[Demo Abstract] Sound and Soundness – Practical Total Functional Data-Flow Programming

Baltasar Trancón y Widemann
Ilmenau University of Technology, Ilmenau, DE
baltasar.trancon@tu-ilmenau.de

Markus Lepper
semantics GmbH, Berlin, DE
post@markuslepper.eu

Abstract

The field of declarative data-stream programming (discrete time, clocked synchronous, compositional, data-centric) is divided between the visual data-flow graph paradigm favored by domain experts, the functional reactive paradigm favored by academics, and the synchronous paradigm favored by developers of low-level systems. Each approach has its particular theoretical and practical merits and target audience. The programming language SIG has been designed to unify the underlying paradigms in a novel way. The natural expressivity of visual approaches is combined with the support for concise pattern-based symbolic computation of functional programming, and the rigorous, elementary semantical foundation of synchronous approaches. Here we demonstrate the current state of implementation of the SIG system by means of example programs that realize typical components of digital sound synthesis.

Categories and Subject Descriptors H.5.5 [*INFORMATION INTERFACES AND PRESENTATION*]: Sound and Music Computing—Methodologies and techniques; D.3.2 [*PROGRAMMING LANGUAGES*]: Language Classifications—Applicative (functional) languages, Data-flow languages

Keywords data flow; real time; sound synthesis; stream programming

1. Context: The Total Functional Data-Flow Language SIG

For many computations a static mapping from input to output values is not sufficient; outputs need to change over time according to corresponding changes in inputs and/or internal state. Applications range from reactive systems, in a context with input devices, sensors or communication channels, to processors and generators of time-series data, such as audio signals or dynamic simulations in numerous branches of science, engineering and digital arts.

In [2] we have discussed current programming approaches to such systems, their respective merits and shortcomings, and presented our design of a core programming language, SIG, that instantiates a novel approach and covers a restricted but pervasive and practically important class of time-dependent computations.

FARM '14, September 6, 2014, Gothenburg, Sweden.

ACM 978-1-4503-3039-8/14/09.
http://dx.doi.org/10.1145/10.1145/2633638.2633644

We assume computations to occur discretely in time, driven by global clocks with known rates. All computational components are synchronized unless explicitly decoupled by resampling connectors. All data flow is conceptually instantaneous unless explicitly delayed. Instantaneous data flow must be cycle-free. Except for explicitly delayed data, there is no persistent state. Computations must not diverge or abort spontaneously. In summary, programs denote online (causal, potentially real-time) total functions on infinite data streams.

1.1 Elementwise Computation, Delay and State

As usual in the data-flow style, computations are specified as-if per element, with no explicit reference to data streams as wholes. Indeed most atomic computational components are stateless, with the notable exception of delay components, of which we consider only a single-step delay δ for simplicity reasons. Complex components with many kinds of interesting and practically relevant stateful behavior can be constructed from these.

The SIG front-end language, where state is implicit in delay, is translated to an intermediate representation (IR), where state is explicit. The atomic components of this IR are modeled (after the style of the schema notation in the Z formal method) as quaternary relations. A single step of a computation that globally takes a stream of As to a stream of Bs is represented by a relation of type $S \times A \leftrightarrow B \times S$, where the two occurrences of S denote pre- and post-state, respectively (see Figure 1).

Stream semantics are obtained by infinite replication of a step, with the post-state of each instance equaling the pre-state of the next. The state space S and the initial state $s_0 \in S$ are inferred inductively over the syntactic structure, from the use of individual delay components. Note that each delay component is a trivial relation with $S = A = B$, and $a = s'$ and $s = b$.

The IR is in static single-assignment (SSA) form, state of the art in low-level compilers, and retains the high degree of fine-grained parallelism typical of data flow languages, where operations are only partially and implicitly sequenced by data dependency. Control flow in the functional style of pattern matching on algebraic data types is integrated orthogonally by creative abuse of the data-flow merging φ-nodes of SSA. The combination of patterns and delayed feedback loops gives an effective and convenient expression of stream transducer automata.

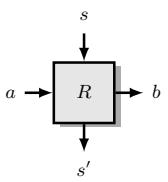

Figure 1. Stateful single-step computation model

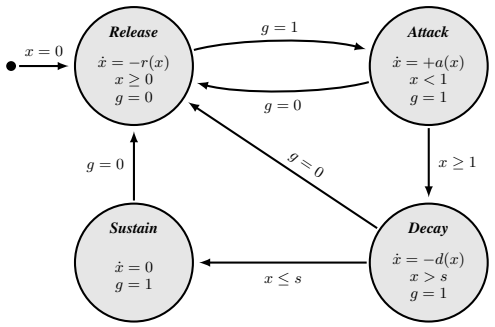

Figure 2. ADSR hybrid automaton specification

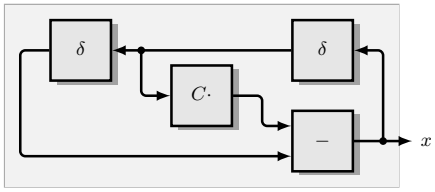

Figure 3. Harmonic oscillator data-flow graph

```
data State = { A | D | S | R }

def adsr = [ g : bool -> x' : real
  var p, p' : State; x : real
  where
  x = x' ; 0            -- (";" is delay with initial
  p = p' ; R            --  value, aka "followed-by")
  x' = case p of {
    A -> min(1, x + a(x))      --     (guards against
    D -> max(s, x - d(x))      --        overshooting
    S -> x
    R -> min(0, x - r(x))      --  by discretization)
  }
  p' = case p of {
    R if  gate                 -> A
    A if  gate && out' >= 1.0  -> D
    D if  gate && out' <= s    -> S
    else                    -- no implicit first-fit
    _ if !gate                 -> R
    else
    _                          -> p
  }
]
```

Figure 4. ADSR SIG program

```
def osci = [ -> x : real
  var y : real
  where
  y = A ; x
  x = (C * y) - (B ; y)
]
```

Figure 5. Harmonic oscillator SIG program

2. Demonstration: Sound Synthesis Components

We demonstrate the current state of implementation of the SIG system by means of example programs that realize typical components of digital sound synthesis.

2.1 The SIG Programming System

A prototype compiler and runtime environment for the SIG language has been implemented in Java. It features a front end for textual programming (integration with visual editing is planned for future work), and translation to the SSA-based IR, as described in [2].

The back end as currently implemented emphasizes immediate execution, with the compiler and runtime environment sharing a single Java virtual machine (JVM), in order to support dynamic interactive programming without recurring to external tools. The IR is translated to threaded code for a light-weight interpretation layer on top of the JVM, and can optionally be translated to JVM bytecode for more direct execution at the expense of instrumentation.

Both variants are designed for good interoperability with the JVM just-in-time compiler, and hence lead to binary code with decent performance characteristics. The demonstration will show SIG components running in a real-time setting with online performance monitoring.

2.2 Sound Applications

The application domain of digital sound synthesis has several requirements that can serve to highlight and evaluate the features of SIG particularly well:

- Synthesizers can be run both offline and in (soft) real time.
- Components running at different rates (typically *audio* and *control* rate) need to interact.
- Standard algorithms make heavy use of delayed interference and delayed feedback, as well as essentially discrete, symbolic data types; see below.
- The human ear is a merciless testing device that can easily detect many computational deficiencies, such as phase errors and off-by-one bugs in loops resulting from careless discretization.
- Users are often artists without formal training in programming languages; they require a simple and safe approach to program semantics, and can benefit from functional typing discipline.

2.3 Example Components

A typical example of a control-rate sound component with symbolic state, delayed feedback and control flow (transitions), namely the ADSR envelope model, is shown in Figures 2/4 as a continuous-time hybrid automaton and SIG program, respectively. (Shape parameter functions a, d, r and constant s are omitted.) Contrast the syntactic and semantic treatment of time and control in functional reactive programming, e.g. in [1].

A complementary example of an audio-rate sound component with purely numerical data and delayed interference, namely a harmonic oscillator (sine wave generator) model, is shown in Figures 3/5 as a visual data-flow graph and SIG program, respectively. (Phase, amplitude and frequency parameters A, B, C are omitted.)

For demonstration purposes, the generic SIG runtime environment has been extended with a real-time control- and audio-rate scheduler, sound output and GUI input controls. The details of which sound synthesis components will be demonstrated in which configuration depends on the state of the implementation, which is currently very much in progress, and will be decided on short term.

References

[1] G. Giorgidze and H. Nilsson. Switched-on Yampa: Declarative programming of modular synthesizers. In *Practical Aspects of Declarative Languages (PADL 2008)*, volume 4902, pages 282–298. Springer, 2008.

[2] B. Trancón y Widemann and M. Lepper. Foundations of total functional data-flow programming. In N. Krishnaswami and P. B. Levy, editors, *Mathematically Structured Functional Programming (MSFP 2014)*, volume 153 of *EPTCS*, pages 143–167, 2014.