Compilation of synchronous programs

General problem

Transform a (hierarchic) parallel program into a (simple) sequential program.
Whole implementation of a reactive program $P$

```plaintext
var I, O, M;
M := m0; proc P_step() ...;
foreach step do
  read(I);
  P_step(); // combines: O := f(M, I); M := g(M, I);
  write(O);
end foreach
```

Job of the compiler

- Find the memory $M$ and its initial value $m0$
- Build the core of the loop (the $P$.step procedure)
- As far as possible, generate efficient code

---

Modular compilation problem

The "obvious" way of compiling

A Lustre node $\rightarrow$ a step procedure.

```
var A, X, Y, S;
proc F_step() begin X := ... end
proc G_step() begin Y := ... end
proc P_step()
begin
  F_step();
  G_step();
  S := X + Y;
end
```

Compilation
Problem

What about feed-back loops?

\[ X = A \times (2 - A) \]

• The program "is" correct (in a "parallel" world),
• but no \( F \) step procedure can work!

Solution(s)

• Lustre (academic): expansion (i.e. inlining) of node calls
  \( \hookrightarrow \) Strictly compliant with the principle of substitution.
  \( \hookrightarrow \) Forbids modular compilation.

• Scade: feedback loops (without \texttt{pre}) are forbidden.
  \( \hookrightarrow \) Reject correct parallel programs.
  \( \hookrightarrow \) Allow modular compilation.
  \( \hookrightarrow \) Reasonable choice in a industrial framework.

• Compilation into ordered blocks aka Modular Static Scheduling
  \( \hookrightarrow \) Intermediate solution
  \( \hookrightarrow \) Split the step into a minimal set of (sequential) blocks,
  \( \hookrightarrow \) Only expand this simplified structure.
An example of Modular Static Scheduling

- Interesting theoretical result.
- Not (yet?) used in industry.

Compilation of Lustre

Example: a filtered counter

- count rising edges of \( X (F) \),
- reset with a delay (\( R \)).

```plaintext
node CptF(X, reset: bool) returns (cpt: int);

var F, R : bool;

let

    cpt = if R then 0
    else if F then pre cpt + 1
    else pre cpt;

R = true -> pre reset;
F = X -> (X and not pre X);

tel
```

Compilation of Lustre
Simple loop compilation

Intuitively, do what is necessary to make definitions equivalent to assignments, i.e.:

- translate classical operators (trivial),
- replace pre's and ->'s with memory constructs,
- sequentialize according to data-dependencies (i.e. static scheduling).

Identify the memory

- Introduce a explicit variable for each pre:
  - pcpt = pre cpt;
  - preset = pre reset;
  - pX = pre X;
- Introduce a special memory
  - init = true -> false;
  and replace each:
  - x -> y
  with
  - if init then x else y
New version of the Lustre program

cpt = if R then 0
    else if F then pcpt + 1
    else pcpt;
R = if init then true else preset;
F = if init then X else (X and not pX);
pcpt = pre cpt;
preset = pre reset;
pX = pre X;
init = true -> false;

Sequentialization

Must take into account:

• Instantaneous dependences between values,

  ↦ an (partial) order MUST exist (no combinational loop),
  example: R before cpt and F before cpt

  ↦ chose a compatible complete order (schedule),
  example R, then F then cpt.

• Memorisations

  ↦ Must be done at the end of the step, in any order.
Simple loop implementation (C-like code)

- Arithmetic and logic are translated “as it”
  (ex. `and` becomes `&&`, `if..then..else` becomes `..?..:`)
- `pre`’s are replaced with memories
- `->`’s are replaced with `init?..:`
- Inputs/outputs are stores in global variables (for instance)

```c
int cpt; bool X, reset; /* I/O global vars */
int pcpt; bool pX, preset; /* non initialized memories */
bool init = true; /* the only necessary initialization */

void CptFiltre_step() {
    bool R, F; /* local vars */
    R = init ? true : preset;
    F = init ? X : (X && ! pX);
    cpt = R ? 0 : F ? pcpt + 1 : pcpt;
    pcpt = cpt; pX = X; preset = reset;
    init = false;
}
```
Optimizations

- Control structure: ? becomes if
- Factorize conditions
- Eliminate useless local vars

```plaintext
if (init) {
    cpt = 0;
    init = false;
} else {
    F = (X && ! pX);
   cpt = preset ? 0 : F ? (pcpt+1) : pcpt;
}
pcpt = cpt; pX = X; preset = reset;
```

Compilation of Lustre

Compilation into automaton

Idea

The following reactive automaton:

```
N := 0
N := N + 1
off
on
¬ on
¬ off
```

is exactly equivalent to a Lustre program:

```plaintext
node Chrono(on, off : bool) returns (N : int);
var R : bool;
let
    R = false -> pre(if R then not off else on);
    N = if R then (pre N + 1) else 0;
tel
```

Problem: how to build the automaton from the Lustre code?
Goal
• Automatically build an automaton equivalent to a Lustre program

How?
• Idea: an (explicit) state ⇔ a valuation of the memory
• N.B. finite number of states ⇒ finite memory (e.g Boolean)

Example of CptF
• $S1 =$ initial state = “init true, all other undefined”
• simplified code : \(\text{cpt} = 0\)
  • integer memorization: still the same
  • Boolean memorization: state transition

Compilation into automaton ___________________________ 16/24

Transitions
• State $S1$ (initial):
  \[\text{init} = \text{false}; \ pX = X; \ \text{preset} = \text{reset};\]

Depending on the values of $X$ and \text{reset}, 4 next states:
• \(X \land \text{reset} \rightarrow S2 \equiv \overline{\text{init}} \land pX \land \text{preset}\)
• \(X \land \overline{\text{reset}} \rightarrow S3 \equiv \overline{\text{init}} \land pX \land \overline{\text{preset}}\)
• \(\overline{X} \land \text{reset} \rightarrow S4 \equiv \text{init} \land \overline{pX} \land \text{preset}\)
• \(\overline{X} \land \overline{\text{reset}} \rightarrow S5 \equiv \text{init} \land \overline{pX} \land \overline{\text{preset}}\)

Compilation into automaton ___________________________ 17/24
- Code of the other states:
  \( S2 \rightarrow \text{cpt} = 0 \)
  \( S3 \rightarrow \text{cpt} = \text{pcpt} \)
  \( S4 \rightarrow \text{cpt} = 0 \)
  \( S5 \rightarrow F = X, \text{cpt} = X? (\text{pcpt} + 1) : \text{pcpt} \)
- Transitions of the other states:
  \( \leftarrow \) same than \( S1 \) (only depend on inputs)

Compilation into automaton 18/24

Finally ...

⇒ problem: size!
Remarks on the size

- \( n \) memories \( \iff \) (worst case) \( 2^n \) states, \( 2^{2n} \) transitions
  \( \Rightarrow \) **Combinatorial explosion**

But not always:

- Unreachable states
  \( \Leftarrow \) Example: \( (\text{pre } X, \text{pre}(X \text{ or } Y)) \Rightarrow \text{“only” 3 states} \)
  \( \Leftarrow \) Counter-example: \( \text{CpfF} \)!

- State equivalence
  \( \Leftarrow \) Example \( \text{CpfF}: S1, S2 \text{ et } S4 \text{ “are doing the same thing”} \)
  \( \Rightarrow \) **Importance of producing a minimal automaton**

Minimal automaton of \( \text{CpfF} \)

\[
\begin{array}{c}
\text{M1} \\
cpt = 0;
\end{array}
\begin{array}{c}
\text{M2} \\
cpt = \text{pcpt};
\end{array}
\begin{array}{c}
\text{M3} \\
cpt = X \ ? \\
(pcpt + 1) : \text{pcpt};
\end{array}
\]
Implementation en C

With a switch (for instance):

typedef enum {M1, M2, M3} TState;
TState state = M1;

void CptFiltre_step()
{
    switch (state) {
        case M1: cpt = 0; break;
        case M2: cpt = pcpt; break;
        case M3: cpt = X? (pcpt + 1):pcpt; break;
    }
    pcpt = cpt;
    if (reset) state = M1;
    else if (X) state = M2;
    else state = M3;
}

Simple loop or automaton ?

Automate

- Optimal in computation time
- Possible huge size

Simple loop

- Less slower
- Linear size

⇒ Only “reasonable” solution from an industrial point of view

Automaton, what for?

- Not reasonable for code generation, but ...
- Precious for reasoning about programs, i.e. for validation.
The stopwatch

```python
node Stopwatch(on_off, reset, freeze: bool) returns(time:int);
var running, freezed:bool; cpt:int;
let
    running = Switch(on_off, on_off);
    freezed = Switch(
        freeze and running,
        freeze or on_off);
    cpt = Count(reset and not running, running);
    time = if freezed then (0 -> pre time) else cpt;
```

- expand, identify memory, sequentialize ...
- automaton ...

Example/demo _____________________________ 24/24