Synchronous Functional Programming

Marc Pouzet
LRI & INRIA & IUF

Digiteo Meeting
October 2008, 2nd
Typed Functional Programming

Program in a **mathematical language** as an attempt to achieve code with zero defect. High-level languages abstracting some details to focus on *what* computes a system.

A computation is a sequence of reductions:

\[
\text{fact}(3) \rightarrow 3 \times \text{fact}(2) \rightarrow 3 \times 2 \times \text{fact}(1) \rightarrow 3 \times 2 \times 1 \rightarrow 3 \times 2 \rightarrow 6
\]

**Follow few principles:**
- Function composition
- Types as specifications/properties of these functions
- A method to check that a function agrees with its type

**A rich collection of languages:**
- Lazy languages restricted to *pure* functions (without side-effect): Haskell, etc.
- Strict languages: Objective Caml, StandardML, etc.
- Proof assistants (e.g., Coq) to write **total** functions (which always terminate)

An **important vehicule of ideas** for other languages and the use of formal methods in industry (Esterel-Tech., Microsoft, etc.)
Real-time Systems

Focus on systems which continuously interact with each others.
– with a **physical environment** (e.g., fly-by-wire command, control-engine)
– or **other digital devices** (e.g., phone, TV boxes)

Real time is always **related to the environment** and is not an absolute notion. To ensure safety, think of "**what is the worst case**"?

The environment is often not precisely known: most systems run in **closed-loop**

How can we program those systems, focusing on the **functionality** and abstracting from subtle implementation details?
What is new, why do we need mathematical languages?

Conciliate three notions:

- a **formal** (and computable) model of time
  - express deadlines, simultaneous events, etc.
- **parallelism** to describe complex systems from simpler ones
  - control *at the same time* rolling and pitching
  - *closed-loop* systems (the controller and the plant run in parallel)
- **statically guaranty safety properties** (both functional and non functional)
  - determinism, dead-lock freedom
  - execution in bounded time and memory

**Safety is important:**

- critical systems: fly-by-wire, braking, airbags, etc.
- some systems do not have a stable position (plane?)
  - properties must be guaranteed statically: “**dynamic**” = “too late”
A bit of History

In the 80’s, several teams invented (at about the same time) languages dedicated to the design/implementation of control-systems.

- Lustre (Caspi & Halbwachs, Grenoble) : data-flow (block-diagrams), functional model (deterministic);
- Signal (Benveniste & Le Guernic, Rennes) : data-flow but relational (to define also non-deterministic systems);
- Esterel (Berry & Gonthier, Sophia) : hierarchical automata and process algebra

Base it on the **mathematical culture and models** of the field of embedded control-systems

Quite a successful story: security systems in nuclear plants (Schneider Electric), fly-by-wire (Airbus A340-380), automotive, trains, etc.
Control-theory, Signal Processing, Electronic

- A discrete signal/event is a stream
  
  → *stream equations, Z transforms*
  
  → *graphical formalisms (block-diagrams) to represent these networks*
  
  - manual transcription of these equations into imperative code
  
  - hard and error-prone

\[ Y_0 = bX_0, \forall n Y_{n+1} = aY_n + bX_{n+1} \]

The idea of Lustre:

- write stream equations as **executable specifications**
- provide **static analysis/verification** tools and a compiler to produce code
- the generated code is **correct-by-construction**
The Synchronous Model of Time

Separate the **functionality** of the system from its **implementation**.

- Time is a **logical notion** as a sequence of atomic reactions of the system. 
  *think as if the machine was infinitely fast and react instantaneously*
- Is my abstraction reasonable? Is the **machine fast enough**?

  **Worst Case Execution Time analysis**

This coincide with the **zero-model** of synchronous circuits but for software

**Important consequences**:

- Specifications become mathematical objects which can be statically verified, transformed and compiled
- They become portable and can be reused
Synchronous Functional Programming

Lustre is a (first-order) functional language: a system is a stream function and we write invariants

\[ y = a \times \text{pre}(y) + z; \]
\[ z = b \times x \]

for \( \forall t \in D. \{ y_t = (a \times \text{pre}(y) + z), y_{t-1} + z \wedge z_t = b \times x_t \} \)

Questions:
- increase modularity/expressiveness, re-usability (software components): E.g., type synthesis, polymorphism, higher-order, etc.
- mix data and control-dominated systems in a unified model
- more dedicated static analysis to ensure safety properties
  E.g., does the program behave synchronously? Is it causal? Is it deterministic?
- (mathematically) certify code-generation? Improve compilation techniques?

What can we learn from the relationships with typed functional languages?
Lucid Synchrone

How to extend Lustre in a conservative way (without breaking it) ?

Build a “laboratory” language
– study and prototype extensions of Lustre
– experiment things, language extensions/static analysis and manage all the compilation chain

Follow a few principles :
– be conservative wrt the Lustre semantics
– formulate the synchronous data-flow model into the typed lambda-calculus
– functional composition, static properties as (special) types
– modularity everywhere (type analysis, separate compilation)
Clocks as Types

How to mix several time scales in a safe manner?

- **multi-sampled** systems (software), **multi-clock** (hardware)
- what does it mean to communicate between two software components which do not agree on a common time scale?
- **statically detect** possible synchronisation issues

\[
\begin{align*}
X &= X_0 \ast \ast \ast X_1 \ast X_2 \ast \ast X_3 X_4 \ast X_5 \ast \ast \ast \\
Y &= \ast Y_0 Y_1 Y_2 \ast Y_3 \ast \ast Y_4 \ast Y_5 \ast \ast \ast 
\end{align*}
\]

- Express the **clock** information as a **type** which express the instants where a value is defined
- Express the verification as a (classical) type system. Can I prove?

\[ H \vdash e : ct \]

- Safety property: every well-clocked program can be executed synchronously
Unifying Data-dominated and Control-dominated Systems

**Data-dominated systems**: typically from periodic sampling of continuous systems
- simulation tools (e.g., Simulink), programming tool (e.g., SCADE)

**Control-dominated systems**: discrete control, transition systems, automata
- StateFlow, StateCharts or synchronous models (e.g., SSM, Esterel)

Real systems rarely fall in one of these category
- systems have **modes**: take-off, landing, full-flight
- each mode is defined by its control-law, naturally described by stream equations
- the control part is made as an automaton which activate some of the modes

Most tools provide means to combine both: Simulink/StateFlow, SCADE/Esterel
SSM, Ptolemy Mocs, etc.

This is rather ad-hoc and there is no real unified semantics of the whole.
Cruise-control in SCADE+SSM (V5)
Unifying Data-flow and Control-flow

Follow a **clock-based approach**. Take a basis synchronous data-flow with clocks (e.g., Lustre)

- efficient code generators exist
- in the case of SCADE, the code generator is **certified** (DO 178B)
- clocks play a central role in synchronous compilers (semantics/optimisation)
- clocks are about “when” a data is ready

Extending the basic language with rich automata constructs and define a **translation semantics**

- a clock-preserving program transformation into the basic language

\[
H \vdash e : \texttt{ct} \Rightarrow H \vdash T(e) : \texttt{ct}
\]

- reuse the existing code generation techniques
- the final code appeared to be as good than dedicated techniques
Practical Applications

This embedding of synchronous data-flow into a functional setting is fruitful.

Several features originally introduced in Lucid Synchrone are now integrated in industrial tools.

- the ReLuC compiler of SCADE is based (and improves) techniques introduced in Lucid Synchrone
- same philosophy: types everywhere, modularity, etc.
- program constructs (e.g., \texttt{merge}), control-structures
- static analysis (initialisation, clock calculus)
- design/semantics of SCADE 6

Now new applications for CAD tools, mixing both programming and simulation.
What Else?

Can we relax the synchronous model to address simulation problems and general reactive systems (e.g., graphical interfaces, games)?
Reactive Programming

Fundational work by Boussinot (Reactive C, Loft, etc.). ReactiveML (by Mandel)

**Simulation of sensor networks (VERIMAG and FT, 2006-2008)**

- The system is both real-time and dynamical
- Global simulation: each node, the interaction between nodes and its environment, simulation aspects (display, computing metrics, etc)...

Example: Simulation of the power consumption in a sensor network
Conclusion and Perspectives

Mixing discrete/continuous time
- can we have more faithful models by integrating programming and numerical simulation techniques (for continuous laws)?
- a unified model/language for both discrete and continuous time with clean semantics?

Video Intensive Computation
- Kahn Process Networks widely used (e.g., NXP)
- A synchronous model to program parallel architectures?

Certified compilation
- do not program into C anymore (or rarely) embedded software
- only prove the specification, not the code
- can we certify a synchronous compiler using a proof assistant (e.g., Coq)?
References

– Synchronous Kahn Networks [ICFP’96]
– Clocks as Dependent types [ICFP’96]
– Modular compilation of higher-order stream functions [CMCS’98]
– ML-like clock calculus [EMSOFT’03]
– Type-based causality analysis [ESOP’01]
– Initialisation analysis [SLAP’02, STTT’04]
– Type-system and higher-order [EMSOFT’04]
– Mixing data-flow and hierarchical automata [EMSOFT’05, EMSOFT’06]
– N-Synchronous Kahn Networks [EMSOFT’05, POPL’06, APLAS’08]
– Formalisation of a synchronous compiler [LCTES’08]
– ReactiveML [PPDP’05, SLAP’08]