Integer Clocks and Local Time Scales
Part I – Part II

Adrien Guatto
ENS - PARKAS

SYNCHRON 2014
Part I
Critical Control Software:
- Process unbounded sequences of data
- ... within bounded memory
- ... and bounded reaction time.

Synchronous Digital Hardware:
- Process unbounded sequences of data
- ... within bounded memory
- ... and bounded reaction time.

Synchronous Programming Languages: program both!
Synchrony and Performance-Sensitive Code

- Traditional use cases: control laws, protocols, etc.
- Signal processing: involve...
  - subtle space/time tradeoffs
  - architecture-dependent optimizations
- Can we use Synchronous Languages for such applications?

Long-Term Objective

Design and implement a...
- synchronous functional language
- compiling to hardware and software
- with the usual safety guarantees
- but generating code of a different shape
Ingredients

**Integer Clocks**
- Compute streams by bursts of value
- Generate nested loops from purely functional code

**Local Time Scales**
- Time may pass faster inside than outside
- Time is now *ambient* rather than *global*
- Make the type system more uniform

**Linear Higher-Order Functions**
- Call every function you receive exactly once
- Enable *modular* compilation to hardware
This Talk

- Present Integer Clocks and Local Time Scales intuitively
  - Reason purely on stream functions à la Lustre, Lucid S., Lucy-n
  - Focus on first-order parts
- Show how the intuitions can be implemented as a type system
  - (Check buffers sizes)
  - Reject non-causal programs
- Discuss soundness results
  - Proof by realizability
Streams and Partiality

- Streams are *infinite* sequences of values
  - Think of them as produced by programs running forever
- However, streams may be *partial*, i.e. block after some time!
  - Happens when the producer program does an infinite, silent loop.
- Here is a picture of \textit{Stream}(B), ordered by \textit{information}:

```
      ⊥ ~ 0
       ↕ ← 0. ⊥ ~ 0
          ↕ ← 0. ⊥ ~ 0
             ↕ ← ⊥ ~ 0
                ↕ ← ⊥ ~ 0
                   ↕ ← 1
                      ↕ ← 1
                         ↕ ← 1
                              ↕ ← ⊥ 1
```

Adrien Guatto (ENS - PARKAS)
Consider the following function

\[ f : \text{Stream}(\mathbb{N}) \rightarrow \text{Stream}(\mathbb{N}) \]
\[ f(x.xs) = (x + 1).(f(xs)) \]

Can it be implemented as a state machine? Yes. For example:

\[ m : \mathcal{M}(\mathbb{N}, \mathbb{N}) \]
\[ m = (\{\ast\}, \ast, \lambda(\ast, x).(*, x + 1)) \]

The machine \( m \) processes one element per transition. It was easy since the function is \emph{length-preserving}. 

---

Adrien Guatto (ENS - PARKAS)  Integer Clocks and Local Time Scales  SYNCHRON 2014
What about the following function?

\[
g : \text{Stream}(\mathbb{N}) \rightarrow \text{Stream}(\mathbb{N})
g(x.xs) = (x + 1).(x - 1).(g xs)
\]

Yes, if we cheat a bit.

\[
m_1 : \mathcal{M}(\mathbb{N}, \text{List}(\mathbb{N}))
\]
\[
m_1 = (\{\ast\}, \ast, \lambda(\ast, x).(\ast, [x + 1; x - 1]))
\]

Another possibility:

\[
m_2 : \mathcal{M}(\text{List}(\mathbb{N}), \mathbb{N})
\]
\[
m_2 = (\mathbb{N} \cup \{\ast\}, \ast,
\lambda(s, x).\text{if } s = \ast \text{ then } (\text{hd } x, \text{hd } x + 1) \text{ else } (\ast, s - 1))
\]
Naively speaking, the function $g$ is not length-preserving.

$$
g : \text{Stream}(\mathbb{N}) \rightarrow \text{Stream}(\mathbb{N})
g (x.xs) = (x + 1).(x - 1).(g \hspace{2pt} xs)
$$

However, we can make it so by changing its (co)domain!

$$
g_1 : \text{Stream}(\text{List}(\mathbb{N})) \rightarrow \text{Stream}(\text{List}(\mathbb{N}))
g_1 ([x].xs) = [x + 1; x - 1].(g_1 \hspace{2pt} xs)
$$

$$
g_2 : \text{Stream}(\text{List}(\mathbb{N})) \rightarrow \text{Stream}(\text{List}(\mathbb{N}))
g_2 ([x].xs) = [x + 1].(\text{let } [] \hspace{2pt} xsp = xs \hspace{2pt} \text{in } [x - 1].(g_2 \hspace{2pt} xsp))
$$

Functions $g_1$ and $g_2$ are length-preserving.
Synchronizing Functions

How to describe the relationship between \( g \), \( g_1 \) and \( g_2 \)?

\[
\begin{align*}
g & : \text{Stream}(\mathbb{N}) \rightarrow \text{Stream}(\mathbb{N}) \\
g_1 & : \text{Stream}(\text{List}(\mathbb{N})) \rightarrow \text{Stream}(\text{List}(\mathbb{N})) \\
g_2 & : \text{Stream}(\text{List}(\mathbb{N})) \rightarrow \text{Stream}(\text{List}(\mathbb{N}))
\end{align*}
\]

Remember that \( g_1 \) and \( g_2 \) work only for specific list sizes:

<table>
<thead>
<tr>
<th></th>
<th>Input list sizes</th>
<th>Output list sizes</th>
</tr>
</thead>
<tbody>
<tr>
<td>( g_1 )</td>
<td>(1)( \omega )</td>
<td>(2)( \omega )</td>
</tr>
<tr>
<td>( g_2 )</td>
<td>(1 0)( \omega )</td>
<td>(1)( \omega )</td>
</tr>
</tbody>
</table>

These integer streams, \textit{clocks}, fully characterize \( g_1 \) and \( g_2 \). We write:

\[
\begin{align*}
g_1 & :: (1) \rightarrow (2) \\
g_2 & :: (1 0) \rightarrow (1)
\end{align*}
\]
From Streams to Clocked Streams, and back

A clock $w$ is just a stream of integers!
What can we do with such a $w \in \text{Stream}(\mathbb{N})$?

$\text{Stream}(V)$

$\text{Stream}(\text{List}(V))$

$\text{pack}_w$

$\text{unpack}$

For example:

$x = \text{pack}_{1 (10)}(a. b. c. d \ldots) \quad [a] \quad [b] \quad [] \quad [c] \quad [] \quad \ldots$

$y = \text{pack}_{0 (2)}(a. b. c. d \ldots) \quad [] \quad [a; b] \quad [] \quad [c; d] \quad [] \quad \ldots$

Obviously:

$\text{unpack } x = \text{unpack } y$
We now define the functions $g_1$ and $g_2$ purely from their clocks:

\[
\begin{align*}
g_1 &:: (1) \to (2) \\
g_1 & = \text{pack}_{(2)} \circ g \circ \text{unpack} \\
g_2 &:: (10) \to (1) \\
g_2 & = \text{pack}_{(1)} \circ g \circ \text{unpack}
\end{align*}
\]

What about the following function?

\[
\begin{align*}
g_3 &::? (01) \to (1) \\
g_3 & = \text{pack}_{(1)} \circ g \circ \text{unpack}
\end{align*}
\]

It is wrong, since it breaks its contract at the first time step:

\[
g_3 ([].\perp) = \perp
\]
From Synchronization to Desynchronization

\[ g_1 : (1) \rightarrow (2) \]

\[ desync_{(1) \rightarrow (2)} \]

\[ g : Stream(N) \rightarrow Stream(N) \]

\[ sync_{(1) \rightarrow (2)} \]

\[ desync_{(1 \ 0) \rightarrow (1)} \]

\[ g_2 : (1 \ 0) \rightarrow (1) \]

\[ sync_{(1 \ 0) \rightarrow (1)} \]
A buffer shifts the values of a clocked stream to the left:

| x :: (10) | [a] [] [b] [] [c] [] ... |
| --- | --- | --- | --- | --- | --- |
| x' :: (01) | [] [a] [] [b] [] [c] ... |

The relation $w <_{k} w'$ models a buffer with producer $w$, consumer $w'$ and $k$ steps of delay. For example:

- $(10) <_{1} (01)$

$(10)$ 1 0 1 0 1 0 1 0 ...

$(01)$ 0 1 0 1 0 1 0 1 ...

- $(101) <_{0} (011)$ but not $(101) <_{1} (011)$

$(10)$ 1 0 1 1 1 0 1 0 1 ...

$(011)$ 0 1 1 0 1 1 1 ...
Now, given a function $h :: w_1 \rightarrow w_2$, we may put a buffer on its…

- **Output:** if $w_2 <: k w'_2$, we define

  $$h' :: w_1 \rightarrow w'_2$$
  $$h' = buffer_{w_2 <: k w'_2} \circ h$$

  For example:

  $$(1) \rightarrow (10) <: (1) \rightarrow (01)$$

- **Input:** if $w'_1 <: k w_1$, we define

  $$h'' :: w'_1 \rightarrow w_2$$
  $$h'' = h \circ buffer_{w'_1 <: k w_1}$$

  For example:

  $$(01) \rightarrow (1) <: (10) \rightarrow (1)$$
Given a function \( h :: w_1 \rightarrow w_2 \), is it safe to compute \( x = h x \)?

What about...

\[
\begin{align*}
h_1 &:: (1) \rightarrow (1) \quad \text{KO} \\
h_2 &:: (01) \rightarrow (10) \quad \text{OK} \\
h_3 &:: (011) \rightarrow (101) \quad \text{KO}
\end{align*}
\]

We allow feedback only when \( w_2 <:_1 w_1 \).
This makes sure that \( x = h x \) is total.
Part II
Recap of Part I

In part I, we saw...

- How the compilation of Lustre-like languages can be seen as making stream functions length-preserving by cheating with (co-)domains:

  \[
  \text{from } \text{Stream}(\mathbb{N}) \rightarrow \text{Stream}(\mathbb{N}) \quad \text{to } \text{Stream}(\text{List}(\mathbb{N})) \rightarrow \text{Stream}(\text{List}(\mathbb{N}))
  \]

- How these way of making functions length-preserving can be characterized by the sizes of the lists

- How you could play with some operations on stream functions, such as buffering and feedback loops.

Now we turn to the description of local time scales.
Take any function $f$ implemented by state machine $m$, with

$$f :: (10) \rightarrow (01)$$

We can transform $f$ into $f'$ such that

$$f' :: (1) \rightarrow (1)$$

What would be $m'$, the implementation of $f'$?

- A single transition of $m'$ performs two transitions of $m$
- We write

$$(10) \rightarrow (01) \uparrow_{(2)} (1) \rightarrow (1)$$
A local time scale comes with a clock $w$ driving its internal time

- E.g. $(21)$ begins with two internal steps for one external, etc.

How does the inside sees the outside? The converse?

- $w_1 \rightarrow w_2 \uparrow w_1' \rightarrow w_2'$: leaving local time

\[
\begin{align*}
(101) &\rightarrow (011) \uparrow (21) (1) \rightarrow (1) \quad \text{OK} \\
(011) &\rightarrow (101) \uparrow (21) (1) \rightarrow (1) \quad \text{OK}
\end{align*}
\]

- $w_1 \rightarrow w_2 \downarrow w_1' \rightarrow w_2'$: entering local time

\[
\begin{align*}
(1) &\rightarrow (1) \downarrow (21) (101) \rightarrow (011) \quad \text{OK} \\
(1) &\rightarrow (1) \downarrow (21) (011) \rightarrow (101) \quad \text{KO}
\end{align*}
\]
Consider two simple examples:

\[(10) \uparrow_{(2)} (1)\]

What is the action of (2) on (10) that gives (1)?

Let us define clock composition as

\[
\text{Stream}(\mathbb{N}) \times \text{Stream}(\mathbb{N}) \to \text{Stream}(\mathbb{N})
\]

\[
(n \cdot w) \text{ on } (m_1 \ldots m_n \cdot w') = \left( \sum_{1 \leq i \leq n} m_i \right) \cdot (w \text{ on } w')
\]

We can now define:

\[w_1 \uparrow_w w_2 \iff w \text{ on } w_1 = w_2\]

Similarly, \((1) \downarrow_{(2)} (01)\) because \((1) = (2) \text{ on } (10)\).
Going back to our first example: $(10) \circ (01) \uparrow_{(2)} (1) \circ (1)$. Why?

Because we have $(1) \downarrow_{(2)} (10)$

and $(01) \uparrow_{(2)} (1)$

This suggests the reasoning principle

\[
\begin{array}{c}
W'_1 \downarrow_w W_1 & \quad & W_2 \uparrow_w W'_2 \\
\hline \\
W_1 \circ W_2 & \uparrow_w & W'_1 \circ W'_2
\end{array}
\]

More complex principles can be found for $W_1 \circ W_2 \downarrow_w W'_1 \circ W'_2$
Putting it all together (1/2)

Take \( f(x, y) = (0.y, x) \). Is the smallest fixpoint of \( f \) total? Why?

This problem is equivalent to the scheduling of this Lustre code:

\[
\begin{align*}
x &= 0 \text{ fby } y \\
y &= x
\end{align*}
\]

Consider the signature below:

\[
f :: (01) \otimes 0(01) \rightarrow (10) \otimes (01)
\]

It mimics the growth of partial streams in \( \text{lfp } f = \bigsqcup_{i \geq 0} (f^i \bot) \):

<table>
<thead>
<tr>
<th>( x )</th>
<th>( f \times )</th>
</tr>
</thead>
<tbody>
<tr>
<td>((\bot, \bot))</td>
<td>((0.\bot, \bot))</td>
</tr>
<tr>
<td>((0.\bot, \bot))</td>
<td>((0.\bot, 0.\bot))</td>
</tr>
<tr>
<td>((0.\bot, 0.\bot))</td>
<td>((0.0.\bot, 0.\bot))</td>
</tr>
<tr>
<td>((0.0.\bot, 0.\bot))</td>
<td>((0.0.\bot, 0.0.\bot))</td>
</tr>
<tr>
<td>( \ldots )</td>
<td>( \ldots )</td>
</tr>
</tbody>
</table>
Putting it all together (2/2)

So, with $f :: (01) \otimes 0(01) \rightarrow (10) \otimes (01)$, since

$$(10) <:_1 (01)$$
$$(01) <:_1 0(01)$$

we know that the fixpoint is total, and get

$$\text{lfp } f :: (10) \otimes (01)$$

Now, we can wrap it into a local time scale going twice faster

$$(10) \otimes (01) \uparrow_{(2)} (1) \otimes (1)$$

Interestingly, something happens to the internal buffers

<table>
<thead>
<tr>
<th>Inside view</th>
<th>Outside view</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(10) &lt;:_1 (01)$</td>
<td>$(1) &lt;:_0 (1)$</td>
</tr>
<tr>
<td>$(01) &lt;:_1 0(01)$</td>
<td>$(1) &lt;:_1 0(1)$</td>
</tr>
</tbody>
</table>
e ::= x
| λx.e
| e e
| (e, e)
| let (x, x) = e in e
| fix e
| c
| op e
| merge p e e
| e when p
p ::= c∗(c+)

t ::= dt :: ct
| t ⊗ t
| t → t
| dt ::= bool | int | ...
t ⊗ t
| t ⊸ t
c
| ct ::= p
| ct on ct
Γ ::= □
| Γ, x : t
Typing Buffers

\[
\frac{
\begin{align*}
\text{Sub} & \quad \Gamma \vdash e : t \quad \Gamma \vdash t < : k \ t' \\
\end{align*}
}{
\quad \Gamma \vdash e : t'
}
\quad \frac{
\begin{align*}
\text{AdaptFun} & \quad t'_1 < : k_1 \ t_1 \quad t_2 < : k_2 \ t'_2 \\
\end{align*}
}{
\quad t_1 \circ t_2 < : 0 \ t'_1 \circ t'_2
}\]
Typing Feedback

\[
\begin{align*}
\text{Fix} & \\
\Gamma \vdash e : t \rightarrow t' & \quad \vdash t' <_{1} t & \quad \vdash t' \text{ value} \\
\hline
\Gamma \vdash \text{fix} e : t'
\end{align*}
\]
Typing Local Time Scales

\[
\begin{align*}
\text{Scale} & \\
\vdash \Gamma \Downarrow_{ct} \Gamma' & \quad \Gamma' \vdash e : t' & \quad \vdash t' \Uparrow_{ct} t \\
\hline
\Gamma \vdash e : t
\end{align*}
\]
Soundness and Realizability

Two semantics: unclocked $K[\_\_]$ and clocked $S[\_\_]$, e.g.

$K[\vdash e : \text{int} :: ct \rightarrow \text{int} :: ct] : \text{Stream}(\mathbb{N}) \rightarrow \text{Stream}(\mathbb{N})$

$S[\vdash e : \text{int} :: ct \rightarrow \text{int} :: ct] : \text{Stream}(\text{List}(\mathbb{N})) \rightarrow \text{Stream}(\text{List}(\mathbb{N}))$

Soundness theorem

The statics (typing) and dynamics (semantics) agree:

$\forall e, dt, ct, \text{clock} \ S[\vdash e : dt :: ct] = [ct]$

Some interesting, more or less direct corollaries:

- The clocked semantics is causal
  
  $\forall e, dt, ct, S[\vdash e : dt :: ct] \text{ is total}$

- Synchronizing the unclocked semantics gives the clocked one
  
  $\forall e, t, S[\vdash e : t] = synct \ K[\vdash e : t]$
Soundness proof (1/2)

- First, define the set of realizers of some type $t$:

  \[
  \mathcal{W}_t \subseteq S[t] \\
  \mathcal{W}_{dt :: ct} = \{xs \mid \text{clock } xs = \llbracket ct \rrbracket\} \\
  \mathcal{W}_{t_1 \otimes t_2} = \mathcal{W}_{t_1} \times \mathcal{W}_{t_2} \\
  \mathcal{W}_{t \rightarrow t'} = \{f \mid \forall x \in \mathcal{W}_t, (f \ x) \in \mathcal{W}_{t'}\} \\
  \mathcal{W}_\Gamma \subseteq S[\Gamma] \\
  \ldots
  \]

- The soundness theorem then becomes a corollary of the adequacy lemma: for all $\Gamma$, $e$ and $t$, we have

  \[
  \forall \gamma \in \mathcal{W}_\Gamma, (S[\Gamma \vdash e : t \rrbracket \gamma) \in \mathcal{W}_t
  \]

- Unfortunately, it does not work!
Soundness proof (2/2)

- The proof attempt fails on fixpoints: we need information on partial streams.
- Let us refine realizers as follows:

\[
\begin{align*}
\mathcal{W}_n &\subseteq S[t] \\
\mathcal{W}_{dt :: ct}^n &= \{xs \mid \text{clock } xs =_n S[ct]\} \\
\mathcal{W}_{t_1 \otimes t_2}^n &= \mathcal{W}_{t_1}^n \times \mathcal{W}_{t_2}^n \\
\mathcal{W}_{t \rightarrow t'}^n &= \{f \mid \forall m \leq n, \forall x \in \mathcal{W}_t^m, (f \ x) \in \mathcal{W}_{t'}^m\} \\
\mathcal{W}_\Gamma^n &\subseteq S[\Gamma]
\end{align*}
\]

- And restate the adequacy lemma:

\[
\forall n \in \mathbb{N}, \forall \gamma \in \mathcal{W}_\Gamma^n, (S[\Gamma \vdash e : t] \gamma) \in \mathcal{W}_t^n
\]

- An essential lemma for fixpoints:

\[
\forall t, t', \forall k, n \in \mathbb{N}, \forall xs \in \mathcal{W}_t^n, (S[\vdash t <:_k t'] xs) \in \mathcal{W}_{t'}^{n+k}
\]
Related work and Inspiration

- Lustre (Caspi, Halbwachs et al.)
  - General conceptual setting
- Lucid Synchrone (Caspi, Pouzet et al.)
  - Clocks as types
  - Separate compilation
- Lucy-n (Mandel, Plateau, Pouzet)
  - Buffers, adaptability
  - Ultimately periodic clocks
- Clock Domains in ReactiveML (Mandel, Pasteur)
  - Local time scales
- Geometry of Synthesis, Verity (Ghica)
  - Linear HOFs to circuits via $\mathbf{G()}$ (from Abramsky, Girard)
- Cyclic Scheduling of *DFs (Lee, Munier-Kordon, etc.)
  - Algorithms for type inference with periodic clocks
Conclusion and Perspectives

- A setting for unified clocking / initialization / causality analysis
  - The full type system is not overly complex
  - Local time scales important for modularity
  - No need for a scheduling pass after typing

- Relies on standard programming language theory
  - Denotational Semantics, Types, Realizability
  - Realizability is a powerful tool. Too powerful?

- Lots of remaining questions
  - Theoretical: principality, better semantic setting, full abstraction
  - Practical: type inference, optimizations, parallel code generation

Thank you!
Bonus Slides
\[
\text{DownArrowBin}
\]
\[
\Gamma \vdash t'_1 \uparrow_{ct} t_1 \quad \Gamma \vdash t_2 \downarrow_{ct} t'_2 \quad ct \leq (1)
\]
\[
\Gamma \vdash t_1 \leadsto t_2 \downarrow_{ct} t'_1 \leadsto t'_2
\]

\[
\text{DownArrowPos}
\]
\[
\Gamma \vdash t'_1 \downarrow_{ct'} t_1 \quad \Gamma \vdash t_2 \uparrow_{ct'} t'_2 \quad ct \text{ on } ct' = (1)
\]
\[
\Gamma \vdash t_1 \leadsto t_2 \downarrow_{ct} t'_1 \leadsto t'_2
\]

\[
\text{DownOn}
\]
\[
\Gamma \vdash t \downarrow_{ct} t'' \quad \Gamma \vdash t'' \downarrow_{ct'} t'
\]
\[
\Gamma \vdash t \downarrow_{ct \text{ on } ct'} t'
\]