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 *ambiant* rather than *global*
- Make the type system more uniform

Linear Higher-Order Functions

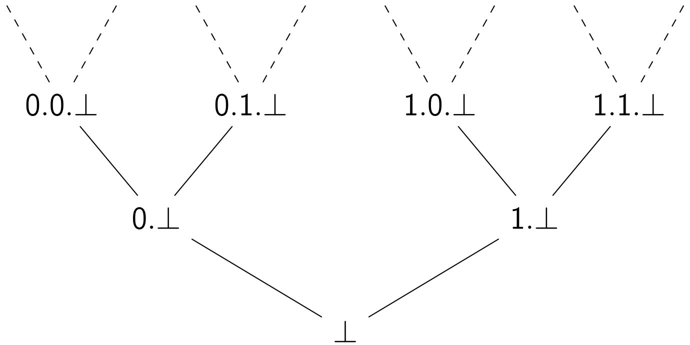
- Call every function you receive exactly once
- Enable *modular* compilation to hardware

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*:



Stream Functions (1/2)

Consider the following function

$$\begin{array}{lcl} f & : & \mathit{Stream}(\mathbb{N}) \to \mathit{Stream}(\mathbb{N}) \\ f(x.xs) & = & (x+1).(f\ xs) \end{array}$$

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

$$\begin{array}{lcl} m & : & \mathcal{M}(\mathbb{N}, \mathbb{N}) \\ m & = & (\{*\}, *, \lambda(*, x).(*, x+1)) \end{array}$$

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?

$$\begin{array}{lcl} g & : & \mathit{Stream}(\mathbb{N}) \to \mathit{Stream}(\mathbb{N}) \\ g(x.xs) & = & (x+1).(x-1).(g\ xs) \end{array}$$

Yes, if we cheat a bit.

$$\begin{array}{lcl} m_1 & : & \mathcal{M}(\mathbb{N}, \mathit{List}(\mathbb{N})) \\ m_1 & = & (\{*\}, *, \lambda(*, x).(*, [x+1; x-1])) \end{array}$$

Another possibility:

$$\begin{array}{lcl} m_2 & : & \mathcal{M}(\mathit{List}(\mathbb{N}), \mathbb{N}) \\ m_2 & = & (\mathbb{N} \cup \{*\}, *, \\ & & \ \lambda(s, x).\text{if } s = * \text{ then } (\mathit{hd}\ x, \mathit{hd}\ x + 1) \text{ else } (*, s-1)) \end{array}$$

Stream Functions and Clocks

Naively speaking, the function g is not length-preserving.

$$\begin{array}{lcl} g & : & \mathit{Stream}(\mathbb{N}) \to \mathit{Stream}(\mathbb{N}) \\ g\ (x.xs) & = & (x+1).(x-1).(g\ xs) \end{array}$$

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

$$\begin{array}{lcl} g_1 & : & \mathit{Stream}(\mathit{List}(\mathbb{N})) \to \mathit{Stream}(\mathit{List}(\mathbb{N})) \\ g_1\ ([x].xs) & = & [x+1; x-1].(g_1\ xs) \end{array}$$

$$\begin{array}{lcl} g_2 & : & \mathit{Stream}(\mathit{List}(\mathbb{N})) \to \mathit{Stream}(\mathit{List}(\mathbb{N})) \\ g_2\ ([x].xs) & = & [x+1].(\mathrm{let}\ [].xs' = xs\ \mathrm{in} \\ & & \qquad [x-1].(g_2\ xs')) \end{array}$$

Functions g_1 and g_2 are length-preserving.

Synchronizing Functions

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

$$\begin{array}{lcrcl} g & : & Stream(\mathbb{N}) & \to & Stream(\mathbb{N}) \\ g_1 & : & Stream(List(\mathbb{N})) & \to & Stream(List(\mathbb{N})) \\ g_2 & : & Stream(List(\mathbb{N})) & \to & Stream(List(\mathbb{N})) \end{array}$$

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

	Input list sizes	Output list sizes
g_1	$(1)^\omega$	$(2)^\omega$
g_2	$(1\ 0)^\omega$	$(1)^\omega$

These integer streams, *clocks*, fully characterize g_1 and g_2.
We write:

$$\begin{array}{lcrl} g_1 & :: & (1) & \multimap (2) \\ g_2 & :: & (1\ 0) & \multimap (1) \end{array}$$

From Streams to Clocked Streams, and back

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

$$\begin{tikzcd} \mathit{Stream}(V) \arrow[rr, bend left, "\mathit{pack}_w"] & & \mathit{Stream}(\mathit{List}(V)) \arrow[ll, bend left, "\mathit{unpack}"] \end{tikzcd}$$

For example:

$x = \mathit{pack}_{1\,(1\,0)^\omega}\ (a.b.c.d\ldots)$	$[a]$	$[b]$	$[]$	$[c]$	$[]$	$\ldots$
$y = \mathit{pack}_{(0\,2)^\omega}\ (a.b.c.d\ldots)$	$[]$	$[a;b]$	$[]$	$[c;d]$	$[]$	$\ldots$

Obviously:

$$\mathit{unpack}\ x = \mathit{unpack}\ y$$

Synchronous Stream Functions

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

$$
\begin{array}{lcl}
g_1 & :: & (1) \multimap (2) \\
g_1 & = & pack_{(2)} \circ g \circ unpack \\
g_2 & :: & (1\,0) \multimap (1) \\
g_2 & = & pack_{(1)} \circ g \circ unpack
\end{array}
$$

What about the following function?

$$
\begin{array}{lcl}
g_3 & ::_{?} & (0\,1) \multimap (1) \\
g_3 & = & pack_{(1)} \circ g \circ unpack
\end{array}
$$

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

$$
g_3\,([].\bot) \;=\; \bot
$$

From Synchronization to Desynchronization

$$
\begin{tikzcd}
g_1 : (1) \multimap (2) \arrow[d, bend right=60, "desync_{(1)\multimap(2)}"'] \\
g : Stream(\mathbb{N}) \to Stream(\mathbb{N}) \arrow[u, bend right=60, "sync_{(1)\multimap(2)}"'] \arrow[d, bend left=60, "sync_{(1\ 0)\multimap(1)}"] \\
g_2 :: (1\,0) \multimap (1) \arrow[u, bend left=60, "desync_{(1\ 0)\multimap(1)}"]
\end{tikzcd}
$$

Playing with Synchronous Functions: Buffers (1/2)

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

$x :: (1\,0)$	$[a]$	$[]$	$[b]$	$[]$	$[c]$	$[]$	$\ldots$
$x' :: (0\,1)$	$[]$	$[a]$	$[]$	$[b]$	$[]$	$[c]$	$\ldots$

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

- $(1\,0) <:_1 (0\,1)$

$$\begin{array}{ccccccccc}
(\mathbf{1\,0}) & \mathbf{1} & \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{1} & \mathbf{0} & \cdots \\
(\mathbf{0\,1}) & \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{1} & \cdots
\end{array}$$

- $(1\,0\,1) <:_0 (0\,1\,1)$ but not $(1\,0\,1) <:_1 (0\,1\,1)$

$$\begin{array}{ccccccccc}
(\mathbf{1\,0}\,) & \mathbf{1} & \mathbf{0} & \mathbf{1} & \mathbf{1} & \mathbf{0} & \mathbf{1} & \cdots \\
(\mathbf{0\,1\,1}) & \mathbf{0} & \mathbf{1} & \mathbf{1} & \mathbf{0} & \mathbf{1} & \mathbf{1} & \cdots
\end{array}$$

Playing with Synchronous Functions: Buffers (2/2)

Now, given a function $h :: w_1 \multimap w_2$, we may put a buffer on its...

- Output: if $w_2 <:_k w_2'$, we define

$$\begin{array}{rcl} h' & :: & w_1 \multimap w_2' \\ h' & = & \mathit{buffer}_{w_2 <:_k w_2'} \circ h \end{array}$$

For example:

$$(1) \multimap (1\,0) <: (1) \multimap (0\,1)$$

- Input: if $w_1' <:_k w_1$, we define

$$\begin{array}{rcl} h'' & :: & w_1' \multimap w_2 \\ h'' & = & h \circ \mathit{buffer}_{w_1' <:_k w_1} \end{array}$$

For example:

$$(0\,1) \multimap (1) <: (1\,0) \multimap (1)$$

Playing with Synchronous Functions: Feedback

Given a function $h :: w_1 \multimap w_2$, is it safe to compute $x = h\ x$?
What about...

$$
\begin{array}{lclcll}
h_1 & :: & (1) & \multimap & (1) & \text{KO} \\
h_2 & :: & (0\,1) & \multimap & (1\,0) & \text{OK} \\
h_3 & :: & (0\,1\,1) & \multimap & (1\,0\,1) & \text{KO}
\end{array}
$$

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:

$$\begin{array}{lll} \text{from} & \mathit{Stream}(\mathbb{N}) & \rightarrow \mathit{Stream}(\mathbb{N}) \\ \text{to} & \mathit{Stream}(\mathit{List}(\mathbb{N})) & \rightarrow \mathit{Stream}(\mathit{List}(\mathbb{N})) \end{array}$$

- 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.

Playing with Synchronous Functions: Local Time

Take any function f implemented by state machine m, with

$$f :: (1\,0) \multimap (0\,1)$$

We can transform f into f' such that

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

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

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

$$(1\,0) \multimap (0\,1) \uparrow_{(2)} (1) \multimap (1)$$

Local Time Scales and Scatter/Gather

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?

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

$$
\begin{array}{llllll}
(1\,0\,1) \multimap (0\,1\,1) & \uparrow_{(2\,1)} & (1) \multimap (1) & & \text{OK} \\
(0\,1\,1) \multimap (1\,0\,1) & \uparrow_{(2\,1)} & (1) \multimap (1) & & \text{OK}
\end{array}
$$

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

$$
\begin{array}{llllll}
(1) \multimap (1) & \downarrow_{(2\,1)} & (1\,0\,1) \multimap (0\,1\,1) & & \text{OK} \\
(1) \multimap (1) & \downarrow_{(2\,1)} & (0\,1\,1) \multimap (1\,0\,1) & & \text{KO}
\end{array}
$$

Scatter/Gather: Streams

Consider two simple examples:

$$(1\,0) \uparrow_{(2)} (1)$$

What is the action of (2) on (1 0) that gives (1)?

Let us define clock composition as

$$\begin{array}{rcl} \_ \ on \ \_ & : & Stream(\mathbb{N}) \times Stream(\mathbb{N}) \to Stream(\mathbb{N}) \\ (n.w) \ on \ (m_1 \ldots m_n.w') & = & (\sum_{1 \leq i \leq n} m_i).(w \ on \ w') \end{array}$$

We can now define:

$$w_1 \uparrow_w w_2 \Leftrightarrow w \ on \ w_1 = w_2$$

Similarly, $(1) \downarrow_{(2)} (0\,1)$ because $(1) = (2)$ *on* $(1\,0)$

Scatter/Gather: Functions

Going back to our first example: $(1\,0) \multimap (0\,1) \uparrow_{(2)} (1) \multimap (1)$. Why?

$$\begin{array}{rccc} \text{Because we have} & (1) & \downarrow_{(2)} & (1\,0) \\ \text{and} & (0\,1) & \uparrow_{(2)} & (1) \end{array}$$

This suggests the reasoning principle

$$\frac{w'_1 \downarrow_w w_1 \qquad w_2 \uparrow_w w'_2}{w_1 \multimap w_2 \uparrow_w w'_1 \multimap w'_2}$$

More complex principles can be found for $w_1 \multimap w_2 \downarrow_w w'_1 \multimap w'_2$

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:

```
x  =  0 fby y
y  =  x
```

Consider the signature below:

$$f \quad :: \quad (0\,1) \otimes 0(0\,1) \multimap (1\,0) \otimes (0\,1)$$

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

x	$f\ x$
$(\bot, \bot)$	$(0.\bot, \bot)$
$(0.\bot, \bot)$	$(0.\bot, 0.\bot)$
$(0.\bot, 0.\bot)$	$(0.0.\bot, 0.\bot)$
$(0.0.\bot, 0.\bot)$	$(0.0.\bot, 0.0.\bot)$
$\ldots$	$\ldots$

Putting it all together (2/2)

So, with $f :: (0\,1) \otimes 0(0\,1) \multimap (1\,0) \otimes (0\,1)$, since

$$\begin{array}{rcl} (1\,0) & <:_1 & (0\,1) \\ (0\,1) & <:_1 & 0(0\,1) \end{array}$$

we know that the fixpoint is total, and get

$$\mathit{lfp}\ f :: (1\,0) \otimes (0\,1)$$

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

Inside view	Outside view	
$(1\,0) <:_1 (0\,1)$	$(1) <:_0 (1)$	Wire
$(0\,1) <:_1 0(0\,1)$	$(1) <:_1 0(1)$	Memory

From Semantics to Syntax

$$
\begin{array}{rcl}
e & ::= & x \\
& | & \lambda x.e \\
& | & e\ e \\
& | & (e, e) \\
& | & \mathtt{let}\ (x, x) = e\ \mathtt{in}\ e \\
& | & \mathtt{fix}\ e \\
& | & c \\
& | & op\ e \\
& | & \mathtt{merge}\ p\ e\ e \\
& | & e\ \mathtt{when}\ p \\
p & ::= & c^*(c^+)
\end{array}
$$

$$
\begin{array}{rcl}
t & ::= & dt :: ct \\
& | & t \otimes t \\
& | & t \multimap t \\
dt & ::= & \mathbf{bool} \mid \mathbf{int} \mid \ldots \\
ct & ::= & p \\
& | & ct\ \mathbf{on}\ ct \\
\Gamma & ::= & \square \\
& | & \Gamma, x : t
\end{array}
$$

Typing Buffers

Sub

$$\frac{\Gamma \vdash e : t \qquad \vdash t <:_k t'}{\Gamma \vdash e : t'}$$

AdaptFun

$$\frac{t_1' <:_{k_1} t_1 \qquad t_2 <:_{k_2} t_2'}{t_1 \multimap t_2 <:_0 t_1' \multimap t_2'}$$

Typing Feedback

Fix

$$\frac{\Gamma \vdash e : t \multimap t' \qquad \vdash t' <:_1 t \qquad \vdash t' \text{ value}}{\Gamma \vdash \texttt{fix}\ e : t'}$$

Typing Local Time Scales

Scale

$$\frac{\vdash \Gamma \downarrow_{ct} \Gamma' \qquad \Gamma' \vdash e : t' \qquad \vdash t' \uparrow_{ct} t}{\Gamma \vdash e : t}$$

Soundness and Realizability

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

$$\begin{array}{lcl} \mathcal{K}[\![\vdash e : \mathbf{int} :: ct \multimap \mathbf{int} :: ct]\!] & : & \mathit{Stream}(\mathbb{N}) \to \mathit{Stream}(\mathbb{N}) \\ \mathcal{S}[\![\vdash e : \mathbf{int} :: ct \multimap \mathbf{int} :: ct]\!] & : & \mathit{Stream}(\mathit{List}(\mathbb{N})) \to \mathit{Stream}(\mathit{List}(\mathbb{N})) \end{array}$$

Soundness theorem

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

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

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

$$\forall e, t, \mathcal{S}[\![\vdash e : t]\!] = \mathit{sync}_t\ \mathcal{K}[\![\vdash e : t]\!]$$

Soundness proof (1/2)

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

$$
\begin{array}{lcl}
\mathcal{W}_t & \subseteq & \mathcal{S}[\![t]\!] \\
\mathcal{W}_{dt\ ::\ ct} & = & \{xs \mid clock\ xs = [\![ct]\!]\} \\
\mathcal{W}_{t_1 \otimes t_2} & = & \mathcal{W}_{t_1} \times \mathcal{W}_{t_2} \\
\mathcal{W}_{t \multimap t'} & = & \{f \mid \forall x \in \mathcal{W}_t, (f\ x) \in \mathcal{W}_{t'}\} \\
\mathcal{W}_\Gamma & \subseteq & \mathcal{S}[\![\Gamma]\!] \\
 & \ldots &
\end{array}
$$

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

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

- 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:

$$
\begin{array}{lcl}
\mathcal{W}_t^{n \in \mathbb{N}} & \subseteq & \mathcal{S}[\![t]\!] \\
\mathcal{W}_{dt\,::\,ct}^{n} & = & \{xs \mid clock\ xs =_n \mathcal{S}[\![ct]\!]\} \\
\mathcal{W}_{t_1 \otimes t_2}^{n} & = & \mathcal{W}_{t_1}^{n} \times \mathcal{W}_{t_2}^{n} \\
\mathcal{W}_{t \multimap t'}^{n} & = & \{f \mid \forall m \leq n, \forall x \in \mathcal{W}_t^m, (f\ x) \in \mathcal{W}_{t'}^{m}\} \\
\mathcal{W}_\Gamma^{n \in \mathbb{N}} & \subseteq & \mathcal{S}[\![\Gamma]\!] \\
 & \ldots &
\end{array}
$$

- And restate the adequacy lemma:

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

- An essential lemma for fixpoints:

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

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 **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

Scatter/Gather: The Whole Story

DownArrowBin
$$\frac{\vdash t_1' \uparrow_{ct} t_1 \qquad \vdash t_2 \downarrow_{ct} t_2' \qquad ct \leq (1)}{\vdash t_1 \multimap t_2 \downarrow_{ct} t_1' \multimap t_2'}$$

DownArrowPos
$$\frac{\vdash t_1' \downarrow_{ct'} t_1 \qquad \vdash t_2 \uparrow_{ct'} t_2' \qquad ct \textbf{ on } ct' = (1)}{\vdash t_1 \multimap t_2 \downarrow_{ct} t_1' \multimap t_2'}$$

DownOn
$$\frac{\vdash t \downarrow_{ct} t'' \qquad \vdash t'' \downarrow_{ct'} t'}{\vdash t \downarrow_{ct \textbf{ on } ct'} t'}$$