Towards a formalization of a clock calculus

Martin Strecker
Université Paul Sabatier / IRIT, Toulouse

1/12/2008

Plan

Group-specific projects

Context: projects Geneauto and Spacify
Here: “Bottom up” approach:

- Start from a language with few primitives
- Work towards a more complex modelling language

Purpose:

- Better understanding of the language
- Basis for correctness proofs of
  - model transformations
  - compilation
- Coherence of analyses like
  - model checking
  - clock calculus

. . . wrt. the language semantics

Related work

Standard programming languages:

- Formalization of language semantics
- Compiler correctness proofs

. . . over the past 30 years

Synchronous languages:

- David Nowak (1999): co-inductive coding of the *semantics* of a synchronous language
  - no formal relation between syntax and semantics
  - therefore: impossible to talk about semantic consequences of syntax manipulations (transformations, compilation, . . .)
- Marc Pouzet (2007/2008): verification of a Lucid Synchrone compiler in Coq (?)

Overview

In this talk:

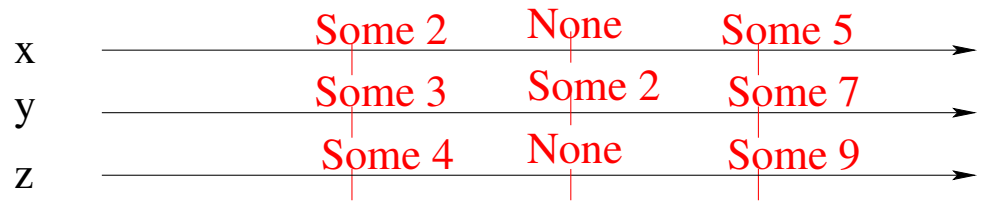
- Formalization of a language in the spirit of SIGNAL
  - first-order
  - “relational” (not: functional) semantics
- Clock types
- *Application:* type soundness proof:
  “no manipulation of undefined values in well-typed programs”

Particularities:

- *no* co-inductive semantics
  - But: mapping from a temporal domain to values
  - *Purpose:* eventually reason about a quantitative notion of time
- named representation of variables
  - (no de Bruijn indices)
  - *Purpose:* integration in a model-driven engineering environment

Plan

Signals and traces




```
types ('t,'v) signal = 't ⇒ 'v option
```

```
types ('t,'n,'v) trace = 't ⇒ 'n ⇒  'v option
```

- `'t` temporal domain (most often: `nat`)
- `'v` value domain
  `'v option`: Value present / absent
- `'n` domain of names / variables

Signal, the language

```
process Sampler = {integer N}
     (? event reset, tick ! integer val; event alarm)
  (| val := Count(reset default alarm)
   | mod := (val = N-1)$1 init true
   | alarm := when mod
   | val ^= reset ^+ tick
   |) where boolean mod;
end;
```

Currently modelled:

- Combinatorial expressions
- Statements: "assignment" `:=`, composition, hiding (`where`)
- Processes (without static parameters)
- partly: synchronization constraints

Signal: Expressions

```
datatype ('n,'v) expr
  =
    (* explicit naming of variables *)
    Var 'n
  | (* (e_0, e_1) *)
    PairE (('n, 'v) expr) (('n, 'v) expr)
  |  (* f (e)  *)
    Fun funid  (('n, 'v) expr)
  | (* e $1 init v *)
    Pre 'v  (('n, 'v) expr)
  | (* e_0  when e_1 *)
    When (('n, 'v) expr)  (('n, 'v) expr)
  | (* e_1 default e_1 *)
    Default (('n, 'v) expr) (('n, 'v) expr)
```

Signal: Statements

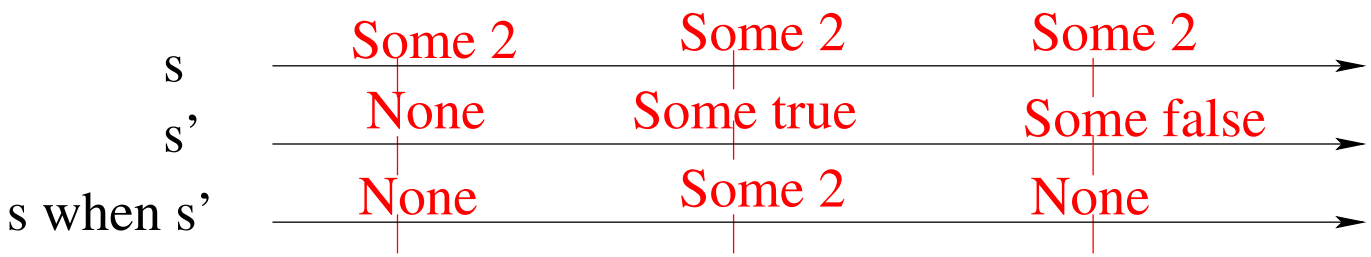
```
datatype ('n, 'v) stmt
  =
    EmptyStmt
  | (* y := e *)
    Eq 'n   (('n, 'v) expr)
  | (*  c_1 | c_2 *)
    Par (('n, 'v) stmt) (('n, 'v) stmt)
  | (* c where {x} *)
    Letv 'n   (('n, 'v) stmt)
  | (* P(i_1 .. i_m) (o_1 .. o_n) *)
    PCall ('n cname) ('n list) ('n list)
```

Functions on signals (1)

. . . for preparing the semantics of expressions.
Functions without restriction of the temporal domain:

```
when :: ('t,'v) signal ⇒ ('t,'v) signal
        ⇒ ('t,'v) signal

when s s' == λ t.
             case (s' t) of
               None ⇒ None
             | Some v ⇒
               if (true_val v) then (s t) else None
```

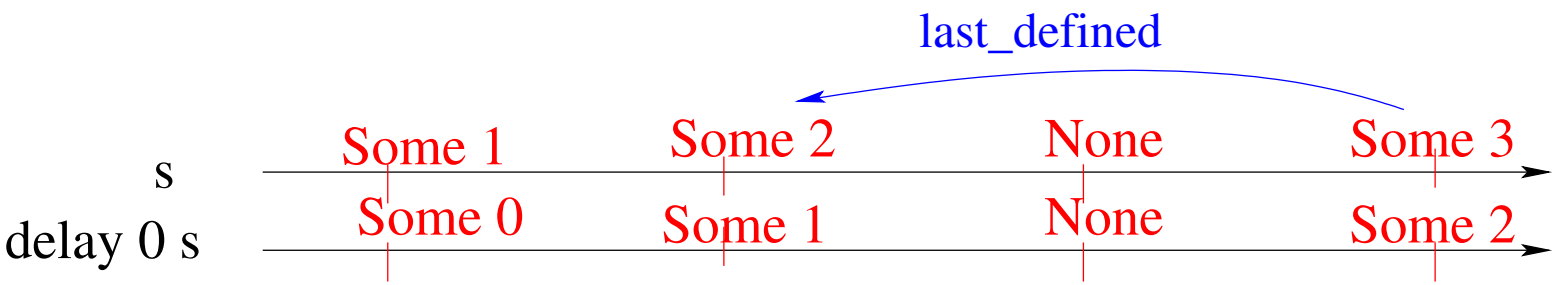


Functions on signals (2)

Functions with restriction of the temporal domain:

```
delay :: 'v ⇒ (nat, 'v) signal ⇒ (nat, 'v) signal

delay v s == λ t.
             case (s t) of
               None ⇒ None
             | Some v' ⇒ Some (last_defined v s t)
```



Other semantic domains than `nat`?

Plan

Canonical form (1)

Aim: Avoid nested expressions like:

```
y := (x $1 init 0) when (b $1 init true)
```

Transformation in canonical form:

```
(|
 | y := x' when b'
 | x' := (x $1 init 0)
 | b' := (b $1 init true)
 |) where { x', b' }
```

Canonical form (2)

Expressions: Only variables, pairs, functions
Statements:

```
datatype ('n,'v,'a) canon_stmt
= CanEmptyStmt
| CanCombin 'n  (('n,'v) canon_expr)
| CanPre 'n 'v 'n
| CanWhen 'n 'n 'n
| CanDefault 'n 'n 'n

| CanPar (('n,'v,'a)canon_stmt) (('n,'v,'a)canon_stmt)
| CanPCall ('n cname) ('n list) ('n list)
| CanLetv 'n (('n,'a) clstmt) (('n,'v,'a) canon_stmt)
```

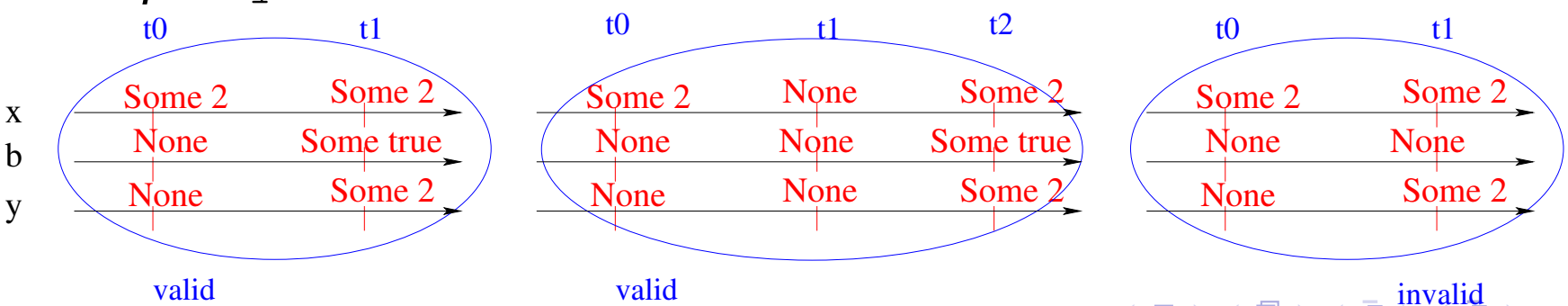
Annotations with a clock type $\rightsquigarrow$ later

Semantics: Simple cases

First approximation: "Trace *tr* is a model of statement *c*"

```
inductive
  interp_canon_stmt ::
  (nat,'n,'v) trace ⇒ ('n, 'v, 'a) canon_stmt ⇒ bool
where
| interp_canonCanWhen:
    trace_proj tr y =
    when (trace_proj tr x) (trace_proj tr b)
  ⟹ interp_canon_stmt tr (CanWhen y x b)
```

Example: `y = x when b`



Semantics with errors (1)

What is the result of (x, y) if x and y are not synchronous?

1. Semantics without errors:

```
signalpair :: ('t,'v) signal ⇒ ('t,'v) signal
      ⇒ ('t,'v) signal
signalpair s s' == λ t. lift_pair (s t) (s' t)
lift_pair (Some v)(Some v') = Some(pair_val (v,v'))
lift_pair _          _          = None
```

2. Semantics with errors:

```
datatype 'v error = Err | OK 'v
signalpair_err :: ('t,'v error) signal
  ⇒ ('t,'v error) signal ⇒ ('t,'v error) signal
lift_pair_err None None      = None
lift_pair_err(Some v) None =Some Err (& viceversa)
lift_pair_err (Some v) (Some v')
          = Some (lift_err v v')
```

Semantics with errors (2)

Extension: Two variants of semantics of expressions:

1. `interp_canon_expr error_fns_flat tr e`
2. `interp_canon_expr error_fns_lifted tr e`

Second approximation: "Trace *tr* is a model of statement *c*, taking / not taking errors into account"
Case `y := e`, blocking in case of errors:

```
inductive
  interp_canon_stmt :: bool ⇒ (nat,'n,'v) trace ⇒
       ('n, 'v, 'a) canon_stmt ⇒ bool
where
| interp_canonCanCombin_bloc:
   s = (interp_canon_exp error_fns_lifted tr e) ∧
   error_free_signal s ∧
   trace_proj tr y = cancel_err_signal s
 ⟹ interp_canon_stmt True tr (CanCombin y e)
```

Semantics with errors (3)

Case `y := e`, non-blocking in case of errors:

```
| interp_canonCanCombin_nonbloc:
    s = (interp_canon_exp error_fns_lifted tr e) ∧
    trace_proj tr y = cancel_err_signal s
 ⟹ interp_canon_stmt False tr (CanCombin y e)
```

Auxiliary functions:

```
consts   cancel_err :: 'v error ⇒ 'v
primrec  cancel_err (OK v) = v

constdefs
  cancel_err_signal ::
      ('t, 'v error) signal ⇒ ('t, 'v) signal
  cancel_err_signal s == (option_map cancel_err) ∘ s
```

Plan

Clocks

Clock: Set of instants when a signal is present

- $cl(x)$: instants when x is present
- $[y > 5]$: instants when $y > 5$ is present and true

Clock calculus: establishes the conditions under which a set of signals is synchronized.

- Statement in SIGNAL: `z := x when (y > 5)`
- Derived clock condition: $cl(z) = cl(x) \cap [y > 5]$

Clock calculus

To be done:

- Definition of `clexpr` (corresponds to `expr` of SIGNAL).
  Semantics: set of instants
- Definition of `clstmt` (corresponds to `stmt` of SIGNAL).
  Semantics: boolean
  $\rightsquigarrow$ Calculus for determining the validity of a `clstmt`
- Extraction of a `clstmt` from a `stmt`
- Show *type soundness*:
  If `stmt` *c* has a valid `clstmt`
  and *tr* is a trace which is a model of *c*,
  then *tr* is an error-free trace

Thus: for every `stmt` *c* with a valid `clstmt`,
the flat and lifted semantics coincide.

Example (1)

- Block B with input (a, b, c) and output (x, y, z), defined by:

```
process B(a, b, c) (x, y, z)
(| x := a when c
 | y := x $1 init 0
 | z := y when c |)
```

- Assume the "interface specification" is: $cl(b) = cl(a) \cap [c]$
- Derive clock equations: (1) $cl(x) = cl(a) \cap [c]$
  (2) $cl(y) = cl(x)$
  (3) $cl(z) = cl(y) \cap [c]$
- After simplification: Constraints on output values:
  $cl(x) = cl(y) = cl(z) = cl(a) \cap [c]$

Example (2)

Treatment of local variables - first approach:

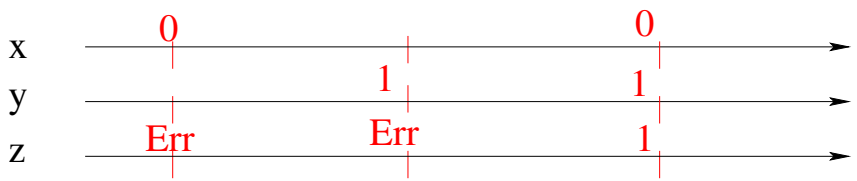
Program:

```
(| x := x $1 init 0
 | y := y $1 init 1
 | z = x + y |)
  where { x, y }
```

Constraints:

$\exists x\ y.$
$cl(x) = cl(x) \wedge$
$cl(y) = cl(y) \wedge$
$cl(z) = cl(x) = cl(y)$
equivalent to *True*

Problem: without additional constraints, the program is eventually non-synchronous.



Example (3)

Second approach:

Program:

```
(| x := x $1 init 0
 | y := y $1 init 1
 | z = x + y |)
  where { x, y }
  with x ^= y ^= z
```

Constraints:

$\forall x\ y.$
$cl(x) = cl(y) = cl(z)$
$\longrightarrow$
$(cl(x) = cl(x) \wedge$
$cl(y) = cl(y) \wedge$
$cl(z) = cl(x) = cl(y))$
. . . also equivalent to *True*

Thus:

Necessary to add constraints for the synchronization of local variables.

Clock expressions (1)

```
datatype ('n, 'a) clexpr
  = ClOf 'n              (* cl(x) *)
  | ClWhen 'a            (* [cond] *)
  | ClInter (('n, 'a) clexpr) (('n, 'a) clexpr)
  | ...
```

- `'n` domain of variable names
- `'a` abstraction of boolean expressions

Abstractions can later be instantiated to:

- simple boolean conditions: `y = x when b`
- temporal conditions: `y = x when (time mod 10 = 0)`
  (see related work by Julien Forget)
- ...

Clock expressions (2)

Interpretation of clock expressions wrt. a concretization function `concr`:

```
interp_clexpr tr concr (ClOf x) = dom(trace_proj tr x)
interp_clexpr tr concr (ClWhen a) = (concr tr a)
interp_clexpr tr concr (ClInter ce ce') =
     (interp_clexpr tr concr ce) ∩
     (interp_clexpr tr concr ce')
```

Clock statements

```
datatype ('n, 'a) clstmt
  = ClEq (('n, 'a) clexpr) (('n, 'a) clexpr)
  | ClConj (('n, 'a) clstmt) (('n, 'a) clstmt)
  | ClLet 'n  (('n, 'a) clstmt) (('n, 'a) clstmt)
  | ...
```

Interpretation:

```
interp_clstmt tr concr (ClEq e e') =
  (interp_clexpr tr concr e) =
  (interp_clexpr tr concr e')
interp_clstmt tr concr (ClConj c c') =
  interp_clstmt tr concr c ∧ interp_clstmt tr concr c'
interp_clstmt tr concr (ClLet v c c') =
  ∀ s. interp_clstmt (trace_upd tr v s) concr c
       ⟶  interp_clstmt (trace_upd tr v s) concr c'
```

Extraction of constraints

```
consts clock_canon_stmt :: (('n, 'v) canon_expr ⇒ 'a)
      ⇒ ('n, 'v, 'a) canon_stmt  ⇒ ('n, 'a) clstmt
primrec
clock_canon_stmt abstr (CanCombin y e) =
  clock_combin y e
   (*  cl(y) = cl(x1) = ... = cl(xn),
         if fv(e) = {x1 .. xn} *)
clock_canon_stmt abstr (CanWhen y x b) =
  ClEq (ClOf y)
       (ClInter (ClOf x) (ClWhen (abstr (CanVar b))))
clock_canon_stmt abstr (CanPar c c') =
   ClConj (clock_canon_stmt abstr c)
          (clock_canon_stmt abstr c')
clock_canon_stmt abstr (CanLetv v clk c) =
   ClLet v clk (clock_canon_stmt abstr c)
```

Type soundness

Theorem

If a trace *tr* is a (non-blocking) model for a statement *c*
and if the clock constraints for *c* are satisfiable for *tr*
then *tr* is a blocking model (in the case of error) for *c*

```
interp_canon_stmt False tr c ∧
interp_clstmt tr concr (clock_canon_stmt abstr c)
 ⟶  interp_canon_stmt True tr c
```

and some side conditions (relation between `concr` and `abstr` ...)
Expressed differently: If *c* has a model *tr* and *c* is well typed, then *c* does not provoke synchronisation errors.

Plan

Summary

Done so far:

- Formalization of the semantics of a synchronous language
- Clock types: equality of instances of occurrence of signals
  - currently an *abstract* notion
  - . . . abstracting away from particular conditions of occurrences

To be done:

- Instantiate abstract notion to concrete domains
- Develop and prove soundness of solvers
  $\rightsquigarrow$ *type checking* of annotated programs
- Develop methods of *type inference*