checking



Type and Clock checking

An intermediate language where every expression is **annotated** with two informations: a type (bt) and a clock (ck)

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

Clock formula:

- The clock $clock(s)$ of a sequence s is a boolean sequence such that $clock(s) = true$ iff s is present
- Compute a correct approximation of it, i.e., a boolean formula ck

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

- For Lustre/MiniLS, clock formula are inferred and expressions are annotated.

Clock Checking

Type and clock checking are performed in order to annotate every expression with its type and clock. Clock checking is done structurally.

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

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

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

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

Clock Checking

For MiniLS, we consider the simpler case (than Lustre) where all input/output of a node are synchronous.

$$\frac{\mathcal{H} \vdash a_1 : ck \ldots \mathcal{H} \vdash a_n : ck \qquad \mathcal{H} \vdash a : ck}{\mathcal{H} \vdash f(a_1, \ldots, a_n)\,(a)\ \texttt{every}\ ck} \text{(Call)}$$

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

$$\vdash x_1 : t_1, \ldots, x_n : t_n : [x_1 : ck_1; \ldots; x_n : ck_n] \text{(Pat)}$$

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

Clock checking

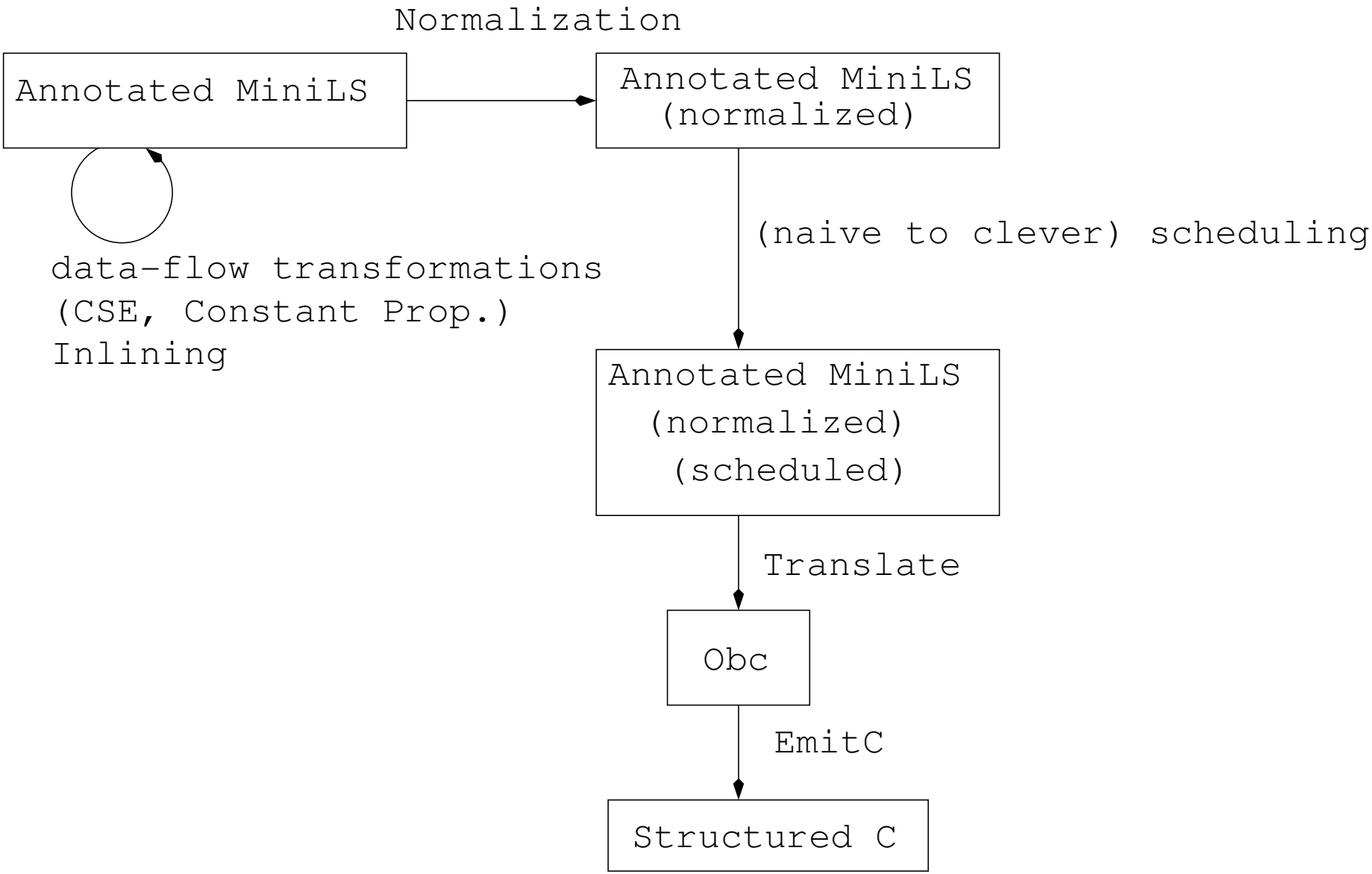
$$\frac{\mathcal{H} \vdash pat : ct \qquad \mathcal{H} \vdash a : ck}{H \vdash pat = a}\ (\textsc{Eq}) \qquad \frac{\mathcal{H} \vdash D_1 \qquad \mathcal{H} \vdash D_2}{\mathcal{H} \vdash D_1 \text{ and } D_2}\ (\textsc{And})$$

This system is very limited. E.g., functions return a single value, all inputs must be synchronous.

Extension

- The clock calculus of Lustre can be defined as a dependent type system that also work for higher-order [**?**, **?**]
- A simpler version reminiscent of the ML-type system with a limite form of existential types [**?**].
- Yet, we stick to this simpler version to illustrate the use of clocks in the compilation process.

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:

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

Alternative

Instead of a new intermediate language, characterize normal forms by a predicate.

$$NA(x_{bt}^{ck}) \qquad NA(v_{bt}^{ck}) \qquad \frac{NA(a)}{NA((a \texttt{ when } C(x))_{bt}^{ck})} \qquad \frac{NA(a_1) \ \ldots \ NA(a_n)}{NA((op(a_1, ..., a_n))_{bt}^{ck})}$$

$$\frac{NCA(a_1) \quad \ldots \quad NCA(a_n)}{NCA((\texttt{merge } x \ (C_1 \rightarrow a_1) \ ... \ (C_n \rightarrow a_n))_{bt}^{ck})} \qquad \frac{NA(a)}{NCA(a)}$$

$$\frac{NEQ(eq) \quad NEQ(D)}{NEQ(eq \texttt{ and } D)} \qquad \frac{NA(a_1) \quad \ldots \quad NA(a_n) \quad NA(a)}{NEQ((x_1, ..., x_m) = f \ (a_1, ..., a_n) \texttt{ every } a)}$$

$$\frac{NA(a)}{NEQ(x = (v \texttt{ fby } a)_{bt}^{ck})} \qquad \frac{NCA(a)}{NEQ(x = a)}$$

Checking the correctness of the normalization

The normalization is a relatively simple rewritting. It can be implemented in Ocaml. Then, it can be checked independently, e.g., by Coq.

- The only thing the normalization is allowed to do is to replace an expression by a name
- Given a list of equations, produce a new list of normalized equations and a substitution such that:

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

- A proof that the two equations:
  $\texttt{var}\ x_1 : t_1\ \texttt{in}\ ...\texttt{var}\ x_k : t_k\ \texttt{in}\ x_1 = e_1\ \texttt{and}\ ...x_k = e_k\ \texttt{and}\ eq'_1\ \texttt{and}\ ...eq'_n$ and $eq_1\ \texttt{and}\ ...eq_n$ are semantically equivalent.
- A **verification** test (e.g., implemented in Coq) such that:

$$subst([eq'_1; ...; eq'_m])[e_1/x_1, ..., e_l/x_l] = [eq_1; ...; eq_n]$$

  with:

$$subst(eqs)[e_1/x_1, ..., e_l/x_l] = subst(subst(eqs)[e_1/x_1])[e_2/x_2, ..., e_n/x_n]$$

- If the substitution succeeds, then the translation preserves the semantics.

Syntactic Dependences and Scheduling

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

- The scheduling function does not have to be proved. Define:

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

- 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)
- A proof that any shuffle of equations preserves the semantics (easy).
- A test function (e.g., in Coq) checks that the result of `schedule(eq_list)` is well formed; otherwise, the compilation stops.
- A test function (e.g., in Coq) checks that two equations lists are equal up-to a permutation.

Well Scheduled Equations

Two equations D_1 and D_2 are schedule-equivalent if they are equal up to permutation.

Definition (Well Scheduled Equations)

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

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

$$\frac{\{x_1, ..., x_m\} \cap (\cup_{0 \leq i \leq n} Left\,(a_i)) = \emptyset}{SCH(\vec{x} = f\,(\vec{a})\ \texttt{every}\ x) : (\cup_{0 \leq i \leq n} Left\,(a_i)) \cup \{x\}, \{x_1, ..., x_m\}, \emptyset}$$

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

Remark: By separating the scheduling function from what is a correctly scheduled sequence of equation, we can implement more clever scheduling functions.

Example (the counting node)

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

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

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

We write b as a short-cut for base.

Example (the counting node)

After the normalization, it becomes:

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

where :

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

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

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

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

Example (the counting node)

After the scheduling, it becomes:

$$
\begin{array}{rcl}
\multicolumn{3}{l}{\texttt{node}\ \mathit{counting}\,(\mathit{tick} : \mathit{bool}, \mathit{top} : \mathit{bool})\ \texttt{with}\,(o : \mathit{int})\ \texttt{in}\,(v : \mathit{int})} \\
v & = & (\texttt{merge}\ \mathit{top}\ (\texttt{true} \to (1^{b}\ \texttt{when}\ \texttt{true}(\mathit{top}))^{ck_3}) \\
 & & \qquad (\texttt{false} \to (0^{b}\ \texttt{when}\ \texttt{false}(\mathit{top}))^{ck_4}))^{b} \\
\texttt{and}\ o & = & (\texttt{merge}\ \mathit{tick}\ (\texttt{true} \to (v^{b}\ \texttt{when}\ \texttt{true}(\mathit{tick}))^{ck_1}) \\
 & & \qquad (\texttt{false} \to ((t^{b} + v^{b})^{b}\ \texttt{when}\ \texttt{false}(\mathit{tick}))^{ck_2}))^{b} \\
\texttt{and}\ t & = & (0\ \texttt{fby}\ o^{b})^{b}
\end{array}
$$

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

Static scheduling and copy variables

Even when a collection of equations is causally correct, it may be necessary to introduce auxiliary variables so as to schedule it.

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

- Each of them is in normal form but the left sequence is not.
- Either introduce an extra variable as shown on the right during normalization or scheduling,
- or eliminate it a posteriori.
- Introduce variables systematically is not good (i.e., a fresh name for e in equation $x = v \texttt{ fby } e$). It increases the number of copies.
- The scheduling heuristic tries to gather equations with the same clock and to store x and $v \texttt{ fby } x$ into the same location.
- By characterising only what is a normal and scheduled form, 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/pointer
- such that the translation into C code is (almost) trivial

Intuition

A synchronous function f defines a "class" or "machine" with

- A set of state variables and a set of instance variables (for "sub-machines").
- 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 consider the simple situation where we produce only two methods `step` and `reset`:

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

Obc is expressive enough to be used as a target for compilation methods that decompose the step function into several methods [Tripakis et al. POPL'09, Pouzet et al., EMSOFT'09].
See extra course.

The Obc Intermediate Language

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

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

Principles of the translation

- Hierarchical memory model which corresponds to the call graph: one local memory for each function call
- the translation is made on a linear traversal of the sequence of normalized and scheduled equations
- 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:

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

- 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** must put equations with the same clock close to each others
- This does not complicates the proof as it can be programmed in Ocaml provided equations are in **well-formed scheduled form**.

Example

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

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

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

Example (modularity)

Principle:

- Each function is compiled separately, once for all.
- A function call needs a local memory added to the caller.

Example:

```
node count(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' = count(input when true(c))
```

Target code:

```
machine condact =
  memory x_2 : int = 0
  instances x_4 : count

  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; }
```

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 [2]

A set of mutually recursive functions. It must be applied on expressions/equations in normal form (normalized and scheduled).

- $TE_m(a)$ translates an expression.
- $TCA_m(x, ca)$ defines the translation of an expression ca to store in variable x.
- $TEq_{\langle m,S,j,L\rangle}(eq)$ 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.
- $TEqList([D_1, ..., D_n])$ translates a list of equations.

[2] Original definitions in [**?**]. Rewritten here into a simpler but equivalent form.

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

$TCA_m\,(y, ca)$ defines the translation of an expression ca to store in variable y.

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

Translation of equations

$$
\begin{array}{lcl}
TEList_m\,[a_1, ..., a_n] & = & [TE_m\,(a_1), ..., TE_m\,(a_n)] \\
TEqList(eq) & = & TEq_{\langle [],\mathtt{skip},[],[]\rangle}\,(eq) \\
TEqList(eq\ \mathtt{and}\ D) & = & TEq_{TEqList(D)}\,(eq)
\end{array}
$$

Instantaneous equations and the unit delay

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

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

Translation of equations

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

Translation of a node definition

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

For short, we write $\mathit{SCH}(D)$ where there exists r, w, m such that $\mathit{SCH}(D) : r, w, m$.

$\mathit{JoinList}(L)$ concatenates instructions from L and fuse adjacent case statements.

The translation from Obc to C is simple and not explained here.

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

- Share memories (CSE), compile `...pre(x)... + ...pre(x)...` as `...m1... + ...m1...` with `x1 = pre(x)`.
- 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 = pre y + 1; z = pre z + 1` as `y = pre y + 1; z = y`.
- share memories between two pieces of code never active in parallel and when the transition between the two is by reset. E.g.:

```
automaton
 | Left ->  let rec nat = 0 -> pre nat + 1 in
            do o = nat + 1 until c then Right
 | Right -> let rec nat2 = 2 -> pre nat2 + 2 in
            do o = nat2 + 1 until c then Left
end
```

The two occurrences of `->` and `pre` can be shared.

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 m -> init_1 then x = 0; /* initialization code */
...
if m -> init_1 then x = 1; /* initialization code */
...
if c_1 then x = m -> pre_1 + 1;
    /* step when clock c_1 is true */
...
if c_1 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

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.

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

Can we do a more aggressive optimization that would lead to 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 [**?**].

An example

```
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, we get:

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

Single loop code

```
(* sequence producing outputs *)
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
```

All occurrences of `top` can be replaced by `ptop`, and `cpt` by `state(pcpt)`. Then, the last two assignment disappear.

The rule is this:

Try to schedule data-flow equations such that equation `y = v fby x` disappear, that is, all equations reading `y` are scheduled before the equation that define modifying `x`.

Compilation into automata

-O0:

- no control-structure = a single code executed infinitely

-O4 ?

- build an automaton by enumerating boolean state variables
- partial evaluation of the code for every value
- in a language that has automata at the language level, this optimisation could be done as a source-to-source transformation.
- this is not done this way, at the moment.

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

The final automaton

```
switch (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;
end;
```

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 worst than in **Esterel**
- 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

Solutions?

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

An example

```
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
```

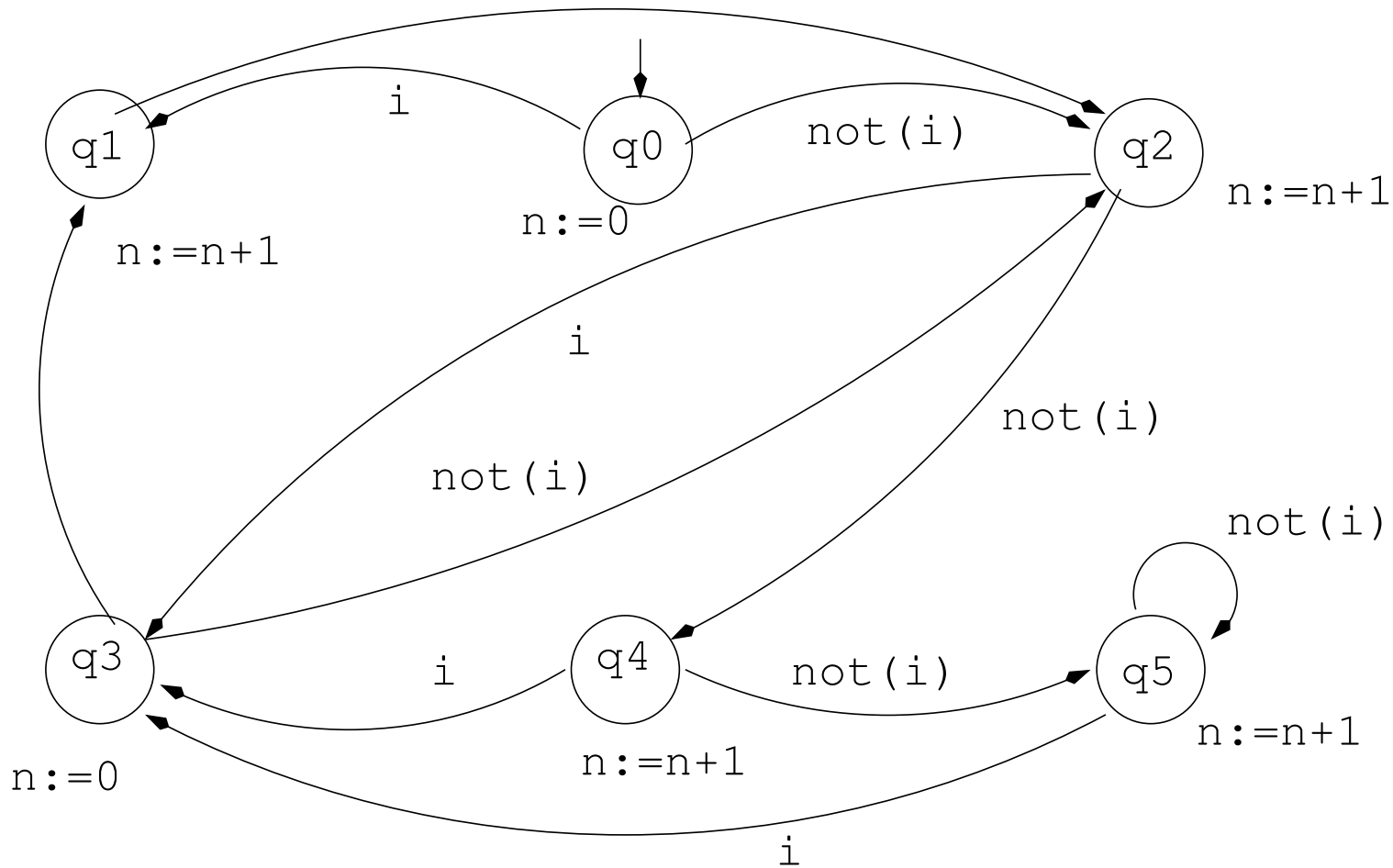
4 state variables (`->`, `pre x`, `pre y` and `pre z`)

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 will be very interesting when compiling a program having a single boolean variable
- what is the minimal automaton for a program 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.
- Industrial compiler generate single-loop code.
- Memory (buffer) and control optimization are important, in particular when the program manipulate arrays (read [**?**])
- The clock-directed translation method is a good compromise (simple et reasonnably efficient code).
- Some mix of the two, possibly expressed as a source-to-source transformation in a language that has automata, would be novel
- Would-it simplify the proof that it is correct?
- The clock-directed code generation technique has been re-implemented in several compilers. Bourke et al. have recently succeeded in formalising and proving it in Coq (late 2016).

Finally, single loop code generation impose causality restrictions on feed-back loops. See course notes on the **Optimal Static Scheduling** problem.

Complement: Automata Minimization

- An automaton $= (Initial, State, Label, \rightarrow)$
  with $\rightarrow \subseteq Label \times State \times State$
- Find a partition of $State$ with a equivalence relation

A bisimulation

$p \sim q$ iff

- $p \xrightarrow{a} p' \Rightarrow \exists q \xrightarrow{a} q' \wedge p' \sim q'$
- $q \xrightarrow{a} q' \Rightarrow \exists p \xrightarrow{a} p' \wedge p' \sim q'$

Algorithm: fix-point computation

- $\forall p, q \in State, p \sim_0 q$
- $p \sim_{n+1} q$ iff $\quad p \xrightarrow{a} p' \Rightarrow \exists q \xrightarrow{a} q' \wedge p' \sim_n q'$
  $\qquad\qquad\qquad\qquad\quad q \xrightarrow{a} q' \Rightarrow \exists p \xrightarrow{a} p' \wedge p' \sim_n q'$

In practice, applying automata minimization on the generated automaton does not work.

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.

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.

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.

Marc Pouzet.

Lucid Synchrone, version 3. Tutorial and reference manual.
Université Paris-Sud, LRI, April 2006.