

# MiniLustre mais il fait le Maximum !

or

*Towards the Development of a Certified Compiler for Lustre<sup>a</sup>*

Marc Pouzet

Université Paris-Sud 11 & INRIA Proval

Orsay

ACI SSURF

LIP6, jan. 30th, 2008

Joint work with Dariusz Biernacki (INRIA Proval), Jean-Louis Colaço (Siemens VDO) and Grégoire Hamon (TheMathWorks)

---

<sup>a</sup>Supported by the French ACI Sécurité Alidecs and INRIA

# A Certified compiler for SCADE/Lustre

Implement a verified compiler for a synchronous data-flow language with the help of the proof assistant Coq

Combines **certified compilation** and **translation validation**

## Certified translation

- Write a compilation function  $C : L_1 \rightarrow L_2$  in Coq with its proof of semantics preservation
- natural for local program transformation (e.g., data-flow to sequential code)

## Translation validation

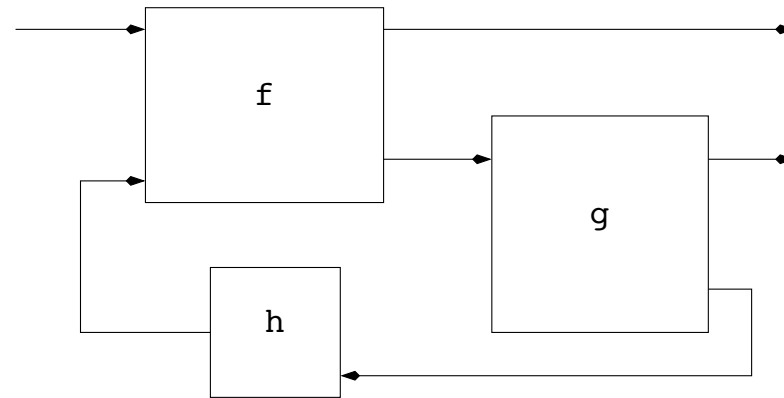
- Write  $C$  independently (e.g., in Caml) and a validation function  $V : L_1 \times L_2 \rightarrow \text{bool}$  with its proof of semantics preservation
- easier for non local transformation (e.g., type or clock inference, find a clever scheduling of equation, memory optimization)

See Xavier Leroy's work for a discussion on pros and cons of both

## Motivations (first step)

First build a reference compiler, as small as possible, purely functional (as much as possible) and based on local rewriting rules

Focus on synchronous block-diagrams as found in Lustre/SCADE or (a subset of) Simulink



- formalize the **code generation** into imperative sequential code (e.g., C)
- as **small** as possible but **realistic** (the code should be efficient)
- make it **modular**, i.e., the definition of a stream function is compiled once for all

as a way to:

- build a certified compiler inside a Proof assistant
- complement previous works on the extension/formalization of synchronous languages

# Code Generation

## Principle:

A stream function  $f : Stream(T) \rightarrow Stream(T')$  is compiled into a pair:

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

a stream equation  $y = f(x)$  is computed sequentially by  $y_n, s_{n+1} = f_t(s_n, x_n)$

## An alternative (more general) solution:

- an initial state:  $s_0 : S$
- a value function:  $f_v : S \times T \rightarrow T'$
- a state modification (“commit”) function:  $f_s : S \times T \rightarrow S'$

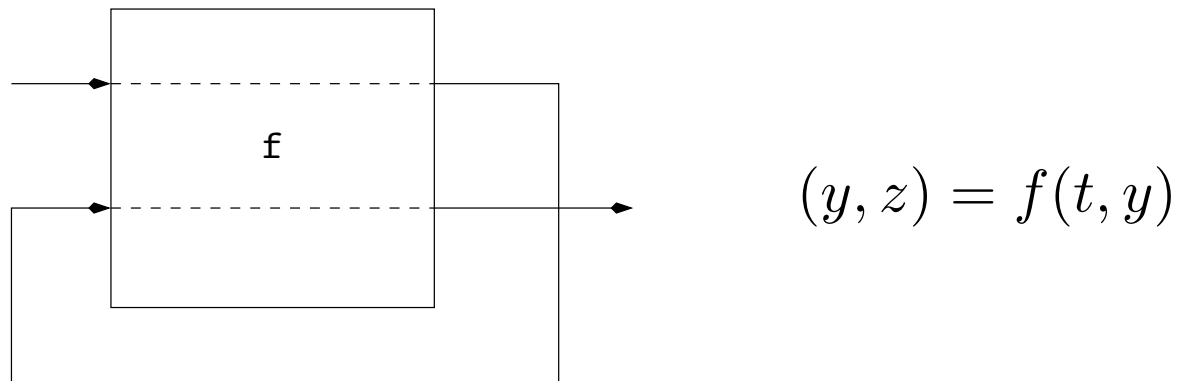
## Final remarks:

- this generalises to MIMO systems
- in actual implementations, states are modified in place
- synchrony finds a very practicle justification here: a data-flow can be implemented as a single scalar variable

## Modular Code Generation

- produce a transition function for each block definition
- compose them together to produce the main transition function
- static scheduling following data-dependences

But modular code generation is not always feasible even in the absence of causality loops



This observation has led to two different approaches to the compilation problem

# Two Traditional Approaches

## Non Modular Code Generation

- full static inlining before code generation starts
- enumeration techniques into (very efficient) automata ([Halbwachs et al., Raymond PhD. thesis, POPL 87, PLILP 91])
- keeps maximal expressiveness but at the price of modular compilation and the size of the code may explode
- finding the adequate boolean variables to get efficient code in both code and size is difficult

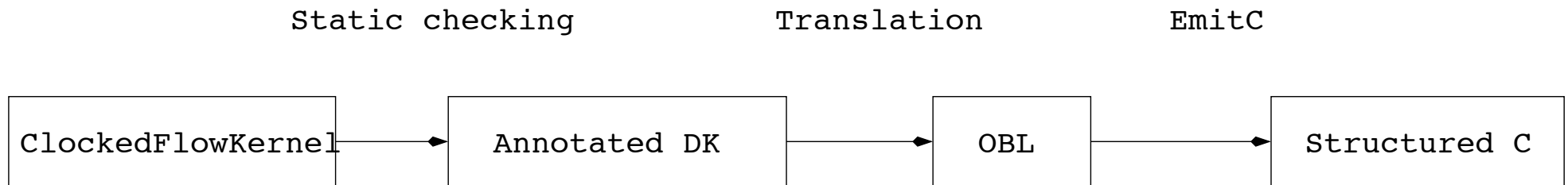
## Modular code generation

- mandatory in industrial compilers
- no preliminary inlining (unless requested by the user)
- imposes stronger causality constraints: every feedback loop must cross an explicit delay
- well accepted by SCADE users and justified by the need for *tracability*

## Proposal

- a compiler where everything can be “traced” with a precise semantics for every intermediate language
- introduce a basic **clocked** data-flow language as the input language
- general enough to be used as a input language for Lustre
- be a “good” input language for modern ones (e.g., mix of automata and data-flow as found in SCADE 6 or Simulink/StateFlow)
- provides a slightly more general notion of clocks
- and a reset construct
- compilation through an intermediate “object based” intermediate language to represent transition function
- provide a translation into imperative code (e.g., structured C, Java)

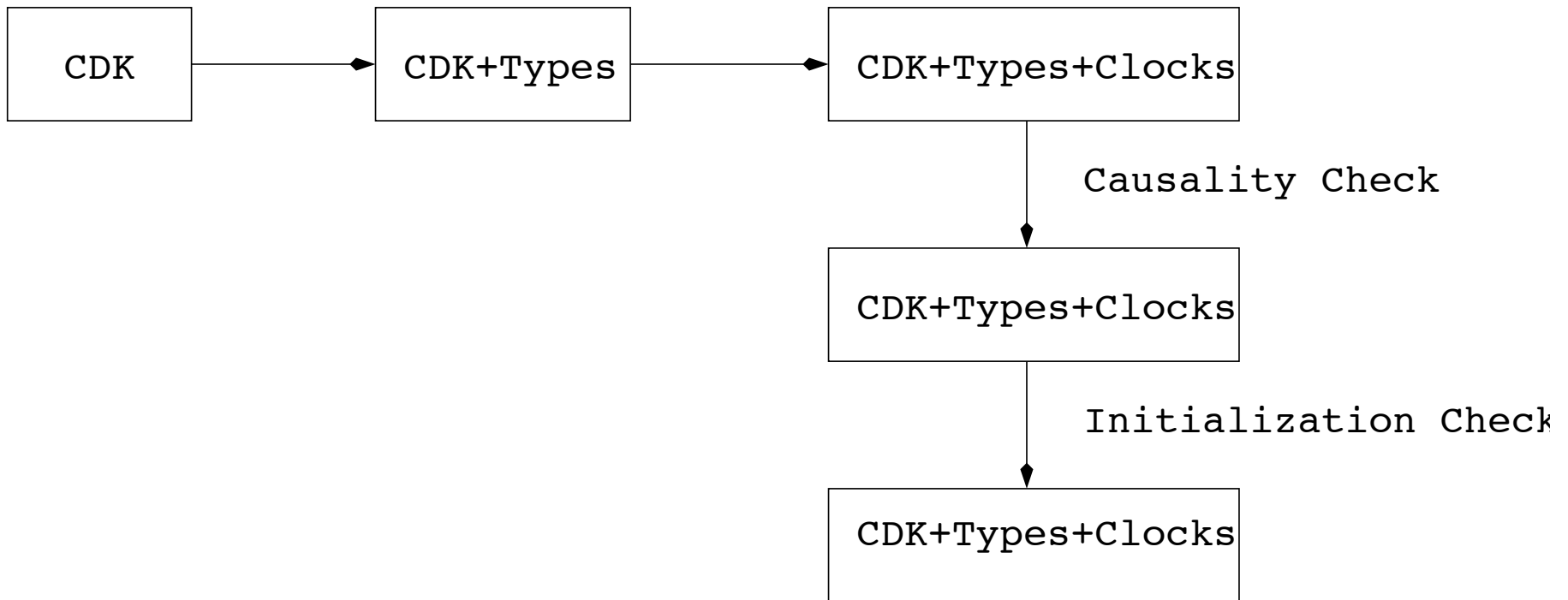
## Organisation of the Compiler



# Static Checking

Type checking

Clock checking



## **A Clocked Data-flow Basic Language**

A data-flow kernel where every expression is explicitly annotated with its clock

$$a ::= e^{ct}$$

$$e ::= v \mid x \mid v \text{ fby } a \mid a \text{ when } C(x) \\ \mid op(a, \dots, a) \mid f(a, \dots, a) \text{ every } a \\ \mid \text{merge } x (C \rightarrow a) \dots (C \rightarrow a)$$

$$D ::= pat = a \mid D \text{ and } D$$

$$pat ::= x \mid (pat, \dots, pat)$$

$$d ::= \text{node } f(p) = p \text{ with var } p \text{ in } D$$

$$p ::= x : bt; \dots; x : bt$$

$$td ::= \text{type } bt \mid \text{type } bt = C + \dots + C$$

$$v ::= C \mid i$$

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

$$ct ::= ck \mid ct \times \dots \times ct$$

## Informal Semantics

$h$	True	False	True	False	...
$x$	$x_0$	$x_1$	$x_2$	$x_3$	...
$y$	$y_0$	$y_1$	$y_2$	$y_3$	...
$v \text{ fby } x$	$v$	$x_0$	$x_1$	$x_2$	...
$x + y$	$x_0 + y_0$	$x_1 + y_1$	$x_2 + y_2$	$x_3 + y_3$	...
$z = x \text{ when } \mathbf{True}(h)$	$x_0$		$x_2$		...
$t = y \text{ when } \mathbf{False}(h)$		$y_1$		$y_3$	...
merge $h$  (True $\rightarrow z$ )  (False $\rightarrow t$ )	$x_0$	$y_1$	$x_2$	$y_3$	...

- $z$  is at a slower rate than  $x$ . We say its clock is  $ck$  on  $\mathbf{True}(h)$
- the `merge` constructs needs its two arguments to be on complementary clocks
- statically checked through a dedicated type system (clock calculus)

## Derived Operators

$\text{if } x \text{ then } e_2 \text{ else } e_3 = \text{merge } x$   
 $(\text{True} \rightarrow e_2 \text{ when } \text{True}(x))$   
 $(\text{False} \rightarrow e_3 \text{ when } \text{False}(x))$

$y = e_1 \rightarrow e_2 = y = \text{if } \textit{init} \text{ then } e_1 \text{ else } e_2$   
and  $\textit{init} = \text{True fby False}$

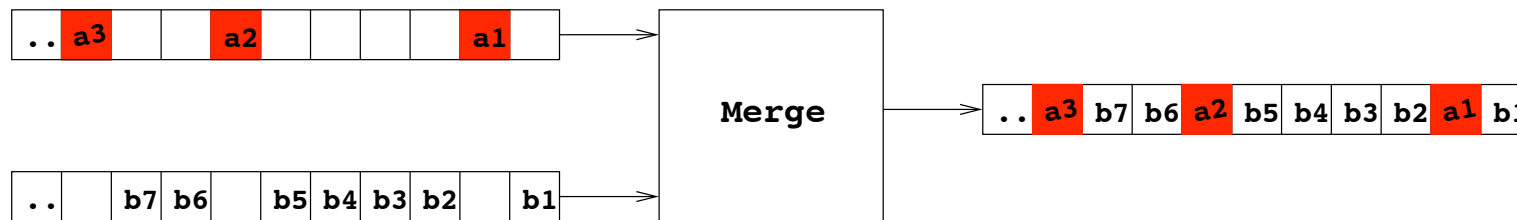
$\text{pre}(e) = \textit{nil} \text{ fby } e$

### Example (counter)

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

## N-ary Merge

`merge` combines two complementary flows (flows on complementary clocks) to produce a faster one:



introduced in Lucid Synchrone V1 (1996), input language of ReLuC

**Example:** `merge c (a when c) (b when not c)`

**Generalization:**

- can be generalized to  $n$  inputs with a specific extension of clocks with enumerated types
- the sampling  $e$  when  $c$  is now written  $e$  when `True( $c$ )`
- the semantics extends naturally and we know how to compile it efficiently
- thus, **a good basic for compilation**

## Resetting a behavior

- in SCADE/Lustre, the “reset” behavior of an operator must be explicitly designed with a specific reset input

```
node count() returns (s:int);
```

```
let
```

```
  s = 0 fby s + 1
```

```
tel;
```

```
node resetable_counter(r:bool) returns (s:int);
```

```
let
```

```
  s = if r then 0 else 0 fby s + 1;
```

```
tel;
```

- painful to apply on large model
- propose a primitive that applies on node instance and allow to reset any node (no specific design condition)

## Modularity and reset

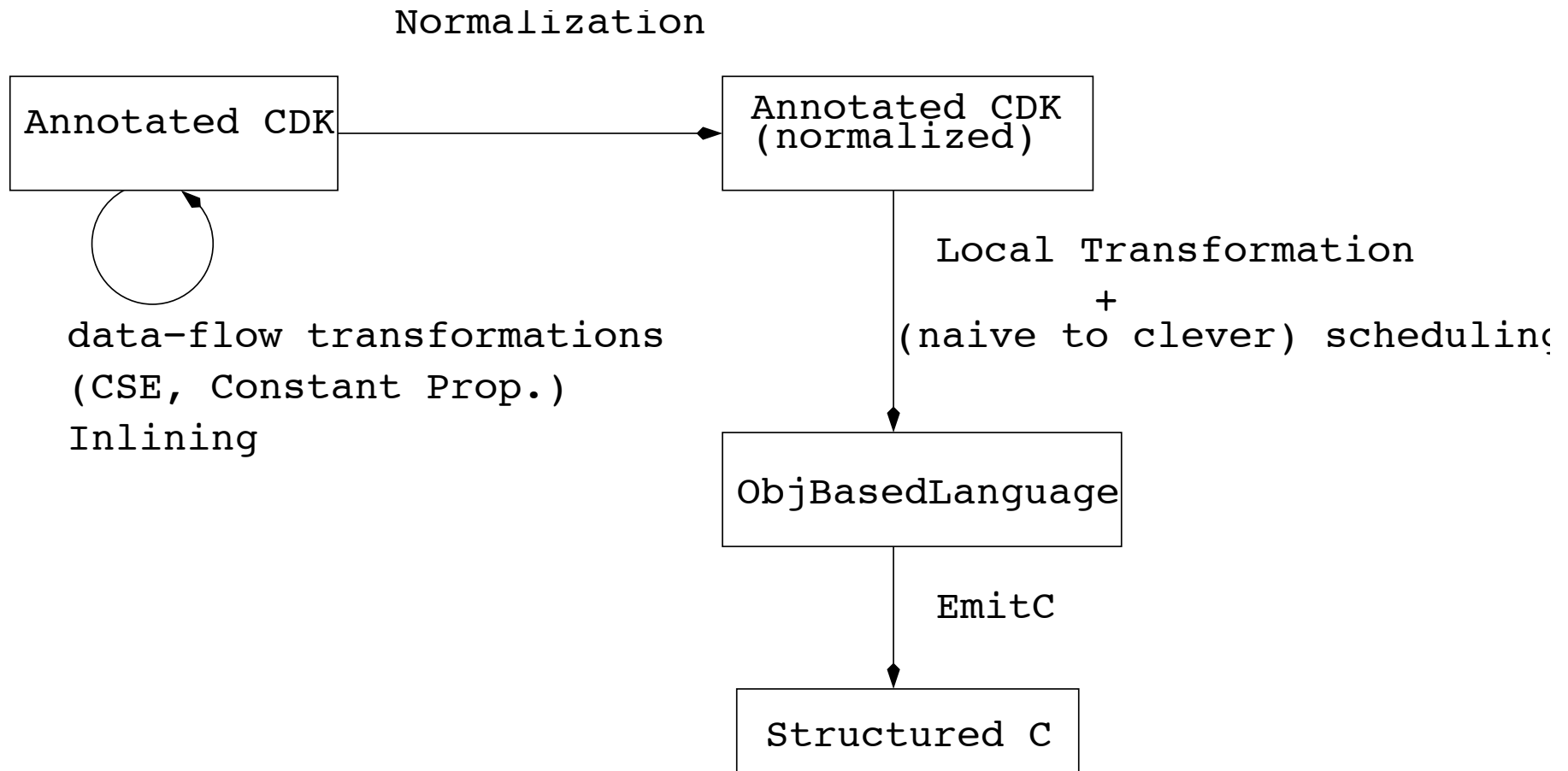
Specific notation in the basic calculus:  $f(a_1, \dots, a_n)$  every  $c$

- all the node instances used in the definition of node  $x$  are reseted when the boolean  $c$  is true
- the reset is “asynchronous”: no clock constraint between the condition  $c$  and the clock of the node instance

**is-it a primitive construct?** yes and no

- modular translation of the basic language with reset into the basic language without reset ([PPDP00], with G. Hamon)
- essentially a translation of the initialization operator  $\rightarrow$
- $e_1 \rightarrow e_2$  becomes `if  $c$  then  $e_1$  else  $e_2$`
- very demanding to the code generator whereas it is trivial to compile!
- useful translation for verification tools, basic for compilation
- thus, **a good basic for compilation**

# Translation



## Syntactic Dependences and Scheduling

Programs which cannot be statically scheduled are rejected during the causality analysis

- we define  $Left(e)$  for the list of variables from  $e$  which are free in  $e$  and not as an argument of a delay  $\text{fb}_y$
- $Left(D)$  is the union of such variables for any expression of  $D$
- for any  $pat = a$  from  $D$ , any variable from  $pat$  depends on  $Left(D)$
- the transitive closure defines the notion of static dependence (Halbwachs et al, [PLILP 91])
- the program can be statically scheduled if there is no cycle
- simple inductive definitions (see [APGES 07] paper)
- an equation  $x = v \text{fb}_y y + 2$  is executed after every equations using  $x$

**Remark:** several classical “graph based” optimization can be applied on this data-flow kernel

- Common Sub-expression Elimination, Constant Propagation, Inlining

## Putting Equations in Normal Form

- prepare equations before the translation
- extract delays from nested expressions by a linear traversal
- Equations are transformed such that delays are extracted from nested computation.

### Normal Form:

$$a ::= e^{ck}$$

$$e ::= a \text{ when } C(x) \mid op(a, \dots, a) \mid x \mid v$$

$$ce ::= \text{merge } x (C \rightarrow ca) \dots (C \rightarrow ca) \mid e$$

$$ca ::= ce^{ck}$$

$$eq ::= x = ca \mid x = (v \text{ fby } a)^{ck}$$

$$\mid (x, \dots, x) = (f(a, \dots, a) \text{ every } x)^{ck}$$

$$D ::= D \text{ and } D \mid eq$$

## Example

$$z = (((4 \text{ fby } o) * 3) \text{ when } \mathbf{True}(c)) + k)^{ck \text{ on } \mathbf{True}(c)}$$

$$\text{and } o = (\text{merge } c (\mathbf{True} \rightarrow (5 \text{ fby } (z + 1)) + 2) \\ (\mathbf{False} \rightarrow ((6 \text{ fby } x)) \text{ when } \mathbf{False}(c)))^{ck}$$

is rewritten into:

$$z = (((t_1 * 3) \text{ when } \mathbf{True}(c)) + k)^{ck \text{ on } \mathbf{True}(c)}$$

$$\text{and } o = (\text{merge } c (\mathbf{True} \rightarrow t_2 + 2) \\ (\mathbf{False} \rightarrow t_3 \text{ when } \mathbf{False}(c)))^{ck}$$

$$\text{and } t_1 = (4 \text{ fby } o)^{ck}$$

$$\text{and } t_2 = (5 \text{ fby } (z + 1))^{ck \text{ on } \mathbf{True}(c)}$$

$$\text{and } t_3 = (6 \text{ fby } x)^{ck}$$

## Intermediate Language

$$\begin{aligned} d & ::= \text{class } f = \\ & \quad \langle \text{memory } m, \\ & \quad \quad \text{instances } j, \\ & \quad \quad \text{reset}() \text{ returns } () = S, \\ & \quad \quad \text{step}(p) \text{ returns } (p) = \text{var } p \text{ in } S \rangle \\ S & ::= x := c \mid \text{state}(x) := c \mid S; S \mid \text{skip} \\ & \quad \mid o.\text{reset} \mid (x, \dots, x) = o.\text{step}(c, \dots, c) \\ & \quad \mid \text{case}(x) \{C : S; \dots; C : S\} \\ c & ::= x \mid v \mid \text{state}(x) \mid \text{op}(c, \dots, c) \\ v & ::= C \mid i \\ j & ::= o : f, \dots, o : f \\ p, m & ::= x : t, \dots, x : t \end{aligned}$$

## Intermediate Language

- the minimal need to represent transition functions
- we introduce an ad-hoc intermediate language to represent them
- it has an “object-based” flavor (with minimal expressiveness nonetheless)
- static allocation of states only
- it can be trivially translated into a imperative language
- we only need a subset set of C (functions and static allocation of structures, very simple pointer manipulation)

## Principles of the translation

- Hierarchical memory model which corresponds to the call graph: one local memory for each function call
- Control-structure (invariant): a computation on clock  $ck$  is executed when  $ck$  is true
- a guarded equations  $x = e^{ck}$  translates into a control-structure

E.g., the equation:

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

is translated into a piece of control-structure:

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

- local generation of a control-structure from a clock

$$\textit{Control}(\textit{base}, S) = S$$

$$\textit{Control}(\textit{ck on } C(x), S) = \textit{Control}(\textit{ck}, \textit{case } (x) \{C : S\})$$

- merge them locally

$$\textit{Join}(\textit{case } (x) \{C_1 : S_1; \dots; C_n : S_n\},$$

$$\textit{case } (x) \{C_1 : S'_1; \dots; C_n : S'_n\})$$

$$= \textit{case } (x) \{C_1 : \textit{Join}(S_1, S'_1); \dots; C_n : \textit{Join}(S_n, S'_n)\}$$

$$\textit{Join}(S_1, S_2) = S_1; S_2$$

- the translation is made on a linear traversal of the sequence of normalized and scheduled equations
- every function defines a machine (a “class”)
- control-optimization: find a static schedule which gather equations guarded by related clocks

## Translation

A context  $(m, si, j, d, s)$ :

- $m$  is the state memory  $[v_1/x_1, \dots, v_n/x_n]$
- $si$  is the initialization code (reset method)
- $j$  is the instance memory  $[f_1/o_1, \dots, f_m/o_m]$
- $d$  is the set of local variables
- $s$  is a sequence of instructions

A few mutually recursive functions:

- $TE_{(m,si,j,d,s)}(e)$  translates an expression in context  $(m, si, j, d, s)$
- $TA_{(m,si,j,d,s)}(x, e^{ck})$  translates an expression storing the result in  $x$
- $TEq_{(m,si,j,d,s)}(eq)$  for equations
- $TEqList_{(m,si,j,d,s)}(eqlist)$  for a list of equations

## Translation

A few definitions (see paper [APGES07] for details)

$$TA_{(m,si,j,d,s)}(x, e^{ck}) = (m, si, j, d, Control(ck, x := TE_{(m,si,j,d,s)}(e)))$$

$$TEq_{(m,si,j,d,s)}(x = ca) = TA_{(m,si,j,d,s)}(x, ca)$$

$$TEq_{(m,si,j,d+[x:t],s)}(x = (v \text{ fby } a)^{ck}) = let\ c = TE_{(m,si,j,d,s)}(a)\ in$$
$$(m + [x : t], [state(x) := v]@si,$$
$$j, d,$$
$$[Control(ck, state(x) := c)]@s)$$

## Example

```
node count (x : int; z : bool) returns (o : int);
var
  i : bool; o2:int;
let
  i = true fby false;
  o = merge i (true -> (x + o2) when true(i))
            (false -> (0 fby o + 1) when false(i));
  o2 = merge i (true -> (42 fby (o when true(i))) + 1)
            (false -> 0);
tel;
```

```

class count {
    x_1 : bool; x_3 : int; x_2 : int;

    reset() { mem x_1 = true; mem x_3 = 42; mem x_2 = 0; }

    step(x : int; z : bool) returns (o : int) {
        i : bool; o2 : int;

        i = mem(x_1);
        mem x_1 = false;
        switch (i) {
            case false :
                o2 = 0;
                o = mem(x_2) + 1;
            case true :
                o2 = mem(x_3) + 1;
                o = x + o2;
                mem x_3 = o;
        };
        mem x_2 = o; }

```

## Example (modularity)

- each function is compiled separately
- a function call needs a local memory

```
node count(x:int) returns (o:int);
```

```
let
```

```
  o = 0 fby o + x;
```

```
tel;
```

```
node conduct(c:bool;input:int) returns (o:int);
```

```
let
```

```
  o = merge c (true -> count(input when true(c)))  
            (false -> (0 fby o) when false(c));
```

```
tel;
```

```

class conduct {
  x_2 : int; x_4 : count;

  reset () {
    x_4.reset ();
    mem x_2 = 0;
  }

  step(c : bool; input : int) returns (o : int) {
    x_3 : int;

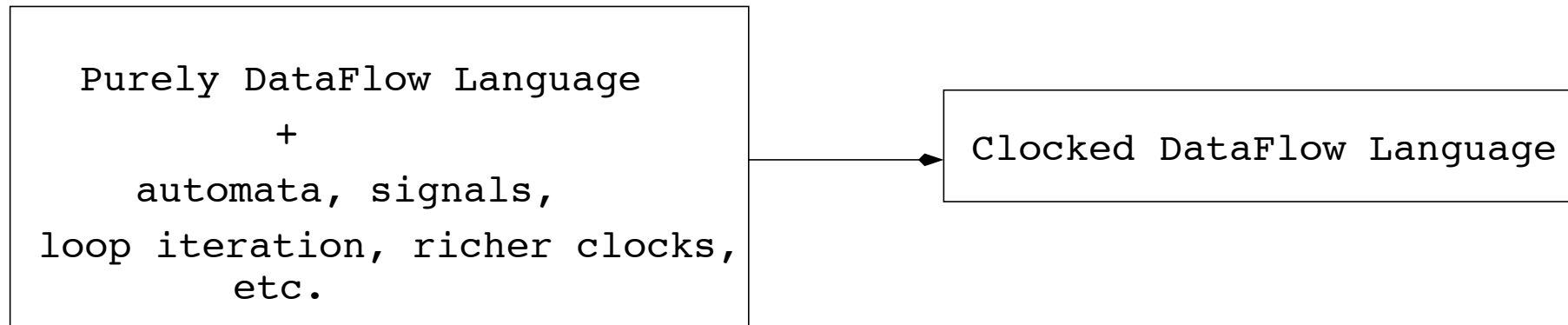
    switch (c) {
      case true :
        (x_3) = x_4.step(input);
        o = x_3;
      case false :
        o = mem(x_2);
    };
    mem x_2 = o; }

```

## MiniLustre in Numbers

<b>Administrative code</b>	Abstract syntax + printers	335
	Lexer&Parser	546
	main (misc, symbol tables, loader)	285
<b>Basis</b>	graph	74
	scheduling	67
	type checking	269
	clock checking	190
	causality check	30
	normalization	95
	translate (to ob)	132
	<b>Emitters to concrete languages</b>	(C, Java and Caml)
<b>Optimizations</b>	Inline + reset	250
	Dead-code Removal	42
	Data-flow network minimization	162

## Extensions (towards a full language)



- extend the source language with new programming constructs
- translation semantics into the basic data-flow language
- this is essentially the approach we have followed previously (Lucid Synchronic, ReLuC compiler of SCADE)
- clocks play a central role
- simple and gives very good code
- reuse of the existing code generator (adequate in the context of a certification process)

**Question:** What about polymorphism and higher-order?

## Formal Certification (Coq programming)

In parallel, we have done:

- an implementation of MiniLustre in the programming language of Coq (1500 loc)
- extracted caml code + hand-coded caml code to get the compiler
- type and clock inference also done

We are currently working on the semantics and proof of equivalence between the source language and the intermediate language

## Semantics

We (finally) choose a “reaction semantics” (in SOS style) for the source language

Values:  $w+ ::= w \mid (w+, \dots, w+)$

$w ::= abs \mid v$

Reaction Environnement:  $R ::= [w_1/x_1, \dots, w_n/x_n] \quad (i \neq j \Rightarrow x_i \neq x_j)$

Reaction:  $R \vdash_{ck} e_1 \xrightarrow{w+} e_2 \quad R \vdash D \xrightarrow{R'} D'$

**Lemma 1 (Normalization)** *if  $D_N \in Norm(D)$  then  $R \vdash D \xrightarrow{R'} D'$  then  $R \vdash D_N \xrightarrow{R'} D'_N \wedge D'_N \in Norm(D')$*

**Lemma 2 (Scheduling)** *if  $D_S \in Sch(D)$  then  $R \vdash D \xrightarrow{R'} D'$  then  $R \vdash D_S \xrightarrow{R'} D'_S \wedge D'_S \in Sch(D)$*

We define the predicate  $R \vdash_{seq} D \xrightarrow{R'} D'$  for normalized and scheduled equations;  $Init(D)$  gives the initial state (left part of  $\text{fby}$ ).

**Lemma 3 (Sequential computation)** *if  $D_S \in Sch(Norm(D))$  then  $R, R' \vdash D \xrightarrow{R'} D'$  iff  $R, Init(D) \vdash_{seq} D_S \xrightarrow{R_0} D'_S \wedge D'_S \in Sch(Norm(D')) \wedge R' = Init(D_S), R_0$*

## Semantics for the Intermediate Language

An operational one (in SOS style two). No fix-point.

$$\rho ::= [v_1/x_1; \dots; v_m/x_n] \text{ where } x_i \neq x_j \text{ for all } i \neq j$$

and

$$m ::= [v_1/x_1; \dots; v_n/x_n]$$

$$j ::= [O_1/o_1; \dots; O_m/o_m]$$

$$M ::= \langle m, j \rangle$$

$$O ::= \langle M, \text{reset} = S, \text{step} = \lambda p.q \text{ with } S \rangle$$

Two predicate:

- $M, \rho \vdash e \Downarrow v$ :  $e$  evaluates to  $v$  in  $M$  and  $\rho$
- $M, \rho \vdash S \Downarrow \rho', M'$  for instructions

Prove the preservation of semantics for the translation function.

# Conclusion

## Current

- a reference (small) MiniLustre compiler has been implemented
- semantics “on paper” (source and intermediate language) and semantics preservation of the translation

## Future (relatively close)

- Coq development of the semantics preservation
- finish the Coq programming of the reference compiler (combines **translation validation** (e.g., scheduling) and **certified compilation**)

## Future (longer term)

- mixed systems (data-flow systems + mode-automata)
- source-to-source transformation into the data-flow system
- translation semantics (as done in ReLuC [EMSOFT'05, EMSOFT'06])
- the reference implementation MiniLustre has been done accordingly (about 300 extra lines of Caml code)