Systèmes Synchrones
MPRI – cours 2.23-1 2014-2015

Final Exam
25 november, 2014

This text has 4 pages. The time limit is 3h. Courses notes are allowed.

Activation Conditions, Sequencing and Modular Reset

In this exam, you will extend a language kernel similar to Lustre with control structures and study their translation into the kernel. The syntax of the language is given below.

$$
\begin{array}{lcl}
d & ::= & \texttt{node}\ f(p)\ \texttt{returns}\ (q)\ B \\
p, q, r & ::= & x \mid x, ..., x \\
B & ::= & \texttt{var}\ r\ \texttt{do}\ E\ \texttt{done} \\
E & ::= & E\ \texttt{and}\ E \mid x = e \\
e & ::= & v \mid x \mid \texttt{true} \mid \texttt{false} \mid e + e \mid e = e \mid e\ \texttt{and}\ e \mid e\ \texttt{or}\ e \mid \texttt{not}\ e \\
 & & \mid \texttt{pre}\ e \mid \texttt{preb}\ e \mid e\ \texttt{->}\ e \mid \texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e
\end{array}
$$

d is the definition of a node with formal parameters p, result q and body B. The body B is a set of equations E where variables from r are local to E. p, q and r denote patterns; here lists of variables. E stands for equations of the form $x = e$, with e an expression, and parallel compositions of equations, E `and` E. v denotes an integer constant. `true` and `false` are boolean constants. $+$ stands for integer addition; `and` for logical conjunction; `or` for logical disjunction, and; `not` for negation. `pre` e is the unit delay for integers. `preb` e is the unit delay for booleans. To avoid initialization issues, the initial value of `pre` e is -1; the initial value of `preb` e is `false`.

Relational Semantics: We define a relation semantics for this kernel. Let V^∞ be the set of sequences of values from V. If $v \in V^\infty$ and $n \in \mathbb{N}$, $v(n)$ is the n-th element of v. An environment ρ is a mapping from names to sequences. Given ρ and an equation E, $[\![E]\!]_\rho$ means that E satisfies ρ. If e is an expression, $[\![e]\!]_\rho(n)$ with $n \in \mathbb{N}$ is the value of e at instant n. We do not require the semantics to be deterministic. That is, there may be an equation E and two environments ρ_1 and ρ_2, with $\rho_1 \neq \rho_2$, such that $[\![E]\!]_{\rho_1}$ and $[\![E]\!]_{\rho_2}$. We give (only) the main cases below.

$$
\begin{array}{lcl}
[\![E_1\ \texttt{and}\ E_2]\!]_\rho & \stackrel{def}{=} & [\![E_1]\!]_\rho \wedge [\![E_2]\!]_\rho \\
[\![x = e]\!]_\rho & \stackrel{def}{=} & \forall n \in \mathbb{N}, [\![x]\!]_\rho(n) = [\![e]\!]_\rho(n) \\
[\![e_1 + e_2]\!]_\rho(n) & \stackrel{def}{=} & [\![e_1]\!]_\rho(n) + [\![e_2]\!]_\rho(n) \\
[\![\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3]\!]_\rho(n) & \stackrel{def}{=} & \textit{if}\ [\![e_1]\!]_\rho(n)\ \textit{then}\ [\![e_2]\!]_\rho(n)\ \textit{else}\ [\![e_3]\!]_\rho(n) \\
[\![\texttt{preb}\ e]\!]_\rho(0) & \stackrel{def}{=} & \texttt{false} \\
[\![\texttt{pre}\ e]\!]_\rho(0) & \stackrel{def}{=} & -1 \\
[\![\texttt{preb}\ e]\!]_\rho(n+1) & \stackrel{def}{=} & [\![e]\!]_\rho(n) \\
[\![\texttt{pre}\ e]\!]_\rho(n+1) & \stackrel{def}{=} & [\![e]\!]_\rho(n) \\
[\![x]\!]_\rho(n) & \stackrel{def}{=} & \rho(x)(n) \\
[\![v]\!]_\rho(n) & \stackrel{def}{=} & v \\
[\![\texttt{var}\ x_1, ..., x_n\ \texttt{do}\ E\ \texttt{done}]\!]_\rho & \stackrel{def}{=} & \exists s_1, ..., s_n . [\![E]\!]_{\rho + [x_1 \mapsto s_1, ..., x_n \mapsto s_n]}
\end{array}
$$

Question 1 Define the following operations in terms of the kernel language:

1. `until`(x) returns a sequence *ok* that is initially false and that only becomes true as soon as x is true in the strict past. Once *ok* becomes true, it stays true.

2. `unless`(x) returns a sequence *ok* with current value true as soon as x is true. The current value of *ok* is false otherwise. Once *ok* is true, it stays true.

3. `init`(x, y) is true whenever x is true, otherwise it becomes false if y was true in the preceding instant, otherwise it keeps its previous value. Note that `init(false, true)` = `true -> false`.

The following questions involve extending the kernel language with new programming constructs by defining the cases of a translation function $Tr(.)$ where $Tr(E)$ takes an equation E and returns another equation E'.

Blocks with Local Variables

The language of equations E is extended with a block construct:

$$E \quad ::= \quad \cdots \mid \texttt{var}\ r\ \texttt{do}\ E\ \texttt{done}$$

The semantics is that of a block.

Question 2 Define a translation function $Tr(.)$ so that $Tr(E)$ translates E from the extended language into the kernel one.

Activation Condition

The kernel language is now extended with an "activation condition" mechanism. The syntax is given below:

$$E \quad ::= \quad \texttt{activate if}\ e\ \texttt{then}\ E \mid ...$$

Intuitively, in `activate if` e `then` E, the equation E is active only at the instants when e is true. Otherwise, variables from E keep their previous values. For example, the following program defines the sequence: `cpt = -1 -1 42 43 43 43 44 45 45 45 46 47 ...`

```
 activate if cond then do cpt = 42 -> pre cpt + 1 done
and
 cond = false -> (preb (false -> not (preb cond)))
```

Question 3 Is the previous program equivalent to the following one? Explain why.

```
    cpt = if cond then 42 -> pre cpt + 1 else pre cpt
and cond = false -> (preb (false -> not (preb cond)))
```

Question 4 Propose an equivalent version that does not use the "activation condition" control structure.

Question 5 Define a translation function $Tr(E)$ which translates E from the extended language into a semantically equivalent equation E' from the kernel language.

Question 6 Propose a sufficient condition on `activate if` e `then` E so that its translation is causally correct, in the Lustre sense.

We now extend the syntax and semantics of activation conditions to allow a default handler to be executed when the boolean condition is false.

$$E \quad ::= \quad \texttt{activate if } e \texttt{ then } E \texttt{ else } E \mid ...$$

For example, the following program:

```
activate if cond then do cpt = 42 -> pre cpt - 1 done
         else cpt = 45 -> pre cpt + 1 done
and cond = false -> (preb (false -> preb (false -> not (preb cond))))
```

defines the sequence `cpt = 45 46 47 42 41 40 48 49 50 39 38 37 ...` (`pre cpt` denotes a local memory updated only when the code in which it appears is active. The two occurrences of `pre cpt` denote different memories).

Question 7 Give an equivalent definition without using the binary activation condition.

Question 8 Extend the translation function $Tr(.)$ accordingly. You may assume that the sets of non-local variables defined in E_1 and E_2 in `activate if` e `then` E_1 `else` E_2 are the same.

Question 9 Extend the translation function $Tr(.)$ to handle the general situation where the two branches do not necessarily define the same variables.

Question 10 Propose a relational semantics $[\![.]\!]_.$ that extends the basic one with the new construct `activate if` e `then` E_1 `else` E_2.

Sequencing Operations

We now introduce sequencing constructs.

$$E \quad ::= \quad \cdots \mid \texttt{do } E \texttt{ until } e \texttt{ then } E \mid \texttt{do } E \texttt{ unless } e \texttt{ then } E$$

`do` E_1 `until` e `then` E_2 gives weak preemption: E_1 is activated up to and including the first instant that the boolean condition e becomes true. The execution of E_2 then starts in the following instant. `do` E_1 `unless` e `then` E_2 gives strong preemption: E_1 is executed up to but not including the first instant that e is true. E_2 starts at the first instant when e is true. Thus, the program:

```
do x = 0 -> pre x + 1 until (x = 5) then x = 10 done
```

defines the sequence `x = 0 1 2 3 4 5 10 10 10 10 ...` The following program:

```
  do x = 0 -> pre x + 1 unless cond then x = 10 done
and
  cond = false -> preb (false -> true)
```

defines the sequence `x = 0 1 10 10 10 ...`

Question 11 Extend the translation function $Tr(.)$ with the two sequencing constructs.

Question 12 Define a causality constraint that ensures the translated code is causally correct in the Lustre sense.

Question 13 Can one of the constructs (weak *versus* strong) be expressed in terms of the other?

Question 14 Propose a relational semantics for these two programming constructs.

Modular Reset

The language is now extended with the construct

$$E \quad ::= \quad \cdots \mid \texttt{reset}\ E\ \texttt{every}\ e$$

that reinitializes every stateful construct to its initial value. Thus, the program:

```
reset
  cpt = 0 -> pre cpt + 1
every
  (false -> pre cpt = 5)
```

defines the sequence `cpt = 0 1 2 3 4 5 0 1 2 3 4 5 ...`

Question 15 What is the translation of `reset x = 0 -> 1 every c`?

Question 16 Extend $Tr(.)$ so that the construction `reset` E `every` e is eliminated and expressed in the kernel language.

Question 17 [Extra] Define a relational semantics for the reset construct.

Exceptions

The kernel language is now extended with a programming construct to raise and trap exceptions.

$$E \quad ::= \quad \texttt{exit}\ T \mid \texttt{try}\ E\ \texttt{with | }\ T\ \texttt{then}\ E\ ...\ \texttt{| }\ T\ \texttt{then}\ E\ \texttt{done}$$

`exit` T raises the exception with name T (we suppose that exception names are declared globally). The construct `try` E `with |` T_1 `then` E_1 ... `|` T_n `then` E_n `done`, where $T_1, \ldots, T_n$ are supposed to be pairwise distinct, executes E and at the instant T_i is raised, the corresponding block E_i becomes active for the rest of the execution.

An exception T can be raised several times in a single instant (e.g., `exit` T `and exit` T) with the same effect as a single raise. Two different exceptions can also be raised simultaneously (e.g., `exit` T_2 `and exit` T_1): the first matching handler in the list of handlers is activated (here E_1).

Question 18 Propose an encoding of `exit` T, and extend $Tr(.)$ accordingly.

Question 19 Propose an encoding of exceptions into the kernel language and extend $Tr(.)$ accordingly.