A Synchronous Functional Language with Integer Clocks

PhD Defense - Adrien Guatto

Project-team PARKAS
École normale supérieure & INRIA
PhD supervisors: Albert Cohen, Marc Pouzet, Louis Mandel
The Context
The Context

Critical Control Software

High-Performance Streaming
The Context

Critical Control Software  High-Performance Streaming

Reactive Systems
Reactive Systems
Reactive Systems
The system runs…
The system runs…
• ... in continuous interaction with its environment,
The system runs…
• … in continuous interaction with its environment,
• … for an unbounded amount of time,
The system runs…
• … in continuous interaction with its environment,
• … for an unbounded amount of time,
• … using limited resources,
The system runs…
• … in continuous interaction with its environment,
• … for an unbounded amount of time,
• … using limited resources,
• … and must not fail.
Synchronous Languages
Synchronous Languages

Esterel
Synchronous Languages

Esterel

Lustre

LUSTRE: a declarative language for programming synchronous systems

Abstract

LUSTRE is a synchronous programming language, whose code, semantics, and implementation are all viewed in the same natural way, as data-flow diagrams. Models are sequences of assignments, which are partial functions with a simple graphical representation. This assumption is completely adequate to formalize the semantics of LUSTRE, which is a state machine with synchronous control. Each assignment X=E defines a transformation of the system state, which is valid at a given instant. From these pure functions and their dependencies, the data-flow of the LUSTRE program is easily projected.

The language must satisfy the following properties:

- '0
- Each sequence of assignments defines a transformation of the system state, which is valid at a given instant.
- The sequence of assignments is finite.
- The sequence of assignments is used for programming purposes.

To describe the behavior of each assignment X=E, we must define the X variables at the left, and a set of constraints at the right. As these constraints are expressed in a language of their own, we first present a syntax and semantics for this environment, which is a simple program transformation. The dual language, which is used for programming purposes, is then presented.

This language is particularly well suited for the communication between independent agents, for the specification of data-processing algorithms, and for the generation of test-cases.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ACM Reference Format


http://doi.acm.org/10.1145/221566.221567

Asynchronous Languages
Synchronous Languages

Esterel

Lustre

Signal
Synchronous Languages

Domain-specific languages for reactive systems…
… based on discrete logical time,
… with limited theoretical expressiveness (sub-Turing),
… and tractable mathematical semantics,
… implemented via dedicated compilers.
Synchronous Languages

<table>
<thead>
<tr>
<th>Esterel</th>
<th>Lustre</th>
<th>Signal</th>
</tr>
</thead>
</table>

**Domain-specific languages** for reactive systems…
... based on *discrete logical time*,
... with limited theoretical expressiveness (sub-Turing),
... and tractable *mathematical semantics*,
... implemented via *dedicated compilers*.

The **compilers for synchronous languages**…
... apply mandatory static analyses to *reject programs*,
... generate finite-state code, software or hardware.
JPEG Zig-Zag walk
JPEG Zig-Zag walk

Zig-Zag Scan
JPEG Zig-Zag walk
JPEG Zig-Zag walk

Zig-Zag Scan
JPEG Zig-Zag walk

Zig-Zag Scan
let node zigzag x = o where
    rec (x0, x1, x2, x3, x4, x5, x6, x7, x8)
          = split x with (A B D G E C F H I)
    and o = merge (A B C D E F G H I) with
          | 0 -> buffer x0
          | 1 -> buffer x1
          ...
          | 8 -> buffer x8
    end

JPEG Zig-Zag Scan, source view
JPEG Zig-Zag Scan, as soon as possible

\[(A \ B \ D \ G \ E \ C \ F \ H \ I) \rightarrow (A \ B \ C \ D \ E \ F \ G \ H \ I)\]
JPEG Zig-Zag Scan, as soon as possible

(A B D G E C F H I) (A B C D E F G H I)

split

merge

7
JPEG Zig-Zag Scan, bursty

\[(A \ B \ D \ G \ E \ C \ F \ H \ I)\]

split

\[A \ B \ C \ D \ E \ F \ G \ H \ I\]

merge

\[8\]
JPEG Zig-Zag Scan, bursty

(A B D G E C F H I) (A B C D E F G H I)

split

merge

(A B D G E C F H I) (A B C D E F G H I)
Contributions
Contributions

My thesis is about capturing new scheduling trade-offs at the language-level and in a manner that respects the local structure of programs. The main ingredients are:
Contributions

My thesis is about capturing new scheduling trade-offs at the language-level and in a manner that respects the local structure of programs. The main ingredients are:

**Typed functional programming** as a viewpoint:
- express scheduling constraints as *types*,
- compile modularly typed programs to abstract circuits,
- handle functions of any order.
Contributions

My thesis is about capturing new scheduling trade-offs at the language-level and in a manner that respects the local structure of programs. The main ingredients are:

Typed functional programming as a viewpoint:
• express scheduling constraints as types,
• compile modularly typed programs to abstract circuits,
• handle functions of any order.

Integer clocks…
• describe transfers of atomic bursts of data between subprograms,
• enjoy new properties compared to binary clocks,
• extend n-synchrony [Cohen, Duranton, Mandel, Plateau, Pouzet…].
Contributions

My thesis is about capturing new scheduling trade-offs at the language-level and in a manner that respects the local structure of programs. The main ingredients are:

Typed functional programming as a viewpoint:
- express scheduling constraints as types,
- compile modularly typed programs to abstract circuits,
- handle functions of any order.

Integer clocks…
- describe transfers of atomic bursts of data between subprograms,
- enjoy new properties compared to binary clocks,
- extend n-synchrony [Cohen, Duranton, Mandel, Plateau, Pouzet…].

Local time scales…
- aggregate local steps to hide them from the surrounding context,
- allows new forms of modularity,
- enables a large simplification of the language.
Outline

Introduction

From functions to machines

Typing and compilation

Perspectives and conclusion
The Identity Function

let node id x = x
The Identity Function

```ml
let node id x = x
```

\[ id : (1) \rightarrow (1) \]
The Identity Function

\[
\text{let node id } x = x
\]

\[
id : (1) \rightarrow (1)
\]
The Identity Function

let node id x = x

id : (1) -> (1)
id : (2) -> (2)
The Identity Function

\[
\text{let node } \text{id } x = x
\]

\[
\text{id : (1) -> (1)}
\]

\[
\text{id : (2) -> (2)}
\]
The Identity Function

let node id x = x

id : (1) -> (1)
id : (2) -> (2)
The "Zero" Function

\[ \text{let node zero x} = \text{merge } A(B) \ 0 \ x \]
The "Zero" Function

let node zero x = merge A(B) 0 x

zero : 0(1) -> (1)
The "Zero" Function

\[
\text{let node zero } x = \text{merge } A(B) \ 0 \ x
\]

zero : \(\emptyset(1) \rightarrow (1)\)
The "Zero" Function

\[
\text{let node zero } x = \text{merge } A(B) \ 0 \ x
\]

zero : 0(1) -> (1)
The "Zero" Function

let node zero x = merge A(B) 0 x

zero : 0(1) -> (1)

zero : 1(2) -> (2)
The "Zero" Function

let node zero x = merge A(B) 0 x

zero : 0(1) -> (1)
zero : 1(2) -> (2)
The "Zero" Function

let node zero x = merge A(B) 0 x

zero : 0(1) -> (1)
zero : 1(2) -> (2)
The "Half" Function

\[
\text{let node even } x = x \text{ when } (1 \ 0)
\]
The "Half" Function

\[ \text{let node even } x = x \text{ when } (1 \ 0) \]

\[ \text{even : } (1) \rightarrow (1 \ 0) \]
The "Half" Function

let node even \( x = x \) when \( (1 \ 0) \)

\[
\text{even} : (1) \rightarrow (1 \ 0)
\]
The "Half" Function

\[\text{let node even } x = x \text{ when (1 0)}\]

\[\text{even : (1) -> (1 0)}\]
The "Half" Function

let node even x = x when (1 0)

even : (1) -> (1 0)

even : (2) -> (1)

even : (1) -> (1 0)

even : (2) -> (1)
The "Half" Function

```
let node even x = x when (1 0)
```

**even : (1) -> (1 0)**

**even : (2) -> (1)**
The "Halfsum" Function

\[
\text{let node halfsum } x = \hspace{1cm} \text{(x when (1 \ 0)) + (x when (0 \ 1))}
\]
The "Halfsum" Function

let node halfsum x =
  (x when (1 0)) + (x when (0 1))

halfsum : (1) -> (0 1)

graphic representation
let node halfsum x =
  (x when (1 0)) + (x when (0 1))

The "Halfsum" Function

halfsum : (1) -> (0 1)
The "Halfsum" Function

let node halfsum x =
  (x when (1 0)) + (x when (0 1))

halfsum : (1) -> (0 1)
The "Halfsum" Function

```
let node halfsum x =
  (x when (1 0)) + (x when (0 1))
```

halfsum : (1) -> (0 1)
The "Halfsum" Function

let node halfsum x =
  (x when (1 0)) + (x when (0 1))

halfsum : (1) -> (0 1)

14
The "Halfsum" Function

\[
\text{let node halfsum } x = \\
(x \text{ when } (1 \ 0)) + (x \text{ when } (0 \ 1))
\]

halfsum : (1) -> (0 1)
The "Halfsum" Function

let node halfsum x =
  (x when (1 0)) + (x when (0 1))

halfsum : (1) -> (0 1)
halfsum : (2) -> (1)

halfsum : (1) -> (0 1)
halfsum : (2) -> (1)
The "Halfsum" Function

\[
\text{let node halfsum } x = (x \text{ when (1 0)}) + (x \text{ when (0 1)})
\]

halfsum : (1) -> (0 1)

halfsum : (2) -> (1)
"Halfsum" and Local Time Scales

\[
\text{let node halfsum2 } x = \text{ halfsum } x
\]

halfsum : (1) -> (0 1)

halfsum2 : (2) -> (1)
"Halfsum" and Local Time Scales

let node halfsum2 x = halfsum x

halfsum : (1) -> (0 1)

halfsum2 : (2) -> (1)

halfsum2 : (2) -> (1)
"Halfsum" and Local Time Scales

\[
\text{let node halfsum2 } \ x = \text{ halfsum } \ x
\]

\[
\text{halfsum : (1) } \rightarrow \text{ (0 1)}
\]

External Step 1

\[
\text{halfsum2 : (2) } \rightarrow \text{ (1)}
\]
"Halfsum" and Local Time Scales

let node halfsum2 x = halfsum x

halfsum : (1) -> (0 1)

halfsum2 : (2) -> (1)

External Step 1

halfsum2 : (2) -> (1)
"Halfsum" and Local Time Scales

\[
\text{let node } \text{halfsum2 } x = \text{halfsum } x
\]

\[
\text{halfsum : (1) } \rightarrow (0 \ 1)
\]

\[
\text{halfsum2 : (2) } \rightarrow (1)
\]
"Halfsum" and Local Time Scales

\[
\text{let node halfsum2 x = halfsum x}
\]

halfsum : (1) -> (0 1)

External Step 1

Internal Step 2

halfsum2 : (2) -> (1)
"Halfsum" and Local Time Scales

let node halfsum2 x = halfsum x

halfsum : (1) -> (0 1)

halfsum2 : (2) -> (1)

halfsum2 : (2) -> (1)

15
"Halfsum" and Local Time Scales

let node halfsum2 x = halfsum x

halfsum : (1) -> (0 1)

halfsum

up (2)

halfsum2 : (2) -> (1)
"Halfsum" and Local Time Scales

\[
\text{let node } \text{halfsum2 } x = \text{halfsum } x
\]

halfsum : (1) \to (0 1)

\[
\begin{array}{c}
\downarrow (2) \quad \text{halfsum} \quad \uparrow (2)
\end{array}
\]

halfsum2 : (2) \to (1)
The "Acc" Function

\[
\text{let node acc } x = y \text{ where } \\
\quad \text{rec } y = (\text{zero } y) + x
\]
The "Acc" Function

let node acc x = y where
  rec y = (zero y) + x

acc : (1) -> (1)
The "Acc" Function

let node acc x = y where
  rec y = (zero y) + x

acc : (1) -> (1)
The "Acc" Function

let node acc x = y where rec y = (zero y) + x

acc : (1) -> (1)
let node acc x = y where
  rec y = (zero y) + x

acc : (1) -> (1)
The "Acc" Function

let node acc x = y where rec y = (zero y) + x

acc : (1) -> (1)

acc : (1) -> (1)
The "Acc" Function

let node acc x = y where
  rec y = (zero y) + x

acc : (1) -> (1)

16
Adaptability and Delays
Adaptability and Delays

(1) \( \leq \) \( O(1) \)
Adaptability and Delays

\[ \langle :, 0(1) \rangle \]
Adaptability and Delays
Adaptability and Delays
Adaptability and Delays

(1) \textless; 1 0(1)

(1) \textless; 1 0(1)
Adaptability and Delays

\[ (1) \xrightarrow{\text{<:}} 0(1) \]

\[ (1) \xrightarrow{\text{<:}} 0(1) \]

\[ (2) \xrightarrow{\text{<:}} 1(2) \]
Adaptability and Delays

(1) \(<: 1 \ 0(1)

(2) \(<: 1 \ 1(2)

17
Adaptability and Delays

(1) <:<₁ 0(1)

(2) <::₁ 0(1)

(2) 1(2)
Adaptability and Delays

(1) $<: 0(1)$

(1) $<: 1 0(1)$

(2) $1(2)$
Adaptability and Delays

(1) $\leq_1 0(1)$

(2) $\leq_0 1(2)$

(2) $\leq_1 1(2)$
The "Acc" Function, Two-by-Two?

let node acc x = y where
  rec y = (zero y) + x

zero : 1(2) -> (2)

acc : (2) -> (2) ?
The "Acc" Function, Two-by-Two?

\[
\text{let node acc } x = y \text{ where rec } y = (\text{zero } y) + x
\]

\[
\text{zero : } 1(2) \rightarrow (2)
\]

\[
\text{acc : } (2) \rightarrow (2)
\]
The "Acc" Function, Two-by-Two?

\[
\text{let node acc x = y where}\ \\
\text{rec y = (zero y) + x}\ \\
\text{zero : 1(2) -> (2)}
\]

\[
\text{acc : (2) -> (2) ?}
\]
The "Acc" Function, Two-by-Two?

let node acc x = y where
  rec y = (zero y) + x

zero : 1(2) -> (2)

acc : (2) -> (2) ?
The "Acc" Function, Two-by-Two?

\[
\begin{align*}
\text{let node acc x = y where} \\
\text{rec y = (zero y) + x}
\end{align*}
\]

zero : 1(2) -> (2)

\[
\begin{array}{c}
\text{acc : (2) -> (2)} \\
\end{array}
\]
let node acc x = y where
rec y = (zero y) + x

zero : 1(2) -> (2)

acc : (2) -> (2) ?
The "Acc" Function, Two-by-Two?

let node acc x = y where
  rec y = (zero y) + x

zero : 1(2) -> (2)

acc : (2) -> (2) ?
The "Acc" Function, Two-by-Two?

let node acc x = y where
  rec y = (zero y) + x

zero : 1(2) -> (2)
acc : (2) -> (2) ?
"Acc" Two-by-Two and Local Time Scales

let node acc2 x = acc x

acc : (1) -> (1)

acc2 : (2) -> (2)
"Acc" Two-by-Two and Local Time Scales

```latex
let node acc2 x = acc x

acc : (1) -> (1)
acc2 : (2) -> (2)
```

External Step 1

acc2 : (2) -> (2)
"Acc" Two-by-Two and Local Time Scales

\[
\text{let node acc2 \( x = acc ~ x \)}
\]

\[
\text{acc : (1) -> (1)}
\]

\[
\text{acc2 : (2) -> (2)}
\]

External Step 1

\[
\text{acc2 : (2) -> (2)}
\]
"Acc" Two-by-Two and Local Time Scales

let node acc2 x = acc x

acc : (1) -> (1)
acc2 : (2) -> (2)

Internal Step 1
External Step 1
"Acc" Two-by-Two and Local Time Scales

let node acc2 x = acc x

acc : (1) -> (1)

acc2 : (2) -> (2)

External Step 1

Internal Step 2

19
"Acc" Two-by-Two and Local Time Scales

let node acc2 x = acc x

acc : (1) -> (1)

acc2 : (2) -> (2)
Outline

Introduction

From functions to machines

Typing and compilation

Perspectives and conclusion
From Typing Derivations to Abstract Circuits

e
From Typing Derivations to Abstract Circuits

\[
\Gamma \vdash e : t
\]
From Typing Derivations to Abstract Circuits

\[
\Gamma \vdash e : t
\]
From Typing Derivations to Abstract Circuits

\[ \Gamma \vdash e : t \]

First-order
From Typing Derivations to Abstract Circuits

\[ \Gamma \vdash e : t \]

First-order

Higher-order
From Typing Derivations to Abstract Circuits

\[
\begin{array}{c}
\Gamma \vdash e : t \\
\end{array}
\]

First-order

Linear Higher-order
From Typing Derivations to Abstract Circuits

\[ \Gamma \vdash e : t \]

First-order

Linear Higher-order

Not real circuits!
Compiling the "Pair" Rule

\[
\text{PAIR} \\
\Gamma \vdash \Gamma_1 \otimes \Gamma_2 \quad \Gamma_1 \vdash e_1 : t_1 \quad \Gamma_2 \vdash e_2 : t_2 \\
\hline \\
\Gamma \vdash (e_1, e_2) : t_1 \otimes t_2
\]
Compiling the "Pair" Rule

\[
\begin{align*}
\text{PAIR} & \quad \Gamma \vdash \Gamma_1 \otimes \Gamma_2 \\
& \quad \Gamma_1 \vdash e_1 : t_1 \\
& \quad \Gamma_2 \vdash e_2 : t_2 \\
\hline
& \quad \Gamma \vdash (e_1, e_2) : t_1 \otimes t_2
\end{align*}
\]
Compiling the "Pair" Rule

\[ \text{PAIR} \]

\[
\Gamma \vdash \Gamma_1 \otimes \Gamma_2 \\
\Gamma_1 \vdash e_1 : t_1 \\
\Gamma_2 \vdash e_2 : t_2
\]

\[
\Gamma \vdash (e_1, e_2) : t_1 \otimes t_2
\]
Compiling the "Sub" Rule

\[
\begin{array}{c}
\text{SUB} \\
\Gamma \vdash e : ct \quad \vdash ct <: k ct' \\
\hline
\Gamma \vdash e : ct'
\end{array}
\]
Compiling the "Sub" Rule

Sub
\[ \Gamma \vdash e : ct \quad \vdash ct <: k ct' \]
\[ \Gamma \vdash e : ct' \]

\[
\begin{array}{c}
\text{[d]} \\
\Gamma \\
\end{array} \xrightarrow{\text{[d]}} \quad \xrightarrow{<:} \quad \begin{array}{c}
ct \\
ct' \\
\end{array}
\]
Compiling the "Sub" Rule

\[
\begin{array}{c}
\text{SUB} \\
\Gamma \vdash e : ct \quad \vdash ct <: k ct' \\
\hline \\
\Gamma \vdash e : ct'
\end{array}
\]
Compiling the "Sub" Rule

\[
\begin{array}{c}
\text{SUB} \\
\Gamma \vdash e : ct \quad \vdash ct <: k ct' \\
\hline
\Gamma \vdash e : ct'
\end{array}
\]

\[
\begin{array}{c}
\Gamma \\
[d] \\
ct \\
<: \\
ct' \\
\hline
\Gamma ct ct' \\
[ct']
\end{array}
\]

Current State

Next State
Compiling the "Pair" Rule, with State

\[
\begin{align*}
\text{PAIR} & \quad \Gamma \vdash \Gamma_1 \otimes \Gamma_2 \\
\Gamma_1 & \vdash e_1 : t_1 \\
\Gamma_2 & \vdash e_2 : t_2 \\
\hline
\Gamma & \vdash (e_1, e_2) : t_1 \otimes t_2
\end{align*}
\]
Compiling the "Pair" Rule, with State

\[ \begin{align*}
\text{PAIR} & : \Gamma \vdash \Gamma_1 \otimes \Gamma_2 \\
\Gamma & \vdash e_1 : t_1 \\
\Gamma & \vdash e_2 : t_2 \\
\Gamma & \vdash (e_1, e_2) : t_1 \otimes t_2
\end{align*} \]
Compiling the "Scale" Rule

\[
\frac{
    \text{SCALE} \\
    \vdash \Gamma \downarrow_{ct} \Gamma' \\
    \Gamma' \vdash e : t' \\
    \vdash t' \uparrow_{ct} t
}{
    \vdash \Gamma \vdash e : t
} 
\]
Compiling the "Scale" Rule

\[
\begin{array}{c}
\Gamma \downarrow_{(2)} \Gamma' \\
\Gamma' \vdash e : t' \\
\Gamma' \vdash t' \uparrow_{(2)} t \\
\hline
\Gamma \vdash e : t
\end{array}
\]
Compiling the "Scale" Rule

\[
\Gamma \vdash \Gamma' \quad \Gamma' \vdash e : t' \quad \Gamma' \vdash t' \uparrow (2) t
\]

\[
\Gamma \vdash e : t \quad \Gamma' \vdash t' \uparrow (2) t
\]
Compiling the "Scale" Rule

\[
\begin{align*}
\text{d1} & \quad \text{d2} & \quad \text{d3} \\
\vdash \Gamma \downarrow (2) \Gamma' & \quad \Gamma' \vdash e : t' & \quad \vdash t' \uparrow (2) t \\
\hline
\Gamma \vdash e : t
\end{align*}
\]
Compiling the "Scale" Rule

\[
\begin{align*}
\Gamma & \vdash e : t' \\
\Gamma' & \vdash e : t
\end{align*}
\]

\[
\begin{array}{c}
\Gamma \vdash e : t \\
\Gamma' \vdash e : t'
\end{array}
\]

\[
\begin{align*}
\Gamma & \vdash e : t' \\
\Gamma' & \vdash e : t
\end{align*}
\]

\[
\begin{array}{c}
\Gamma \vdash e : t \\
\Gamma' \vdash e : t'
\end{array}
\]

\[
\begin{align*}
\Gamma & \vdash e : t' \\
\Gamma' & \vdash e : t
\end{align*}
\]

\[
\begin{array}{c}
\Gamma \vdash e : t \\
\Gamma' \vdash e : t'
\end{array}
\]
Outline

Introduction

From functions to machines

Typing and compilation

Perspectives and conclusion
My Thesis

Formal development of the language:
• a semantics for untyped (« raw ») programs,
• a semantics to well-typed (« annotated ») programs,
• algebraic rules for the ↑ and ↓ judgments.

Soundness proofs:
• typed programs do not deadlock,
• annotations « refine » raw programs (pseudo-coherence).

Language extensions:
• reusable programs, beyond linear types (k-linear, nodes),
• bounded clock polymorphism to increase code reuse,
• data-dependent clocks to go beyond ultimately periodic clocks.
let node zigzag x = o where
    rec (x0, x1, x2, x3, x4, x5, x6, x7, x8)
          = split x with (A B D G E C F H I)
    and o = merge (A B C D E F G H I) with
    | 0 -> buffer x0
    | 1 -> buffer x1
    ... 
    | 8 -> buffer x8 
end

Prototype Implementation
Prototype Implementation

```
let node zigzag x = o
  where
    rec (x0, x1, x2, x3, x4, x5, x6, x7, x8)
          = split x with (A B D G E C F H I)
 ... x8
end

$ asc -i zigzag.as
val zigzag : (float :: (1)) -> (float :: 11000(1))
```
let node zigzag x = o where
    rec (x0, x1, x2, x3, x4, x5, x6, x7, x8)
          = split x with (A B D G E C F H I)
 ... (float :: (1)) -> (float :: 11000(1))
 $ asc -i zigzag.as -max_burst 9
 val zigzag : 
(float :: (9)) -> (float :: (9))
An Alternative: Arrays and Iterators

- Several versions and dialects of Lustre [Lustre v4+, Heptagon] have functional arrays and iterators.
- Some programs can be written with either arrays or relying on integer clocks (e.g., ZigZag Scan).
An Alternative: Arrays and Iterators

- Several versions and dialects of Lustre [Lustre v4+, Heptagon] have functional arrays and iterators.
- Some programs can be written with either arrays or relying on integer clocks (e.g., ZigZag Scan).

- Integer clocks benefit for compiler support:
  - correct by construction,
  - support for automated conversions,
  - enable design-space exploration.
- Integer clocks feel closer to the philosophy of SPLs.
- Arrays give more control to the programmer:
  - random access and arbitrary traversals;
- We know how to optimize arrays [Morel, Pasteur].
An Alternative: Arrays and Iterators

Several versions and dialects of Lustre [Lustre v4+, Heptagon] have functional arrays and iterators.

Some programs can be written with either arrays or relying on integer clocks (e.g., ZigZag Scan).

- Integer clocks benefit for compiler support:
  - correct by construction,
  - support for automated conversions,
  - enable design-space exploration.
- Integer clocks feel closer to the philosophy of SPLs.
- Arrays give more control to the programmer:
  - random access and arbitrary traversals;
- We know how to optimize arrays [Morel, Pasteur].
An Alternative: Causality Analysis

- Lustre, Lucid Synchrone and Lucy-n use a *causality analysis* after clock typing. It forbids instantaneous cycles in the bodies of a set of recursive definitions.
- We use a more brutal criterion: there must be a delay on each feedback arc. But…
  - Delays do not have to be initialized (n-synchrony),
  - Lost steps recoverable by local time scales.
An Alternative: Causality Analysis

• Lustre, Lucid Synchrone and Lucy-n use a *causality analysis* after clock typing. It forbids instantaneous cycles in the bodies of a set of recursive definitions.
• We use a more brutal criterion: there must be a delay on each feedback arc. But...
  • Delays do not have to be initialized (n-synchrony),
  • Lost steps recoverable by local time scales.

• One formal system to rule them all.
• The intra-step scheduling pass disappears.
• In n-synchronous programs causality and clocks are not orthogonal anyway.
• Requires a complex (n-synchronous) clock language
• Puts greater burden on the type inference engine;
• Code generation aspects still unclear.
Clock Equivalence and Principality

- There is a preorder of "free time transforms".
- Its equivalence classes capture "causally equivalent" types.
- The existence of smallest ("principal") types generalizes the classic problem of modular scheduling for Lustre.
Related Work

Cyclic Scheduling (*DF, Marked graphs…)
- Powerful scheduling and optimization techniques,
- Beyond ultimately periodic words (scenario-aware, parametric).

Programming Languages for Real Circuits
- Languages with explicit time:
  - VHDL/Verilog: semantics unclear, low-level;
  - Lava [Sheeran et al.]: high-level, not modular, layout constraints;
  - Esterel [Berry et al.]: expressive, tackles difficult low-level issues (boolean optimization, multiclock circuits, ECO…).
- Timeless languages:
  - High-Level Synthesis: mostly seen as an optimization, scheduling, and binding problem;
  - Geometry of Synthesis [Ghica et al.]: concurrent, imperative, higher-order, recent work on schedules in types (ESOP’14).
A functional, typed point-of-view on stream processing:
  • traditional questions are relevant and interesting.

Typing as refinement, rather than only as bondage:
  • precise types for precise operational properties.

Modular synchrony, with local time scales:
  • no more distinction between intra- and inter-step time,
  • needed to deal with causality, as remarked by Pasteur.
BONUS SLIDES
The Cost of Concurrent Communication

Synchronous Data-Flows and Clocks

(a) - An example SDF graph

\[
\begin{align*}
& w_a \text{ on } (2)^\omega <_1 \text{ on } 6 \text{ (1)}^\omega \\
& w_b \text{ on } (3)^\omega <_1 \text{ on } 9 \text{ (1)}^\omega \\
& w_c \text{ on } (4)^\omega <_1 \text{ on } 8 \text{ (1)}^\omega \\
\end{align*}
\]

(b) - Inequations obeyed by its schedules

\[
\begin{align*}
& w_a = (1)^\omega, \ w_b = 2^2 (1^2 0)^\omega, \ w_c = 3^3 (1^2 0^2)^\omega \\
\end{align*}
\]

(c) - A valid solution; describes a schedule
# The Type System

\[ \Gamma \vdash e : t \]

**VAR**
\[
\Gamma, x : t \vdash x : t
\]

**Weak**
\[
\Gamma \vdash e : t \quad \Gamma \vdash e' : t' \\
\Gamma, x : t \vdash x : t
\]

**λ**
\[
\Gamma, x : e, t \vdash e : t' \\
\Gamma \vdash \text{fun}(x, e) : t \rightarrow t'
\]

**App**
\[
\Gamma \vdash \Gamma_1 \otimes \Gamma_2 \quad \Gamma_1 \vdash e : t \rightarrow t' \quad \Gamma_2 \vdash e' : t \\
\Gamma \vdash e e' : t'
\]

**Pair**
\[
\Gamma \vdash \Gamma_1 \otimes \Gamma_2 \quad \Gamma_1 \vdash e_1 : t_1 \quad \Gamma_2 \vdash e_2 : t_2 \\
\Gamma \vdash (e_1, e_2) : t_1 \otimes t_2
\]

**Fix**
\[
\Gamma \vdash e : t \rightarrow t' \quad \Gamma \vdash e' : t \\
\Gamma \vdash \text{fix}(e) : t'
\]

**Const**
\[
\Gamma \vdash \text{of} : 
\Gamma \vdash \text{op} : (\text{int} \otimes \text{ct}) \otimes (\text{int} \otimes \text{ct}) \rightarrow (\text{int} \otimes \text{ct})
\]

**Merge**
\[
p \leq (1) \\
\Gamma \vdash \text{merge}(p : (dt \otimes \text{ct on } p) \otimes (dt \otimes \text{ct on } p)) \rightarrow (dt \otimes \text{ct})
\]

**When**
\[
p \leq (1) \\
\Gamma \vdash \text{when}(p : (dt \otimes \text{ct}) \rightarrow (dt \otimes \text{ct on } p))
\]

**Sub**
\[
\Gamma \vdash e : t \quad t \leq \lambda t'
\[
\Gamma \vdash e : t'
\]

**Scale**
\[
\Gamma \vdash \Gamma' \quad \Gamma' \vdash e : t' \quad e \uparrow \lambda t \\
\Gamma \vdash e : t
\]
The Type System

Most rules are the usual ones from the linear λ-calculus.
The Type System

- Most rules are the usual ones from the linear λ-calculus.
- Most rules are the usual ones from the linear λ-calculus.
- The Sub rule signals the insertion of a bounded buffer at its application point.
The Type System

- Most rules are the usual ones from the linear $\lambda$-calculus.
- The $\mathbf{Sub}$ rule signals the insertion of a bounded buffer at its application point.
## The Type System

<table>
<thead>
<tr>
<th>Rule</th>
<th>premises</th>
<th>conclusion</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>VAR</strong></td>
<td>$\Gamma \vdash \sigma$</td>
<td>$\Gamma, x : t \vdash x : t$</td>
</tr>
<tr>
<td><strong>Weaken</strong></td>
<td>$\Gamma \vdash e : t$  \quad $\Gamma \vdash e' : t'$ \quad $x \notin \text{FV}(e)$</td>
<td>$\Gamma, x : t \vdash e : t$</td>
</tr>
<tr>
<td><strong>Lambda</strong></td>
<td>$\Gamma, x : t \vdash e' : t'$</td>
<td>$\Gamma \vdash \text{fun}(x,e) : t \rightarrow t'$</td>
</tr>
<tr>
<td><strong>App</strong></td>
<td>$\Gamma \vdash \Gamma_1 \otimes \Gamma_2$</td>
<td>$\Gamma \vdash e : t$</td>
</tr>
<tr>
<td><strong>Pair</strong></td>
<td>$\Gamma \vdash \Gamma_1 \otimes \Gamma_2$</td>
<td>$\Gamma \vdash (e_1, e_2) : t_1 \otimes t_2$</td>
</tr>
<tr>
<td><strong>Let</strong></td>
<td>$\Gamma \vdash \Gamma_1 \otimes \Gamma_2$</td>
<td>$\Gamma \vdash \text{let}(x,y) = e \text{ in } e' : t$</td>
</tr>
<tr>
<td><strong>Fix</strong></td>
<td>$\Gamma \vdash e : t \rightarrow t'$  \quad $t' \prec_1 t$ \quad $t' \text{ value}$</td>
<td>$\Gamma \vdash \text{fix}(e) : t'$</td>
</tr>
<tr>
<td><strong>Const</strong></td>
<td>$\sigma : \text{dToF}(s) \otimes ct$</td>
<td>$\sigma : \text{op} : \text{(int} \otimes \text{ct}) \otimes \text{(int} \otimes \text{ct})$</td>
</tr>
<tr>
<td><strong>OP</strong></td>
<td>$\sigma : \text{merge}(p) : \text{(dt} \otimes \text{ct on \overline{p})} \otimes \text{(dt} \otimes \text{ct)}$</td>
<td>$\sigma : \text{merge}(p) : \text{(dt} \otimes \text{ct on \overline{p})} \otimes \text{(dt} \otimes \text{ct)}$</td>
</tr>
<tr>
<td><strong>Merge</strong></td>
<td>$p \leq (1)$</td>
<td>$\sigma : \text{merge}(p) : \text{(dt} \otimes \text{ct on \overline{p})} \otimes \text{(dt} \otimes \text{ct)}$</td>
</tr>
<tr>
<td><strong>When</strong></td>
<td>$p \leq (1)$</td>
<td>$\sigma : \text{when}(p) : \text{(dt} \otimes \text{ct on \overline{p})} \rightarrow \text{(dt} \otimes \text{ct)}$</td>
</tr>
</tbody>
</table>

- Most rules are the usual ones from the linear $\lambda$-calculus.
- The **Sub** rule signals the insertion of a bounded buffer at its application point.
- The **Fix** rule ensures the productivity of recursive definitions.
The Type System

<table>
<thead>
<tr>
<th>Rule</th>
<th>Conclusion</th>
</tr>
</thead>
<tbody>
<tr>
<td>VAR</td>
<td>$\Gamma \vdash \Gamma$ value $\vdash \Gamma, x : t \vdash x : t$</td>
</tr>
<tr>
<td>WEAKEN</td>
<td>$\Gamma \vdash e : t \vdash t'$ value $\vdash \Gamma, x : t' \vdash e : t$</td>
</tr>
<tr>
<td>LAMBDA</td>
<td>$\Gamma, x : t \vdash e : t'$ $\vdash \Gamma \vdash \text{fun} , x. e : t \vdash t'$</td>
</tr>
<tr>
<td>APP</td>
<td>$\Gamma \vdash \Gamma_1 \otimes \Gamma_2$ $\Gamma_1 \vdash e : t \vdash t'$ $\Gamma_2 \vdash e' : t$ $\vdash \Gamma \vdash \Gamma_1 \otimes \Gamma_2$ $\Gamma_1 \vdash e_1 : t_1$ $\Gamma_2 \vdash e_2 : t_2$ $\Gamma \vdash (e_1, e_2) : t_1 \otimes t_2$</td>
</tr>
<tr>
<td>LETPAIR</td>
<td>$\Gamma \vdash \Gamma_1 \otimes \Gamma_2$ $\Gamma_1 \vdash e : t_1 \otimes t_2$ $\Gamma_2, x : t_1, y : t_2 \vdash e' : t$ $\vdash \Gamma \vdash \Gamma_1 \otimes \Gamma_2$ $\Gamma_1 \vdash e : t_1 \vdash t' \vdash t' &lt;1 t \vdash t'$ value $\Gamma \vdash \text{fix} e : t'$</td>
</tr>
<tr>
<td>FIX</td>
<td>$\vdash \Gamma \vdash e : t \vdash t'$ $\vdash \Gamma' \vdash e : t' \vdash t'$ value $\vdash \Gamma \vdash e : t$</td>
</tr>
<tr>
<td>Merged</td>
<td>$p \leq (1)$ $\vdash \text{merge} , p : (dt : ct , on , p) \otimes (dt : ct , on , p') \rightarrow (dt : ct)$</td>
</tr>
<tr>
<td>WHEN</td>
<td>$p \leq (1)$ $\vdash \text{when} , p : (dt : ct) \rightarrow (dt : ct , on , p)$ $\vdash \Gamma \vdash e : t \vdash t' &lt;1 t' \vdash t'$ value $\vdash \Gamma \vdash e : t$</td>
</tr>
<tr>
<td>SUB</td>
<td>$\vdash \Gamma_1 &gt;_\kappa \Gamma'$ $\Gamma' \vdash e : t'$ $\vdash \Gamma \vdash e : t' \vdash t' \vdash t$</td>
</tr>
</tbody>
</table>

- Most rules are the usual ones from the linear $\lambda$-calculus.
- The Sub rule signals the insertion of a bounded buffer at its application point.
- The Fix rule ensures the productivity of recursive definitions.
The Type System

Most rules are the usual ones from the linear λ-calculus.

The **Sub** rule signals the insertion of a bounded buffer at its application point.

The **Fix** rule ensures the productivity of recursive definitions.

The **Scale** rule signals the insertion of a local time scale.
The Type System

Most rules are the usual ones from the linear $\lambda$-calculus.

The Sub rule signals the insertion of a bounded buffer at its application point.

The Fix rule ensures the productivity of recursive definitions.

The Scale rule signals the insertion of a local time scale.