A certified compiler for the synchronous langage Lustre

TYPES 2007

Alexandre Bertails
joint work with D. Biernacki, C. Paulin, M. Pouzet

INRIA – Université Paris-Sud

May 4th 2007

plan

- a certified compiler for Lustre
- static analysis
  - type checking
  - clock calculus
  - causality analysis
- conclusion

Lustre : a synchronous langage

- P. Caspi, N. Halbwachs, D. Pilaud, J. Plaice, 1987
- designed for critical real-time systems (planes, nuclear power plants)
- both hardware and software, well-suited for control
- dataflow paradigm : streams of values
- synchronee : logical time model (*ie.* no time in computations)
- programs are equations compiled to automata with memory

motivations for a certified compiler

- collaboration with Esterel Technologies
  - a certified compiler by extraction closed to Scade 6
  - integration of new features from Lucid Synchrone (inference, higher-order, polymorphism and more)
- first attempt at Dassault Aviation (E. Gimenez and E. Ledinot)
  - extracted compiler
  - part of semantic preservation was proved but no static analysis
  - code unavailable
- inspiration from certified C compiler of X. Leroy (CompCert project)

counter

```
node COUNTER (c: bool) returns (x: int);
let
  x = 0 -> if c
           then pre(x) + 1
           else pre(x)
tel
```

instants	0	1	2	3	4	5	6	7	8	⋯
c	*T*	*T*	*F*	*T*	*F*	*F*	*T*	*T*	*T*	⋯
x	0	1	1	2	2	2	3	3	3	⋯

Outline

a certified compiler for Lustre

static analysis
- type checking
- clock calculus
- causality analysis

conclusion

type checking

- basic types : `bool`, `int`, `float`, ...
- types are implicitely streams (`int` is implicitely `stream int`)
- we want to reject `1 + true` (flows of different types)
- no polymorphism, no higher-order : trivial type system

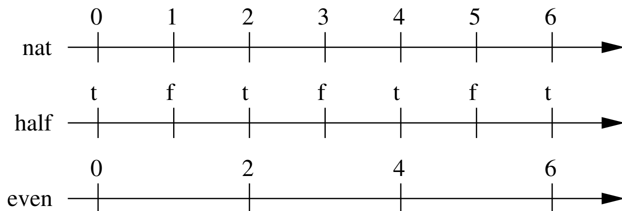
$$\frac{G, L \vdash_{exp} e_1 : int \qquad G, L \vdash_{exp} e_2 : int}{G, L \vdash_{exp} e_1 + e_2 : int} \text{ tplus}$$

$$\frac{G, L \vdash_{exp} e : t}{G, L \vdash_{exp} \texttt{pre}\ e : t} \text{ tpre}$$

nat and even

```
 nat = 0 -> 1 + pre nat
half = true -> not (pre half)
even = nat when half
```

- `nat` and `half` have the same clock
- `even` has clock `half`



clocks are polymorphic and dependent types

```
node COUNTER (c: bool) returns (x: int on c)
  var n : int
let
  (x, n) = (n when x, 0 -> 1 + pre n)
tel
```

$COUNTER :: [c : \alpha] \rightarrow [n : \alpha \texttt{ on } c]$

```
half' = true when half
```

$half' :: \alpha \texttt{ on } half$
$COUNTER\ half' :: \alpha \texttt{ on } half \texttt{ on } half'$

clock calculus (some rules)

$$\frac{G, L \vdash_{exp} e_1 :: ck \qquad G, L \vdash_{exp} e_2 :: ck}{G, L \vdash_{exp} e_1 + e_2 \ :: ck} \text{ cplus}$$

$$\frac{\begin{array}{l} e_2 \textit{ is syntacticaly equal to some variable } c \\ G, L \vdash_{exp} e_1 : ck \\ G, L \vdash_{exp} e_2 : ck \end{array}}{G, L \vdash_{exp} e_1 \texttt{ when } e_2 :: ck \texttt{ on } c} \text{ cwhen}$$

causality analysis (goal)

- a variable must not depend instantaneously on itself

```
x = y
y = z
z = x
```

- we will accept

```
x = 0 -> pre x
```

causality analysis (as it is done)

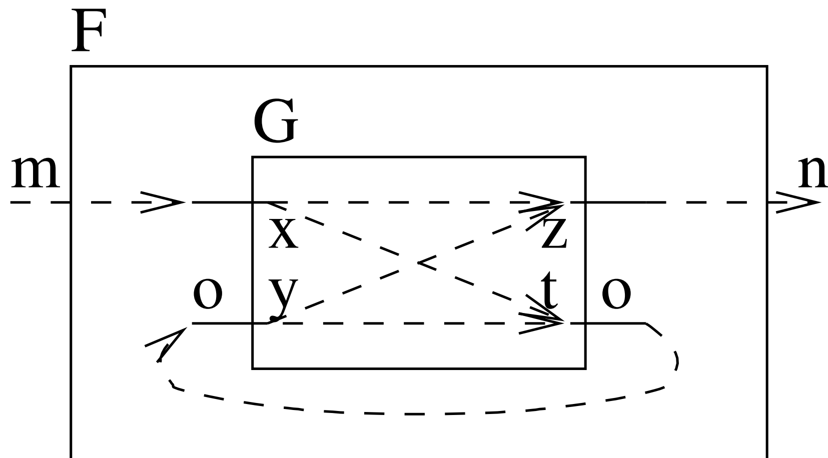
- trade-off between modularity and accepted programs
- academic Lustre (inlining, list of dependencies)
- industrial Lustre – Scade 6 (boxes, modularity)
- we want the best and simplicity

causality analysis (boxes)

```
node F (m: int) returns (n: int)
var o: int
let
  (n, o) = G (m, o)
tel
```

- if `G` is a box, `o` depends on `o` !

causality analysis (boxes)



causality analysis (boxes)

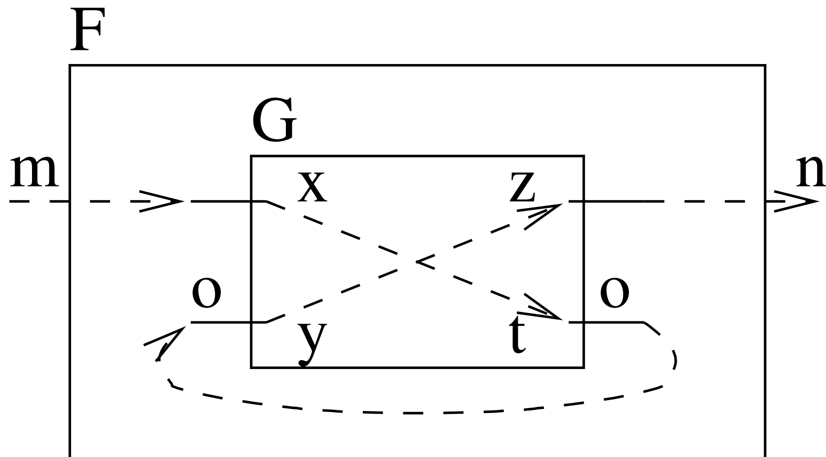
```
node F (m: int) returns (n: int)
var o: int
let
  (n, o) = G (m, o)
tel
```

- if `G` is a box, `o` depends on `o` ! but if `G` is

```
node G (x: int; y : int) returns (z: int; t: int)
let
  z = y
  t = x
tel
```

- here, `n` depends on `o` that depends on `m` : no problem !

causality analysis (what we want)



causality analysis (inlining)

```
node F (m: int) returns (n: int)
var o: int
let
  (n, o) = G (m, o)
tel

node G (x: int; y : int) returns (z: int; t: int)
let
  z = y
  t = x
tel
```

causality analysis (inlining)

```
node F (m: int) returns (n: int)
var o: int
let
  n = o
  o = m
tel
```

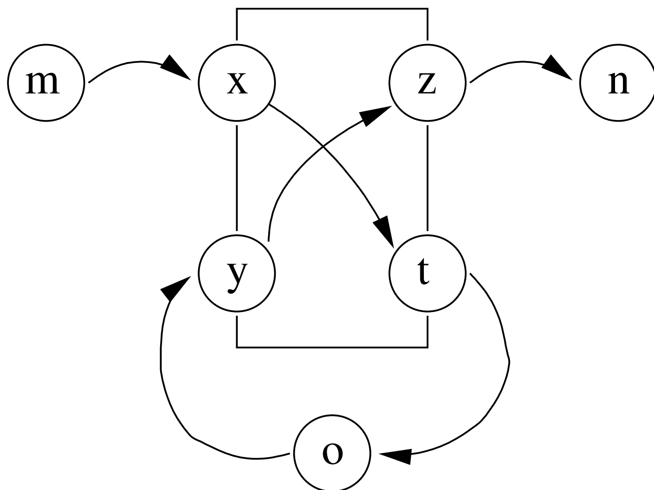
n depends on $\{o\}$
o depends on $\{m\}$

causality analysis (list of dependencies)

```
node F (m: int) returns (n: int)
var o: int
let
  n = o
  o = m
tel
```

n depends on $\{o, m\}$
o depends on $\{m\}$

causality analysis : $(n, o) = G(m, o)$



causality analysis (preliminaries)

- here, causality analysis is a type system where **types are dependent graphs** : $\{G \;:\; graph \mid no_cycle\ G\}$
- **vertexes** are **variables**
- $[x \longrightarrow y] \equiv y\ depends\ on\ x$
- **node application** is just **plugging inputs and outputs**
- there is a loop through $\mathtt{x} \Leftrightarrow \mathtt{x}$ depends on itself

causality analysis (preliminaries)

- a certified Coq graph library with a certified cycle detection algorithm
- types : `vertex`, `edge` and `graph`
- graph construction
  - $\square : \mathtt{graph}$
  - $\cup : \mathtt{graph} \to \mathtt{graph} \to \mathtt{graph}$
  - $+ : \mathtt{graph} \to \mathtt{edge} \to \mathtt{graph}$
  - $@ : \mathtt{graph} \to (\mathtt{set\ vertex} \times \mathtt{vertex}) \to \mathtt{graph}$
    $\mathtt{G}\ @\ [\{\mathtt{x}_1 \cdots \mathtt{x}_n\} \to \mathtt{x}] = \mathtt{G} + [\mathtt{x}_1 \to \mathtt{x}] + \cdots + [\mathtt{x}_n \to \mathtt{x}]$
- predicate $\mathtt{no_cycle} : \mathtt{graph} \to \mathtt{Prop}$

causality analysis (relations)

- node type
  $\mathtt{node_dep} = \mathtt{list\ var} \times \mathtt{list\ var} \times \{\mathtt{G} : \mathtt{graph} \mid \mathtt{no_cycle\ G}\}$
- environment
  $\Gamma : \mathtt{map\ funname\ node_ca}$
- collecting dependencies in expressions
  $\vdash_{exp} e : s$
- expressing constraints in equations
  $\Gamma \vdash_{eq} eq : G$
- typing node
  $\Gamma \vdash_{node} \ node : node_dep$

causality analysis (collecting dependencies)

$$\frac{\vdash_{exp} e_1 : s_1 \qquad \vdash_{exp} e_2 : s_2}{\vdash_{exp} e_1 + e_2 : s_1 \cup_{set} s_2} \text{ ca_plus}$$

$$\frac{}{\vdash_{exp} x : \{x\}} \text{ ca_var}$$

$$\frac{}{\vdash_{exp} \texttt{pre}\ e : \{\ \}} \text{ ca_pre}$$

causality analysis (expressing constraints)

$$\frac{\vdash_{exp} e : s \qquad G = \square \,@\, [s \rightarrow x]}{\Gamma \vdash_{eq} x = e : G} \text{ ca_eqsimple}$$

$$\frac{\vdash_{exp} e_i : s_i \qquad G = \square \,@\, [s_1 \rightarrow x_1] \,@\, \cdots \,@\, [s_n \rightarrow x_n]}{\Gamma \vdash_{eq} (x_1, \cdots, x_n) = (e_1, \cdots, e_n) : G} \text{ ca_eqpat}$$

$$\frac{\vdash_{exp} e : s \qquad \Gamma(f) = i_f \times o_f \times (G_f, H) \qquad fresh(i_f \times o_f \times G_f) = i \times o \times G \qquad G' = G \,@\, [s \rightarrow i] + [o \rightarrow x]}{\Gamma \vdash_{eq} x = f(e) : G'} \text{ ca_eqfun}$$

causality analysis (typing nodes)

$$\frac{\Gamma \vdash_{eq} eq : G \qquad \Gamma \vdash_{eqs} eqs : G'}{\Gamma \vdash_{eqs} eq\ ;\ eqs : G \cup G'}\ \text{ca_eqs}$$

$$\frac{\Gamma \vdash_{eqs} eqs : G \qquad H : no_cycle\ G}{\Gamma \vdash_{node} \begin{array}{l}\texttt{node}\ f(\vec{i})\ \texttt{returns}\ (\vec{o})\\ \texttt{var}\ \cdots\ \texttt{let}\ eqs\ \texttt{tel}\end{array} : \vec{i} \times \vec{o} \times (G, H)}\ \text{ca_node}$$

Outline

a certified compiler for Lustre

static analysis
- type checking
- clock calculus
- causality analysis

conclusion

conclusion and ongoing work

- type checking and clock calculus are already defined by inductives relations in Coq (mostly done by D. Biernacki)
  49 `Definition`, 16 `Fixpoint`, 45 `Inductive`, 80 `Theorem`
- OCaml code is extracted from lemmas of decidability
  1236 lines of extracted OCaml code (3889 of Coq)
- we present a new causality analysis based on types as graphs without cycles
- work in progress
  - Coq graph library
  - causality analysis and its extraction
  - semantics, code generation

conclusion and ongoing work

- type checking and clock calculus are already defined by inductives relations in Coq (mostly done by D. Biernacki)
  49 `Definition`, 16 `Fixpoint`, 45 `Inductive`, 80 `Theorem`
- OCaml code is extracted from lemmas of decidability
  1236 lines of extracted OCaml code (3889 of Coq)
- we present a new causality analysis based on types as graphs without cycles
- work in progress
  - Coq graph library
  - causality analysis and its extraction
  - semantics, code generation

Thank you for your attention. Any questions ?