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

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

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

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

What about the following function?

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

Yes, if we cheat a bit.

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

Another possibility:

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

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

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

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

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

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

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

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

Functions $g_1$ and $g_2$ are length-preserving.
How to describe the relationship between $g$, $g_1$ and $g_2$?

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

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:

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

For example:
\[
\begin{array}{c}
\text{x} = \text{pack}_{1(10)}(a.b.c.d \ldots) & \Rightarrow & [a] [b] [] [c] [] \ldots \\
\text{y} = \text{pack}_{(02)}(a.b.c.d \ldots) & \Rightarrow & [] [a; b] [] [c; d] [] \ldots \\
\end{array}
\]

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

\[
g_1 :: (1) \xrightarrow{} (2) \\
g_1 = \text{pack}(2) \circ g \circ \text{unpack}
\]

\[
g_2 :: (10) \xrightarrow{} (1) \\
g_2 = \text{pack}(1) \circ g \circ \text{unpack}
\]

What about the following function?

\[
g_3 ::? (01) \xrightarrow{} (1) \\
g_3 = \text{pack}(1) \circ g \circ \text{unpack}
\]

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

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

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

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

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

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

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

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

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

A buffer shifts the values of a clocked stream to the left:

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

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

- $(10) <_{1} (01)$
  
  $$(10) \quad 1 \quad 0 \quad 1 \quad 0 \quad 1 \quad 0 \quad 1 \quad 0 \quad 1 \quad \ldots$$  
  
  $$(01) \quad 0 \quad 1 \quad 0 \quad 1 \quad 0 \quad 1 \quad \ldots$$  
  
- $(101) <_{0} (011)$ but not $(101) <_{1} (011)$
  
  $$(10) \quad 1 \quad 0 \quad 1 \quad 1 \quad 0 \quad 1 \quad \ldots$$  
  
  $$(011) \quad 0 \quad 1 \quad 1 \quad 0 \quad 1 \quad 1 \quad \ldots$$
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 $x = h \cdot x$?

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 $x = h \cdot x$ is total.
Part II
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:

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

- 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

$$f :: (1\ 0) \rightarrow (0\ 1)$$

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

$$ (1\ 0) \rightarrow (0\ 1) \uparrow_{(2)} (1) \rightarrow (1) $$
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?

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

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

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

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

More complex principles can be found for \(w_1 \rightarrow w_2 \downarrow_w w_1' \rightarrow w_2'\)
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:

$x = 0 \text{ fby } y$

$y = x$

Consider the signature below:

$f :: (0\ 1) \otimes 0(0\ 1) \rightarrow (1\ 0) \otimes (0\ 1)$

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

<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>$(0.0.0.\perp, 0.0.\perp)$</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
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

$$lfp \; f :: (10) \otimes (01)$$

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

$$\downarrow_{(2)} (1) \otimes (1)$$

Interestingly, something happens to the internal buffers

<table>
<thead>
<tr>
<th>Inside view</th>
<th>Outside view</th>
<th>Wire</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>(10) &lt;:_{1} (01)</td>
<td>(1) &lt;:_{0} (1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(01) &lt;:_{1} 0(01)</td>
<td>(1) &lt;:_{1} 0(1)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
e ::= x | \lambda x.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 | ... |

\[ \Gamma ::= □ \] \\
| □ | □, x : t
Typing Buffers

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

\[ \text{AdaptFun} \]
\[ t_1^0 <:_{k_1} t_1 \quad t_2 <:_{k_2} t_2^0 \]
\[ t_1 \leadsto t_2 <:_0 t_1^0 \leadsto t_2^0 \]
\begin{align*}
\text{Fix} & \quad \Gamma \vdash e : t \rightarrow t^0 \quad \Gamma \vdash t^0 <_{1} t \quad \Gamma \vdash t^0 \text{value} \\
\Gamma \vdash \text{fix} e : t^0 &
\end{align*}
Typing Local Time Scales

Scale
```
\Gamma \downarrow_{ct} \Gamma^0 \quad \Gamma^0 \quad e : t^0 \quad t^0 \uparrow_{ct} t
```
\[\Gamma \quad e : t\]
Soundness and Realizability

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

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

\[
S[\text{int} \to \text{int}] : \text{Stream}(\text{List}(\mathbb{N})) \to \text{Stream}(\text{List}(\mathbb{N}))
\]

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 $t$:

\[
W_t \subseteq S[t]
\]

\[
W_{dt :: ct} = \{xs \mid \text{clock } xs = [ct]\}
\]

\[
W_{t_1 \otimes t_2} = W_{t_1} \times W_{t_2}
\]

\[
W_{t \rightarrow t'} = \{f \mid \forall x \in W_t, (f \ x) \in W_{t'}\}
\]

\[
W_{\Gamma} \subseteq S[\Gamma]
\]

\[
\ldots
\]

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

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

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

Let us refine realizers as follows:

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

And restate the adequacy lemma:

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

An essential lemma for fixpoints:

\[
\forall t, t', \forall k, n \in \mathbb{N}, \forall xs \in W^n_t, (S[\ ` t <:_k t'] \ xs) \in W^{n+k}_{t'}
\]
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 $\mathbf{G}()$ (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
\[
\text{DownArrowBin}
\]
\[
\begin{array}{c}
\DownarrowBin \quad t_1' \uparrow_{ct} t_1 \\
\DownarrowBin \quad t_2 \downarrow_{ct} t_2' \\
\DownarrowBin \quad ct \leq (1)
\end{array}
\]
\[
\begin{array}{c}
\DownarrowBin \quad t_1 \rightarrow t_2 \\
\DownarrowBin \quad t_1' \rightarrow t_2'
\end{array}
\]

\[
\text{DownArrowPos}
\]
\[
\begin{array}{c}
\DownarrowPos \quad t_1' \downarrow_{ct'} t_1 \\
\DownarrowPos \quad t_2 \uparrow_{ct'} t_2' \\
\DownarrowPos \quad ct \text{ on } ct' = (1)
\end{array}
\]
\[
\begin{array}{c}
\DownarrowPos \quad t_1 \rightarrow t_2 \\
\DownarrowPos \quad t_1' \rightarrow t_2'
\end{array}
\]

\[
\text{DownOn}
\]
\[
\begin{array}{c}
\DownarrowOn \quad t \downarrow_{ct} t'' \\
\DownarrowOn \quad t'' \downarrow_{ct'} t'
\end{array}
\]
\[
\DownarrowOn \quad t \downarrow_{ct \text{ on } ct'} t'
\]