Synchronous Dataflow Pattern Matching

Grégoire Hamon
Chalmers Institute of Technology
Göteborg

Synchronous Dataflow Languages

- Dedicated to the design of reactive systems
- And quite successfull at that! numerous industrial applications
- Program close to the description of the system
- Simple semantics, strong properties
- Efficient compilation methods

Synchronous Dataflow Languages

(Small) complains:

- Few data structures
- No control structures
- Both are needed and asked for by users
- This need is becoming critical as reactive systems become more and more common and grow in complexity

Lucid Synchrone

- A functional extension to Lustre
- A functional language (ML) over streams
- Stream operations and clock mechanism similar to Lustre
- Clocks expressed as types, polymorphic, infered
- Created to study the link between functional and synchronous dataflow languages

Synchronous variants

Existing solutions in functional languages? Variant types

- Key features in functional languages
- Clean and efficient way to define data structures
  - code easier to write, close to the specification
  - verification easier
- A control structure through the pattern matching operation

Variant types

- Defined by constructors:

```
type event = Left | Middle | Right
```

- Constructors can have arguments:

```
type event = Left | Middle | Right | Nbr of int
```

- We don't consider recursive types

Lifting variants to streams

- We already lift basic types:

  ```
  0 3 9 3 2 4 5 1 8 ...
  ```

  is of type int (ie of type stream of int).

- Variants are lifted in the same way:

  ```
  type bool = True | False
  True False False True False ...
  ```

  is of type bool.

  ```
  Left Left Right Middle (Nbr 12) Left ...
  ```

  is of type event.

Pattern matching - on streams

- We want to define a control structure: only the branch matching the argument should be executed.
- So not an extension of the conditional!

  `if cond then e1 else e2`

  both `e1` and `e2` are computed.
- Control can be expressed by means of clocks.
- We are going to define a clock operator.

Control and clocks

- The clock mechanism allows combining streams evolving at different speed in a safe way (ie. guarantees reactivity).
- The clock of a stream is its pace.
- Special operators acting on clocks (`when`, `merge`).
- A control structure is an operator on the pace of the computations, it is thus a clock operator.

Synchronous pattern matching

- Each branch defines a clock.
- The right-hand side of the branch should be on this clock
- The pattern matching combines all those back to the clock of the argument.
- those branch-clocks are:
  - exclusive
  - complementary

Using branch clocks

- Variables defined by the pattern are naturally on the branch clock.
- We need the ability to filter fast streams down to the branch clock
  We introduce a way to name the branch clock in the pattern.

Pattern matching - example

```
type event = Left | Middle | Right | Nbr of int

let up_down event = out where
  rec out = 0 -> match event with
                 | Left on c => (pre out) when c + 1
                 | Middle => 0
                 | Right on c => (pre out) when c - 1
                 | Nbr i => i
```

Pattern-matching's semantics

$$\mathcal{R}, [v/x] \vdash e \xrightarrow{P_j(v_j)} e'$$

$$\mathcal{R}, [v/x], [v_j/x_j] \vdash d_j \xrightarrow{v} d'_j$$

$$\forall i \in \{1, ..., n\} \text{ such that } i \neq j,\ \mathcal{R}, [v/x], [[]/x_i] \vdash d_i \xrightarrow{[]} d'_i$$

$$\mathcal{R} \vdash \begin{array}{l} x = \texttt{match}\ e\ \texttt{with} \\ \quad \mid P_1(x_1)\ \texttt{on}\ c_1 \Rightarrow d_1 \\ \quad \ldots \\ \quad \mid P_n(x_n)\ \texttt{on}\ c_n \Rightarrow d_n \end{array} \xrightarrow{[v/x]} \begin{array}{l} x = \texttt{match}\ e'\ \texttt{with} \\ \quad \mid P_1(x_1)\ \texttt{on}\ c_1 \Rightarrow d'_1 \\ \quad \ldots \\ \quad \mid P_n(x_n)\ \texttt{on}\ c_n \Rightarrow d'_n \end{array}$$

Pattern-matching's clock

$$\frac{\begin{array}{c}\mathcal{H}, [x : cl] \vdash e : cl \\ \forall i \in \{1, ..., n\} : \ \mathcal{H}, \mathcal{H}_i \vdash P_i : cl \ \texttt{on} \ c_i \quad \mathcal{H}, \mathcal{H}_i, [x : cl] \vdash d_i : cl \ \texttt{on} \ c_i \\ c_i \notin fv_{cl}(\mathcal{H}), \ \ \mathrm{Dom}(\mathcal{H}_i) = fv_{P_i}\end{array}}{\begin{array}{l}x = \texttt{match} \ \ e \ \texttt{with} \\ \mathcal{H} \vdash \quad | \ P_1 \ \texttt{on} \ c_1 \Rightarrow d_1 \qquad : [x : cl] \\ \qquad \ldots \\ \qquad | \ P_n \ \texttt{on} \ c_n \Rightarrow d_n\end{array}}$$

Compilation

- Compiled into a similar construction (ie a control structure) in the host language.
- Some care needs to be taken, the clock only gives the pace of the output.

Conclusion (1/2)

- Simple and strict extension to the language, integrates smoothly.
- Variations can be proposed.
- Very usefull in practice.
- Implemented in Lucid Synchrone.
- Used in a systematic encoding of mode-automata which is simple and produces efficient code.
- A simplified version is implemented in ReLuC.

Conclusion (2/2)

Control structure using clocks?

- Complex clocks.
- But keeps the semantics simple!
- It needs dedicated constructs, like the `match` to make it simple.
- Clock inference is essential.