Integer Clocks and Local Time Scales
Part I – Part II

Adrien Guatto
ENS - PARKAS
SYNCHRON 2014
Part I
Programming Languages for Reactive Systems

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!
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 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 $\text{Stream}(\mathbb{B})$, ordered by *information*:
Consider the following function

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

\[
f(x.xs) = (x + 1).(f xs)
\]

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

\[
m : \text{M} (\mathbb{N}, \mathbb{N})
\]

\[
m = (\{\star\}, \star, \lambda(\star, x).(\star, x + 1))
\]

The machine \( m \) processes one element per transition. It was easy since the function is length-preserving.
Stream Functions (2/2)

What about the following function?

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

\[ g(x.xs) = (x + 1).(x - 1).(g \; xs) \]

Yes, if we cheat a bit.

\[ m_1 : M (\mathbb{N}, \text{List}(\mathbb{N})) \]

\[ m_1 = (\{\ast\}, \ast, \lambda (\ast, x). (\ast, [x + 1; x - 1])) \]

Another possibility:

\[ m_2 : M (\text{List}(\mathbb{N}), \mathbb{N}) \]

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

$$g : \text{Stream}({\mathbb{N}}) \rightarrow \text{Stream}({\mathbb{N}})$$

$$g (x.xs) = (x + 1).(x - 1).(g xs)$$

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

$$g_1 : \text{Stream}(\text{List}({\mathbb{N}})) \rightarrow \text{Stream}(\text{List}({\mathbb{N}}))$$

$$g_1 ([x].xs) = [x + 1; x - 1].(g_1 xs)$$

$$g_2 : \text{Stream}(\text{List}({\mathbb{N}})) \rightarrow \text{Stream}(\text{List}({\mathbb{N}}))$$

$$g_2 ([x].xs) = [x + 1].(\text{let } [] .xs' = xs \text{ in}
[x - 1].(g_2 xs'))$$

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, \( \text{clocks} \), 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 $w \in \text{Stream}(\mathbb{N})$?

For example:

<table>
<thead>
<tr>
<th>$x = \text{pack}_{1(10)}(a,b,c,d\ldots)$</th>
<th>$[a]$</th>
<th>$[b]$</th>
<th>$[]$</th>
<th>$[c]$</th>
<th>$[]$</th>
<th>$\ldots$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$y = \text{pack}_{(02)}(a,b,c,d\ldots)$</td>
<td>$[]$</td>
<td>$[a;b]$</td>
<td>$[]$</td>
<td>$[c;d]$</td>
<td>$[]$</td>
<td>$\ldots$</td>
</tr>
</tbody>
</table>

Obviously:

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

$$g_1 :: (1) \to (2)$$
$$g_1 = \text{pack}_{(2)} \circ g \circ \text{unpack}$$
$$g_2 :: (10) \to (1)$$
$$g_2 = \text{pack}_{(1)} \circ g \circ \text{unpack}$$

What about the following function?

$$g_3 ::? (01) \to (1)$$
$$g_3 = \text{pack}_{(1)} \circ g \circ \text{unpack}$$

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

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

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

\[ g_2 : (10) \rightarrow (1) \]

\[ \text{sync}_{(1) \rightarrow (2)} \]

\[ \text{desync}_{(1) \rightarrow (2)} \]

\[ g : \text{Stream}(\mathbb{N}) \rightarrow \text{Stream}(\mathbb{N}) \]
A buffer shifts the values of a clocked stream to the left:

<table>
<thead>
<tr>
<th>x :: (10)</th>
<th>[a] [b] [c] ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>x' :: (01)</td>
<td>[a] [b] [c] ...</td>
</tr>
</tbody>
</table>

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

- $(10) <:^1 (01)$

  \[
  (10) \begin{array}{cccccccc}
  1 & 0 & 1 & 0 & 1 & 0 & 1 & \ldots \\
  \end{array}
  \]

  \[
  (01) \begin{array}{cccccccc}
  0 & 1 & 0 & 1 & 0 & 1 & \ldots \\
  \end{array}
  \]

- $(101) <:^0 (011)$ but not $(101) <:^1 (011)$

  \[
  (10) \begin{array}{cccccccc}
  1 & 0 & 1 & 1 & 0 & 1 & \ldots \\
  \end{array}
  \]

  \[
  (011) \begin{array}{cccccccc}
  0 & 1 & 1 & 0 & 1 & 1 & \ldots \\
  \end{array}
  \]
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

  \[
  h' : w_1 \rightarrow w'_2 \\
  h' = \text{buffer}_{w_2 <_{k} w'_2} \circ h
  \]

  For example:

  \[
  (1) \rightarrow (10) <_{(1)} (1) \rightarrow (01)
  \]

- **Input:** if \( w'_1 <_{k} w_1 \), we define

  \[
  h'' : w'_1 \rightarrow w_2 \\
  h'' = h \circ \text{buffer}_{w'_1 <_{k} w_1}
  \]

  For example:

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

What about... 

$$h_1 :: (1) \rightarrow (1) \quad \text{KO}$$

$$h_2 :: (01) \rightarrow (10) \quad \text{OK}$$

$$h_3 :: (011) \rightarrow (101) \quad \text{KO}$$

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:

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

- 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. $(2 1)$ begins with two internal steps for one external, etc.

How does the inside sees the outside? The converse?

- $w_1 \rightarrow w_2 \uparrow_w w'_1 \rightarrow w'_2$: leaving local time

  $$
  \begin{align*}
  (1 0 1) & \rightarrow (0 1 1) \quad \uparrow_{(2 1)} \quad (1) \rightarrow (1) & \text{OK} \\
  (0 1 1) & \rightarrow (1 0 1) \quad \uparrow_{(2 1)} \quad (1) \rightarrow (1) & \text{OK}
  \end{align*}
  $$

- $w_1 \rightarrow w_2 \downarrow_w w'_1 \rightarrow w'_2$: entering local time

  $$
  \begin{align*}
  (1) & \rightarrow (1) \quad \downarrow_{(2 1)} \quad (1 0 1) \rightarrow (0 1 1) & \text{OK} \\
  (1) & \rightarrow (1) \quad \downarrow_{(2 1)} \quad (0 1 1) \rightarrow (1 0 1) & \text{KO}
  \end{align*}
  $$
Scatter/Gather: Streams

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

\[
\text{on} : \text{Stream}(\mathbb{N}) \times \text{Stream}(\mathbb{N}) \rightarrow \text{Stream}(\mathbb{N})
\]

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

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: \((10) \rightarrow (01) \uparrow_{(2)} (1) \rightarrow (1)\). Why?

Because we have \((1) \downarrow_{(2)} (10)\) and \((01) \uparrow_{(2)} (1)\)

This suggests the reasoning principle

\[
\begin{align*}
    w'_1 \downarrow_{w} w_1 & \quad w_2 \uparrow_{w} w'_2 \\
    w_1 & \rightarrow w_2 \uparrow_{w} w'_1 & \rightarrow w'_2
\end{align*}
\]

More complex principles can be found for \(w_1 \rightarrow w_2 \downarrow_{w} w'_1 \rightarrow w'_2\)
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:

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

\[
\begin{align*}
f \text{by } y 
\end{align*}
\]

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

Consider the signature below:

\[
f :: (0 \uplus 0) \otimes 0 \rightarrow (1 \uplus 0) \otimes (0 \uplus 1)
\]

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

<table>
<thead>
<tr>
<th>( x )</th>
<th>( f \ x )</th>
</tr>
</thead>
<tbody>
<tr>
<td>((\perp, \perp))</td>
<td>((0.\perp, \perp))</td>
</tr>
<tr>
<td>((0.\perp, \perp))</td>
<td>((0.\perp, 0.\perp))</td>
</tr>
<tr>
<td>((0.\perp, 0.\perp))</td>
<td>((0.0.\perp, 0.\perp))</td>
</tr>
<tr>
<td>((0.0.\perp, 0.\perp))</td>
<td>((0.0.\perp, 0.0.\perp))</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
So, with \( f :: (0\ 1) \otimes 0(0\ 1) \rightarrowtail (1\ 0) \otimes (0\ 1) \), since

\[
(1\ 0) <_{\!1} (0\ 1) \\
(0\ 1) <_{\!1} 0(0\ 1)
\]

we know that the fixpoint is total, and get

\[
\text{lf}p\ f :: (1\ 0) \otimes (0\ 1)
\]

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

\[
(1\ 0) \otimes (0\ 1) \uparrow_{\!(2)} (1) \otimes (1)
\]

Interestingly, something happens to the internal buffers

<table>
<thead>
<tr>
<th>Inside view</th>
<th>Outside view</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1\ 0) &lt;_{!1} (0\ 1)</td>
<td>(1) &lt;_{!0} (1)</td>
</tr>
<tr>
<td>(0\ 1) &lt;_{!1} 0(0\ 1)</td>
<td>(1) &lt;_{!1} 0(1)</td>
</tr>
</tbody>
</table>

Wire

Memory
\[ e ::= x \]
\[ \lambda x.e \]
\[ e e \]
\[ (e, e) \]
\[ \text{let } (x, x) = e \text{ in } e \]
\[ \text{fix } e \]
\[ c \]
\[ \text{op } e \]
\[ \text{merge } p \ e \ e \]
\[ e \text{ when } p \]
\[ p ::= c^* (c^+) \]

\[ t ::= dt :: ct \]
\[ t \otimes t \]
\[ t \rightarrow t \]
\[ dt ::= \text{bool} \mid \text{int} \mid \ldots \]
\[ ct ::= p \]
\[ ct \text{ on } ct \]
\[ \Gamma ::= \square \]
\[ \Gamma, x : t \]
Typing Buffers

$$\text{Sub} \quad \Gamma \vdash e : t \quad t <: _k t^0$$

$$\quad \Gamma \vdash e : t^0$$

$$\text{AdaptFun} \quad t_1^0 <: _{k_1} t_1 \quad t_2 <: _{k_2} t_2^0$$

$$\quad t_1 \leadsto t_2 <: _0 t_1^0 \leadsto t_2^0$$
\[
\text{Fix} \\
\Gamma ` \ e : t \rightarrow t^0 \quad ` t^0 <:_1 t \quad ` t^0 \text{value} \\
\underline{\Gamma ` \ \textbf{fix} \ e : t^0}
\]
Typing Local Time Scales

\[
\begin{align*}
\text{Scale} & \quad \Gamma \downarrow_{ct} \Gamma^0 \quad \Gamma^0 \vdash e : t^0 \quad t^0 \uparrow_{ct} t \\
\hline
\Gamma \vdash e : t
\end{align*}
\]
Soundness and Realizability

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

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

Soundness theorem

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

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

Some interesting, more or less direct corollaries:

- The clocked semantics is causal
  \[
  \forall e, dt, ct, S[\`\ ` e : dt :: ct] \text{ is total}
  \]
- Synchronizing the unclocked semantics gives the clocked one
  \[
  \forall e, t, S[\`\ ` e : t] = \text{sync}_t K[\`\ ` e : t]
  \]
Soundness proof (1/2)

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

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

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

\[
\forall \gamma \in W_{\Gamma}, (S[\llbracket \Gamma \setminus e : t \rrbracket \gamma]) \in W_t
\]

- 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:

\[ \mathbb{W}_{t^\mathbb{N}} \subseteq S[t] \]
\[ \mathbb{W}^n_{dt :: ct} = \{xs \mid \text{clock } xs =_n S[ct] \} \]
\[ \mathbb{W}^n_{t_1 \otimes t_2} = \mathbb{W}^n_{t_1} \times \mathbb{W}^n_{t_2} \]
\[ \mathbb{W}^n_{t \rightarrow t'} = \{f \mid \forall m \leq n, \forall x \in \mathbb{W}^m_t, (f \ x) \in \mathbb{W}^m_{t'} \} \]
\[ \mathbb{W}_{\Gamma^\mathbb{N}} \subseteq S[\Gamma] \]

... 

- And restate the adequacy lemma:

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

- An essential lemma for fixpoints:

\[ \forall t, t', k, n \in \mathbb{N}, \forall xs \in \mathbb{W}^n_t, (S[^<:k t \ ` t'] xs) \in \mathbb{W}^{n+k}_{t'} \]
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 $G()$ (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
\[
\begin{align*}
\text{DownArrowBin} & \quad \begin{array}{c}
\text{``} t_1' \overset{\text{ct}}{\uparrow} t_1 \quad \text{``} t_2 \overset{\text{ct}}{\downarrow} t_2' \quad \text{ct} \leq (1) \\
\text{``} t_1 \overset{\circ}{\rightarrow} t_2 \quad \text{``} t_1' \overset{\circ}{\rightarrow} t_2'
\end{array} \\
\text{DownArrowPos} & \quad \begin{array}{c}
\text{``} t_1' \overset{\text{ct'}}{\downarrow} t_1 \quad \text{``} t_2 \overset{\text{ct'}}{\uparrow} t_2' \quad \text{ct on ct'} = (1) \\
\text{``} t_1 \overset{\circ}{\rightarrow} t_2 \quad \text{``} t_1' \overset{\circ}{\rightarrow} t_2'
\end{array} \\
\text{DownOn} & \quad \begin{array}{c}
\text{``} t \overset{\text{ct}}{\downarrow} t'' \quad \text{``} t'' \overset{\text{ct'}}{\downarrow} t'
\end{array} \quad \text{``} t \overset{\text{ct on ct'}}{\downarrow} t'
\end{align*}
\]