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 $\text{Stream}(\mathbb{B})$, ordered by *information*:

```
         ⊥
        / \    /   \
   0.0. ⊥ 0.1. ⊥ 1.0. ⊥ 1.1. ⊥
     /     /           /     /
  0. ⊥ 1. ⊥ 0. ⊥ 1. ⊥
   /   /     /     /   /
⊥ ⊥ ⊥ ⊥
```
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 length-preserving.
Stream Functions (2/2)

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.

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

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 $g_2$?

\[
g : \text{Stream}(\mathbb{N}) \to \text{Stream}(\mathbb{N})
g_1 : \text{Stream}(	ext{List}(\mathbb{N})) \to \text{Stream}(	ext{List}(\mathbb{N}))
g_2 : \text{Stream}(	ext{List}(\mathbb{N})) \to \text{Stream}(	ext{List}(\mathbb{N}))
\]

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, clocks, fully characterize $g_1$ and $g_2$. We write:

\[
g_1 :: (1) \leadsto (2)
g_2 :: (1\ 0) \leadsto (1)
\]
A clock $w$ is just a stream of integers!
What can we do with such a $w \in \text{Stream}(\mathbb{N})$?

For example:

| $x = \text{pack}_{1(10)} \omega (a.b.c.d \ldots)$ | $[a] [b] [] [c] [] \ldots$ |
| $y = \text{pack}_{(02)} \omega (a.b.c.d \ldots)$ | $[] [a;b] [] [c;d] [] \ldots$ |

Obviously:

$$\text{unpack } x = \text{unpack } y$$
Synchronous Stream Functions

We now define the functions $g_1$ and $g_2$ purely from their clocks:

$$
g_1 :: (1) \rightarrow (2)$$
$$g_1 = pack_{(2)} \circ g \circ unpack$$

$$
g_2 :: (10) \rightarrow (1)$$
$$g_2 = pack_{(1)} \circ g \circ unpack$$

What about the following function?

$$
g_3 ::? (01) \rightarrow (1)$$
$$g_3 = pack_{(1)} \circ g \circ unpack$$

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

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

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

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

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

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

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

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

\[
g_2 : (1 \ 0) \rightarrow (1)
\]
Playing with Synchronous Functions: Buffers (1/2)

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:

- $(10) <:^1 (01)$
  
  $$(10) \quad 1 \quad 0 \quad 1 \quad 0 \quad 1 \quad 0 \quad ...$$

  $$(01) \quad 0 \quad 1 \quad 0 \quad 1 \quad 0 \quad 1 \quad ...$$

- $(101) <:^0 (011)$ but not $(101) <:^1 (011)$
  
  $$(10) \quad 1 \quad 0 \quad 1 \quad 1 \quad 1 \quad 0 \quad 1 \quad ...$$

  $$(011) \quad 0 \quad 1 \quad 1 \quad 0 \quad 1 \quad 1 \quad ...$$
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:

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

- **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:

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

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

- 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

$$(10) \rightarrow (01) \uparrow (2) (1) \rightarrow (1)$$
Local Time Scales and Scatter/Gather

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?

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

  $\begin{align*}
  (101) &\rightarrow (011) &\uparrow_{(21)} (1) &\rightarrow (1) &\text{OK} \\
  (011) &\rightarrow (101) &\uparrow_{(21)} (1) &\rightarrow (1) &\text{OK}
  \end{align*}$

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

  $\begin{align*}
  (1) &\rightarrow (1) &\downarrow_{(21)} (101) &\rightarrow (011) &\text{OK} \\
  (1) &\rightarrow (1) &\downarrow_{(21)} (011) &\rightarrow (101) &\text{KO}
  \end{align*}$
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

\[
\begin{align*}
\operatorname{on} & : \text{Stream}(\mathbb{N}) \times \text{Stream}(\mathbb{N}) \to \text{Stream}(\mathbb{N}) \\
(n.w) \operatorname{on} (m_1 \ldots m_n.w') & = (\sum_{1 \leq i \leq n} m_i).w w'
\end{align*}
\]

We can now define:

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

Similarly, \((1) \downarrow_{(2)} (01)\) because \((1) = (2) \operatorname{on} (10)\)
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

\[
\frac{w'_1 \downarrow_w w_1 \quad w_2 \uparrow_w w'_2}{w_1 \rightarrow w_2 \uparrow_w w'_1 \rightarrow w'_2}
\]

More complex principles can be found for $w_1 \rightarrow w_2 \downarrow w \quad w'_1 \rightarrow w'_2$
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:

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

Consider the signature below:

$$f :: (01) \otimes 0(01) \rightarrow (10) \otimes (01)$$

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

<table>
<thead>
<tr>
<th>$x$</th>
<th>$f \times$</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>
Putting it all together (2/2)

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

\[
\begin{align*}
(10) &<_{1} (01) \\
(01) &<_{1} 0(01)
\end{align*}
\]

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

Inside view \hspace{1cm} Outside view

\[
\begin{align*}
(10) &<_{1} (01) \\
(01) &<_{1} 0(01)
\end{align*} \hspace{1cm} \begin{align*}
(1) &<_{0} (1) \\
(1) &<_{1} 0(1)
\end{align*}
\]

Wire \hspace{1cm} Memory
From Semantics to Syntax

\[
\begin{align*}
  e & ::= x \\
  & | \lambda x. e \\
  & | e e \\
  & | (e, e) \\
  & | \text{let } (x, x) = e \text{ in } e \\
  & | \text{fix } e \\
  & | c \\
  & | \text{op } e \\
  & | \text{merge } p \ e \ e \\
  & | e \text{ when } p \\
  & | p \\
  \end{align*}
\]

\[
\begin{align*}
  t & ::= dt :: ct \\
  & | t \otimes t \\
  & | t \to t \\
  \end{align*}
\]

\[
\begin{align*}
  dt & ::= \text{bool} \mid \text{int} \mid \ldots \\
  ct & ::= p \\
  & | ct \text{ on } ct \\
  \end{align*}
\]

\[
\begin{align*}
  \Gamma & ::= \square \\
  & | \Gamma, x : t \\
  \end{align*}
\]
Typing Buffers

\[
\begin{align*}
\text{Sub} & \quad \Gamma \vdash e : t \quad \vdash t <:_k t' \\
& \quad \Gamma \vdash e : t'
\end{align*}
\]

\[
\begin{align*}
\text{AdaptFun} & \quad t'_1 <:_{k_1} t_1 \quad t_2 <:_{k_2} t'_2 \\
& \quad t_1 \rightarrow t_2 <:_0 t'_1 \rightarrow t'_2
\end{align*}
\]
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

\[ \Gamma \vdash e : t \]

\[ \Gamma' \vdash e : t' \]

\[ t' \uparrow_{ct} t \]

\[ \Gamma \vdash e : t \]
Soundness and Realizability

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

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

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

Soundness theorem

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

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

Some interesting, more or less direct corollaries:

- The clocked semantics is causal

  $\forall e, dt, ct, \mathcal{S}[\vdash e : dt :: ct] \text{ is total}$

- Synchronizing the unclocked semantics gives the clocked one

  $\forall e, t, \mathcal{S}[\vdash e : t] = \text{sync}_t \mathcal{K}[\vdash e : t]$
Soundness proof (1/2)

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

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

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

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

- Unfortunately, it does not work!
The proof attempt fails on fixpoints: we need information on partial streams.

Let us refine realizers as follows:

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

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:

\[ \forall t, t', \forall k, n \in \mathbb{N}, \forall xs \in \mathcal{W}_t^n, (S[\Gamma \vdash t <: k t'] xs) \in \mathcal{W}_{t'}^{n+k} \]
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 \( G() \) (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
\[ \text{DownArrowBin} \]

\[
\begin{align*}
\vdash t_1' \uparrow_{ct} t_1 & \quad \vdash t_2 \downarrow_{ct} t_2' & \quad ct \leq (1) \\
\vdash t_1 \rightarrow t_2 \downarrow_{ct} t_1' \rightarrow t_2' 
\end{align*}
\]

\[ \text{DownArrowPos} \]

\[
\begin{align*}
\vdash t_1' \downarrow_{ct'} t_1 & \quad \vdash t_2 \uparrow_{ct'} t_2' & \quad ct \text{ on } ct' = (1) \\
\vdash t_1 \rightarrow t_2 \downarrow_{ct} t_1' \rightarrow t_2' 
\end{align*}
\]

\[ \text{DownOn} \]

\[
\begin{align*}
\vdash t \downarrow_{ct} t'' & \quad \vdash t'' \downarrow_{ct'} t' \\
\vdash t \downarrow_{ct \text{ on } ct'} t' 
\end{align*}
\]