Laminar Data Flow: On the Role of Slicing in Functional Data-Flow Programming

Research Paper

Baltasar Trancón y Widemann[1,2] and Markus Lepper[2]

[1] Technische Universität Ilmenau, Ilmenau, DE
[2] `<semantics />` GmbH, Berlin, DE

Abstract. We use the concept of laminar flow, as opposed to turbulent flow, as a metaphor for the decomposition of well-behaved purely functional data-flow programs into largely independent parts, necessitated by aspects with different execution constraints. In the context of the total functional data-flow language Sig, we identify three distinct but methodologically related implementation challenges, namely multi-rate scheduling, declarative initialization, and conditional execution, and demonstrate how they can be solved orthogonally, by decomposition using the standard program transformation technique, slicing.

1 Introduction

In fluid dynamics, *laminar flow* is an ideal transport process where a fluid flows in essentially parallel layers (lamina) at *different speeds*, without *turbulent* interference. We borrow the term as a useful metaphor for the benevolent properties of purely functional data-flow programs, in particular their decomposeability. Form this perspective, we investigate the various uses of decomposing a data-flow network into lamina, in the context of an effective language implementation. All our uses are concerned with lamina delimited by a functional *aspect*, that is, a data-flow closure of variables of interest. As such, they are instances of the well-known program transformation technique *slicing* [16].

This semiformal presentation retains a high level of abstraction from technical details throughout, in order to make the conceptual uniformity and naturality of the approach stand out, and the overarching story concise and readable. The issues of formally precise definition, soundness, completeness and complexity of methods are out of scope here, to be discussed in technical companion papers.

The structure of this paper and its novel contributions are organized as follows: Section 2The Sig Languagesection.1.2 introduces the basic semantic design of the Sig language, as defined in previous technical papers [13]. Section 3Slicing for Multi-Rate Data Flowsection.1.3 specifies the treatment of multi-rate systems in Sig, so far characterized largely by example [15]. Section 3.1Transparent Local Schedulingsubsection.1.3.1 gives a novel local scheduling algorithm for Sig multi-rate systems. Section 4Slicing for Declarative Initializationsection.1.4 summarizes the Sig approach to delayed computations from [13], and then gives

a semiformal but complete specification of the novel extension to non-literal initial value expressions. Section 5Slicing for Conditional Executionsection.1.5 motivates and discusses a transformation to conditionally executed code, so far published only in German as a Master's Thesis [9].

2 The Sig Language

The Sig language is a novel, total, purely functional data-flow programming language [12,13]. Its aim is to allow the expression of synchronous data stream processing algorithms in an elegant declarative style, with semantics clean and simple enough for domain experts without professional computer science background to experience programming as an orderly mathematical activity rather than an exercise in some 'black art' of tinkering and hacking.

Central to the semantics of Sig is a compositional view on synchronous data-flow computations, no matter whether primitive operations, subclauses, or complex networks, as clocked Mealy machines with private state. In [13] we have specified the formal semantics framework and a stack of program transformations that normalize higher-level functional programs into machines, compare also [8]. State spaces are inferred from the use of quasi-functional *delay* operators; the programmer never manipulates state variables directly.

Data flow is synchronous in a strict sense: all values are communicated as if by shared memory, where many readers and a single writer are arbitrated by an external clock. Race conditions, messages, events and other observable side effects are forbidden by the semantics. The following additional features of Sig semantics are of particular interest for the present paper, as they each give rise to a different application of program slicing:

1. The writer and reader of a variable are implicitly synchronized, that is, operating at the same rate; the writer always before the reader for a given clock tick. The exceptions are *up*- and *downsampling* connectors which transmit data between subnetworks operating at distinct, interleaved clock rates.
2. A delay operator applied to a data stream prepends one (or more) values to the stream. These initial values may be defined by complex expressions, with the possibility to share work between the initialization and running phase of a network, and the obligation to check for causality violations by initial values depending on on-line input.
3. Sig expressions are totally functional, that is, they produce no side effects during their evaluation (*pure*), and they may neither diverge nor abort nor block (*total*). Thus control flow can be implemented transparently, either as conditional evaluation of alternative subexpressions, or as 'posthumous' selection of alternative subresults, whichever is more convenient. The transformation of functional front-end programs into the machine representation naturally yields the latter, but many sequential execution platforms favor the former.

The following sections explore these applications in turn, in the same order as they occur in the Sig compiler pipeline.

$$\left[\frac{x \to s}{s := 0\,;(s + x)}\right] \qquad \left[\frac{x \to s}{s := (0\,;s) + x}\right]$$

Fig. 1. Simple Sig component definitions

Since the actual front-end notation of Sig is irrelevant for the present discussion, we present example programs in pseudocode.

Components are the first-class citizens of the language. They can be thought of as stream-processing functions, but in contrast to lambda expressions, inputs and outputs are notated symmetrically; both are named and can occur in any multiplicity. We enclose component definitions in brackets, enumerate inputs and outputs without giving their types, separated by a line from a block of assignments that constitutes the component body, and use mathematical operators and constants with their obvious semiformal meaning. For instance, the example components depicted in Fig. 1Simple Sig component definitionsfigure.1.1 each have a single input x and output s, respectively, and operate on some unspecified numerical data type.

We write $i\,;s$ for the initialized single-step delay operator with initial element i prepended to delayed stream s (sometimes written *fby* or $\gg$ in other data-flow languages). That is, if the expression s attains the stream of values s_n for $n = 0, 1, \dots,$ then $t = i\,;s$ attains values t_n where $t_0 = i$ and $t_{n+1} = s_n$. For instance, the two examples in Fig. 1Simple Sig component definitionsfigure.1.1 each define a component that outputs a cumulative sum s of its input stream x.

In the left component, the sum is initially zero, and each input element is added by the next clock tick. Hence this version has a latency of one tick, whereas the right component is latency-free; each input element contributes to the sum immediately. That is, $s_n = \sum_{k<n} x_k$, or $s_n = \sum_{k\leq n} x_k$, respectively. Note the pattern of delayed feedback that is ubiquitous in synchronous data-flow algorithms; other forms of recursion are forbidden in Sig. Component bodies are understood as systems of equations; evaluation order is implicitly constrained by data-flow dependencies only.

3 Slicing for Multi-Rate Data Flow

Algorithms implemented in Sig are on-line side-effect-free computations on data streams. Streams are accessed in a very disciplined way: there is no random access, only the element associated with the current clock tick is available, and if past elements are needed they must be retained explicitly, using delay. In summary, data flow behaves as if each conceptual stream is realized as a single clocked buffer variable.

While this model of communication is rather restrictive, it is very easy to grasp and use correctly and reliably, and there are various different application domains where algorithmic requirements fit this pattern neatly. In particular,

Sig algorithms can run in real time, given sufficient computational resources, because they never violate causality or productivity: values may never depend on the future or circularly on the present, nor take infinitely many steps to compute. We have demonstrated the use of the Sig language and its Java-based execution system in moderate real-time settings by creating a simple but nontrivial, polyphonic live audio synthesis system with interactive MIDI input [14].

The audio domain has a characteristic feature shared by many other real-time application fields: subsystems operate at various rates, with the slow parts controlling parameters of the fast parts (*modulation*), and the fast parts in turn providing summary information to the slow parts for feedback (*aggregation*). For instance in audio parlance,

- *wave generators* operate at *audio rate*, such as 44 kHz (CD), 96 kHz (studio);
- *modulators* of parameters such as pitch, volume and filter shape operate at *control rate*, defined as either a fraction, such as 1/64, of audio rate, or as a fixed rate, such as 1 kHz;
- *notes* and other *sequencer events* are controlled at a yet much slower rate, such as the MIDI resolution of 24 per quarter note, or the infamous 120 bpm 'techno' beat;
- some computations concern only *initialization*, and operate at rate zero.

Data-flow networks written in Sig are not declared explicitly to operate at particular rates. Rather, they constrain usable rates implicitly at two different levels of abstraction:

- The abstract program itself imposes a system of *qualitative* constraints, that is, equations and inequations between the rates of inputs and outputs, by its formal *synchronization* properties.
- Any concrete implementation imposes additional *quantitative* constraints, that is, ranges of achievable rates, by its technical *throughput* limits.

Since the latter can only be discussed properly in very detailed technological context, we focus here on the former, which can be understood in terms of a few language primitives and a static analysis.

The default behavior of primitive operations in Sig is to synchronize their in- and outputs. Assume that a program has been reduced to a core representation in static single-assignment (SSA) form, as discussed in detail in [13]. Then any assignment of the form

$$y_1, \dots, y_n := f(x_1, \dots, x_m)$$

that is, any primitive hyperedge of the data flow graph, induces equations on the rates of all concerned variables:

$$R(x_1) = \cdots = R(x_m) = R(y_1) = \cdots = R(y_n) \tag{1}$$

It follows for a whole network, that variables are synchronized if and only if they are connected by a path; data-independent subsystems can run at independent

rates. However, this does not yet allow for interference such as modulation or aggregation. To this end, we add *directed resampling* primitives to the language. The operation

$$y := \mathrm{upsample}(x)$$

is a functional identity, such that $x = y$ holds instantaneously at all times, but y is allowed to be (re)used at a faster rate than x is produced.

$$R(x) \leq R(y) \tag{2}$$

Note that data flow from slow to fast subsystems is taken as instantaneous: whenever clock ticks for x and y coincide, y reflects the new value of x immediately. This also allows us to subsume constant values consistently as data streams of rate zero, provided that all clocks begin to tick simultaneously at initialization time of the system ('big bang').

By contrast, the converse operation

$$y := \mathrm{downsample}(i, x)$$

allows y to be used at a slower rate than x is produced. Namely, at each clock tick of y, the most recent value of x is observed. Whenever clock ticks for x and y coincide, y reflects the *previous* value of x. Obviously, the value of y at initialization time is not determined by x; this is the purpose of the additional, type-compatible input i; compare the analogous, synchronized expression $i \,;\, x$.

$$R(y) \leq R(x) \tag{3}$$

The asymmetry of these two operations ensures that the scheduling strategy of subsystems at different rates is independent of the actual program: the instantaneous data flow at each coincidence of ticks is always from slow to fast.

Note that, when components are composed, their rate constraints accumulate. Since all inequalities are non-strict, there composite system is always satisfiable. However, it is possible for rates to be equated emergently in the composite, say, by an inequation $R(x) \leq R(y)$ arising from one component, and a converse inequation $R(y) \leq R(x)$ independently from another component. Resampling operations along the path are rendered ineffective; the rate analysis pass should notify the programmer about this potential usage error.

As an example of rate analysis and separation, consider the following manifestly multi-rate program:

$$\left[\begin{array}{ll} \multicolumn{2}{l}{() \to \mathit{wave}, \mathit{high}} \\ \hline \mathit{wave} := \mathrm{upsample}(\mathit{amp}) \cdot \sin(\mathit{phase}) & m := \max(\mathrm{abs}(\mathit{wave}), (0 \,;\, m)) \\ \mathit{phase} := 0 \,;\, (\mathit{phase} + \alpha) & \mathit{high} := \mathrm{downsample}(0, m) \\ \mathit{amp} := 1 \,;\, (\mathit{amp} \cdot \gamma) & \end{array}\right]$$

It produces an oscillation *wave* with current phase *phase* and amplitude *amp*, which increase arithmetically and geometrically with parameters α and γ, respectively. Since amplitude is a long-term modulating property in relation to phase,

the former is upsampled. Additionally the attained maximum absolute value m of $wave$ is recorded, and a downsampled copy $high$ provided for monitoring.

Note that this program is not in proper SSA form, because there remain nested expressions with unnamed intermediate variables. However, these play no significant role in the rate analysis of the program. We shall take the same liberty for harmless abbreviation in the following examples.

Note also the references to α and γ, which are parameters of the generic component definition, and become private life-time constants for each component instance. They are supplied by the higher-order programming mechanism of Sig, the details of which are out of scope here.

For the example, straightforward application of rules (1Slicing for Multi-Rate Data Flowequation.1.3.1)–(3Slicing for Multi-Rate Data Flowequation.1.3.3) finds a synchronous cluster $R_1 R(wave) = R(phase) = R(m)$, and both $R_2 = R(amp) \leq R_1$ and $R_3 = R(high) \leq R_1$. Thus for instance, setting R_1 to audio rate, R_2 to control rate, and R_3 to the refresh rate of a graphical output device would yield a consistent real-time execution context.

If different subnetworks are to be actually operated at different rates, they can no longer be implemented directly as the transition of a monolithic Mealy machine. Rather, the component should be sliced according to synchronicity, and each slice translated to machine form independently. To this end, each resampling operator is split into a fresh input–output pair of matching variables s^+, s^-, respectively, and the program is sliced backwards, based on the synchronicity partition of both original and synthetic outputs.

For our example, we obtain two synthetic variable pairs, $up^\pm$ and $down^\pm$, where $R(up^-) = R_2$ and $R(down^-) = R_1$, and thus the following three subcomponents:

$$\left[\begin{array}{l} up^+ \to wave, down^- \\ \hline wave := up^+ \cdot \sin(phase) \\ m := \max(\mathrm{abs}(wave), (0 \,;\, m)) \\ phase := 0 \,;\, (phase + \alpha) \\ down^- := 0 \,;\, m \end{array}\right] \qquad \left[\begin{array}{l} () \to up^- \\ \hline amp := 1 \,;\, (amp \cdot \gamma) \\ up^- := amp \end{array}\right] \qquad \left[\begin{array}{l} down^+ \to high \\ \hline high := down^+ \end{array}\right]$$

The runtime scheduler takes care of the 'anionic' asynchronous data flow $up^- \rightsquigarrow up^+$ and $down^- \rightsquigarrow down^+$ behind the scenes; see Fig. 2Single-rate slices of example multi-rate componentfigure.1.2. Downsampling is translated to delay at the operand rate. Note that the third component is trivial and serves only to mask the synthetic variable $down^+$ behind the original variable $high$; by contrast, the second component exposes the previously internal original variable amp as the synthetic output up^-.

3.1 Transparent Local Scheduling

Under certain mild assumptions, the slices of a component for different rates can be reassembled, as a component operating at the fastest concerned rate.

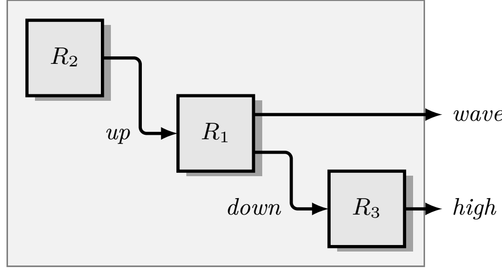

Fig. 2. Single-rate slices of example multi-rate component

The slower rates are triggered intermittently by an internal, local scheduler, whose action is transparent to the component's environment. Local scheduling may or may not be an option, depending on the technical context and the actual rate proportions; however, the mere possibility adds to the compositionality of the language, and hence merits some consideration. Furthermore, it is also interesting from a purely algorithmic perspective.

Assume that a component operates at a finite number of distinct rates, $0 < R_1 < \cdots < R_n$. Assume furthermore that these rates are *commensurable*, that is in rational proportion: there are integer numbers $0 < \rho_1 < \cdots < \rho_n$ such that $R_i = \rho_i R_0$ for some *fundamental* rate R_0, which need not be present in the component.

Equivalently, let $0 < T_n < \cdots < T_1$ denote the periods of operation, with $T_i = R_i^{-1}$. Then there are integer numbers $0 < \delta_n < \cdots < \delta_1$ such that $T_i = \delta_i T_*$ for some *atomic* period T_*; namely $\delta_i = \rho_* / \rho_i$ and $T_* = T_0 / \rho_*$, where ρ_* is the least common multiple of $\{\rho_1, \dots, \rho_n\}$.

By normalization, we obtain rational numbers $0 < \pi_1 < \cdots < \pi_n = 1$, where $\pi_i = \rho_i / \rho_n = \delta_n / \delta_i$. The time of the k-th tick of the i-th clock is given as $t_{i,k} = k \cdot \delta_i$.

Now assume that it is valid to quantize all clocks at the fastest occurring rate ρ_n, as opposed to the atomic rate ρ_*, as long as the causal order is maintained. When a tick of the (slower) i-th clock falls *between* two ticks of the (fastest) n-th clock, say $t_{n,m} < t_{i,k} < t_{n,(m+1)}$, it can be safely quantized to the *successor*, since the operational model of Sig explicitly allows instantaneous data flow from slow to fast subcomponents. From the perspective of the n-th clock, we obtain the most recent tick of the i-th clock by rounding $t_{n,k}$ first *down* to a multiple of δ_i, then we obtain its quantization by rounding *up* to a multiple of δ_n. This is conveniently expressed as $q_{i,k} = \delta_n \big\lceil \lfloor k \cdot \pi_i \rfloor / \pi_i \big\rceil$.

Note that, with respect to the exact tick sequence t_i, q_i is both rounded up to a multiple of δ_n and stretched out to match the pacing of $t_n = q_n$; whereas t_i is injective, q_i has runs of average length π_i^{-1}. Note also that $q_{i,k} \leq q_{n,k}$. Morally, the i-th component is still assumed to operate at a constant rate, reflected by the special nullary Sig primitive **dt**, which evaluates to its own clock period and is a lifetime constant for each component instance. Unless a component is run

```
var s := 0
invariant −ρ2 < s ≤ +ρ1
for each step do
  let up := s ≥ 0
  s := s + ρ1
  if up then
    s := s − ρ2
    step component 1
  end if
  step component 2
end for
```

Fig. 3. Component scheduling, Bresenham style

in an embedded context and connected to very hard real-time input/output, the micro-latency induced by quantification goes undetected.

The local scheduler, which is operated at the fastest rate R_n, needs to perform a computation of the i-th subcomponent at its k-th invocation, if and only if $q_{i,k} = q_{n,k}$. Conveniently enough, this can be achieved by a variant of Bresenham's algorithm for quantized line drawing [1]. Scheduling a commensurable two-rate component for one period of its fundamental rate is analogous to drawing a rastered two-dimensional straight line with extent $\Delta x = \rho_2$ and $\Delta y = \rho_1$: advancement by one pixel in the x and y dimension corresponds to quantized clock ticks at ρ_2 and ρ_1, respectively.

Note that we have $0 < \Delta y \leq \Delta x$, the base case of Bresenham's algorithm to which other cases are reduced. Thus at most one tick at rate R_1 happens per tick at rate R_2. The algorithm is adapted by changing the rounding mode, and omitting the main loop such that one turn is performed at each invocation of the component, and the slope is extrapolated indefinitely to the right. Fig. 3Component scheduling, Bresenham stylefigure.1.3 shows the basic algorithm in pseudocode.

For $n > 2$ components, additional counters $s_2, \ldots, s_{n-1}$ can be added. For rates R_i that also are multiples of, or may be quantized to, R_j for some $j < n$, the scheduling problem can be decomposed hierarchically. The latter approach is generally more efficient, in particular when $R_j \ll R_n$.

Fig. 4*Top* – quantization timelines; $n = 3$, $\rho_* = 30$, interval of $2T_0$ shown. *Bottom* – discrete (clock) time over continuous (real) time; tick sequences t_i as points on identity line (*dashed*), quantized time q_i as step functionsfigure.1.4 shows an example multi-rate ensemble with relative rates $\rho = (2, 3, 5)$, depicting the evolution of its three clocks over an interval of $2T_0$.

4 Slicing for Declarative Initialization

From the semantics perspective it is tempting to neglect the initial values of delayed streams as a minor detail. Indeed, the pattern seen in the preceding examples, namely delayed feedback to a monoid operation, with the monoid unit

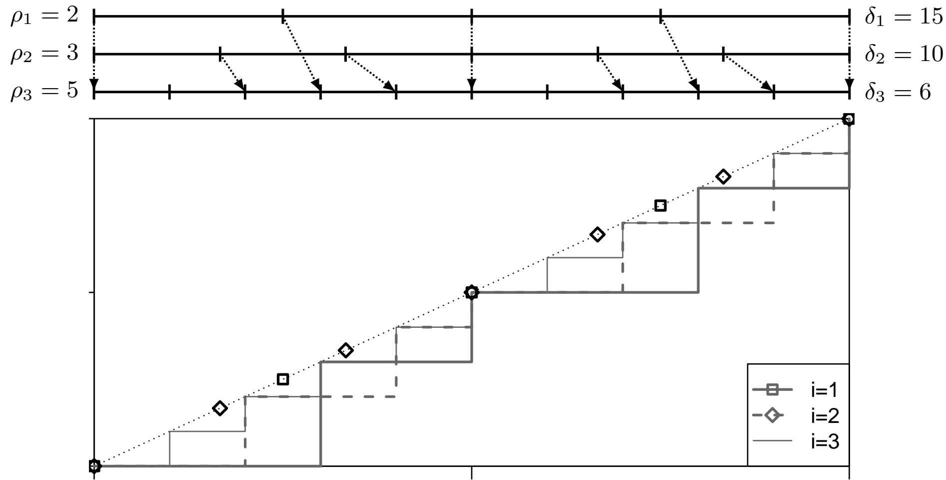

Fig. 4. *Top* – quantization timelines; $n = 3$, $\rho_* = 30$, interval of $2T_0$ shown. *Bottom* – discrete (clock) time over continuous (real) time; tick sequences t_i as points on identity line (*dashed*), quantized time q_i as step functions.

as the natural initial value, is quite common, and suggests an implicit solution by inference. Note that some clocked synchronous data-flow formalisms omit the specification of initial values altogether, for instance Faust [10]. However, not all common uses of delay fit the bill; an example is given in (†) on p. 10Slicing for Declarative InitializationAMS.2 below, after a summary of the Sig implementation of delay [13]. In this section, we discuss a slicing technique to address the issue.

Sig front-end programs are neutral about whether each name is bound to a single value or a stream. For the domain of synchronous data-flow algorithms, this is an elegant and adequate abstraction: virtually all primitive computations operate element-wise anyway, such that the distinction would provide no insight; the only, but ubiquitous exception being delay operations.

In the semantics as specified in [13], delay operations are replaced by private (buffer) state, by a syntax-directed program transformation. The resulting SSA-style intermediate representation can be read directly as element-wise formal semantics, namely as the transition rule of a Mealy machine, in the form of a quaternary relation $R \subseteq (S \times A) \times (B \times S)$, where A, B are the products of ranges of input (x) and output (y) variables, respectively, and S is the product of ranges of inferred state variables, in the double role of pre-state (s) and post-state (s'); see Fig. 5Diagram depiction and syntax for state transitionsfigure.1.5.

In this view, a single-step delay operation is simply the special case of a square identity $\delta_A = I_{A \times A} \subseteq (A \times A) \times (A \times A)$: at each clock tick, the pre-state becomes output, while the input simultaneously becomes post-state, to be output in the next cycle, etcetera.

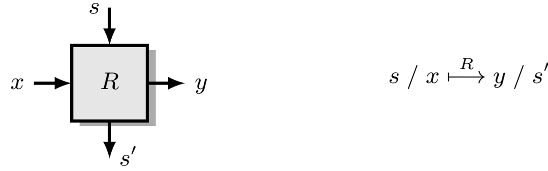

Fig. 5. Diagram depiction and syntax for state transitions.

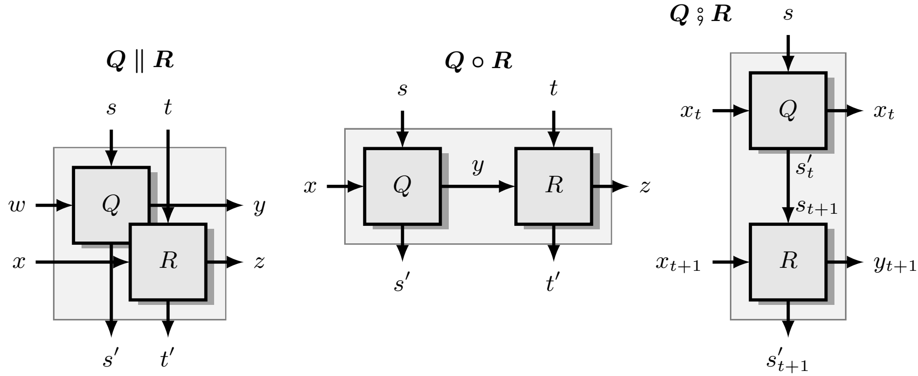

Fig. 6. Composition axes of transition relations, adapted from [13]

These element-wise transition relations can be depicted graphically and admit three different meaningful compositions, namely *parallel* ($\parallel$), *functional* ($\circ$) and *temporal* ($\fatsemi$) composition, respectively; see Fig. 6Composition axes of transition relations, adapted from [13]figure.1.6. The desired stream-wise semantics of a data-flow program is 'morally' the infinite temporal replication of the corresponding element-wise transition relation:

$$\lim_{n\to\infty} \underbrace{R \fatsemi \cdots \fatsemi R}_{n}$$

It takes an initial pre-state and a whole input stream to a whole output stream; there is no final post-state. Between clock ticks, post-state is fed back to pre-state. In [13] we have given a rigorous coinductive construction.

However, there is a catch: because the initial states are outside of this semantic interpretation, they can only be given as uninterpreted constants. In practice, one would certainly like to have the full expression language to denote complex initial values. Hence one-off initialization and repeated element-wise computation should be compiled together, and only separated for code generation by

static analysis and slicing. For example, consider the following program

$$\left[\begin{array}{l} () \to x \\ \hline x := 1 \,;\, y \\ y := (a \,/\, 2) \,;\, (a \cdot y - x) \\ a := 2 - \alpha \cdot \alpha \end{array}\right] \tag{$\dagger$}$$

which computes a very resource-efficient, numerically stable approximation of the sequence $x_n = \cos(n \cdot \alpha)$. It uses a magic constant a both at initialization and at each clock tick. The remainder of this section specifies a general program analysis and transformation in several steps, interleaved with applications to this example for immediate illustration.

The embedding of initialization expressions works as follows: For each variable v that is the result of a delay operation,

$$v := i \,;\, u$$

perform a *statification* (sic): introduce a pre–post pair of fresh matching state variables s_v, s'_v, respectively; then add a pair of statements according to the semantics of delay given earlier in this section

$$v := s_v \qquad\qquad s'_v := \iota(i, u)$$

where ι is a special primitive, a variant of the well-known ϕ operator of SSA, which selects its first operand when evaluated during initialization of the component, and its second operand otherwise. For the example, we obtain:

$$\left[\begin{array}{ll} s_x, s_y \,/\, () \to x \,/\, s'_x, s'_y & \\ \hline x := s_x & s'_x := \iota(1, y) \\ y := s_y & s'_y := \iota(a \,/\, 2, a \cdot y - x) \\ a := 2 - \$step \cdot \$step & \end{array}\right]$$

Note the slash notation to enclose the input/output interface in the state context, as in Fig. 5Diagram depiction and syntax for state transitionsfigure.1.5.

The subsequent static analysis works as follows. Several 'virtual' slices are are formed based on forward or backward data flow:

- The forward slice D (*dynamic*) for all statements depending on pre-state and/or input;
- the backward slice I (*initial*) for all statements affecting post-state, considering only the first operand of each ι operator;
- the backward slice L_0 (*loop*) for all statements affecting output and/or post-state, considering only the second operand of each ι operator; this slice is split into the subslice L of statements also contained in D plus their *immediate* data-flow predecessors, and its relative complement $\ell = L_0 \cap \overline{L}$. The intuition here is that statements in L/ℓ are directly/indirectly relevant for the loop phase, respectively.

They yield a multidimensional classification: statements…

- in $\overline{I} \cap \overline{L_0}$ are dead (they play no role for output or state);
- in $I \cap \overline{L_0}$ are computed for initialization only (they play no role for loop output or post-state);
- in $D \cap I$ are causally illegal attempts to read from a stream during initialization (they depend on loop input or pre-state but affect initial state), except for the safe case of ι operations whose first operand is not in D;
- in $D \cap L$ are recomputed at each clock tick (they depend on loop input or pre-state and affect output or post-state);
- in $\overline{D} \cap L$ are loop invariant, computed at initialization and retained as constants (they do not depend on loop input or pre-state but directly affect output or post-state);
- in ℓ are computed privately at initialization, and used by the preceding (they do not depend on loop input or pre-state but indirectly affect output or post-state).

The full classification of a statement consists of two binary decisions, namely $\{D, \overline{D}\}$ and $\{I, \overline{I}\}$, and a ternary decision, $\{\overline{L_0}, L.\ell\}$. The classification of a statement is inherited by its result variable(s).

For the example, we find that $x, y \in D \cap \overline{I} \cap L$ and $s'_x, s'_y \in D \cap I \cap L$ and $a \in \overline{D} \cap I \cap L$ (and its unnamed intermediates in $\overline{D} \cap I \cap \ell$). It follows that a needs to be retained as a constant.

This is achieved by a transformation that introduces synthetic delay with identical feedback: replace each statement of the form

$$c := e$$

where e is in $\overline{D} \cap L$, with

$$c := e \,;\, c$$

and apply ι-introduction as above. For the example, we obtain:

$$\left[\begin{array}{ll} \multicolumn{2}{l}{s_x, s_y, s_a \;/\; () \to x \;/\; s'_x, s'_y, s'_a} \\ \hline x := s_x & s'_x := \iota(1, y) \\ y := s_y & s'_y := \iota(a \,/\, 2, a \cdot y - x) \\ a := s_a & s'_a := \iota(2 - \alpha \cdot \alpha, a) \end{array}\right]$$

The ι-introduction rule has created trivial copy statements. Clean up by performing copy propagation on statified variables, substituting s_v for v, with one crucial exception: for references to statified variables v in the *first* operand of a ι operator, s'_v is substituted instead.

For the example, we obtain:

$$\left[\begin{array}{ll} \multicolumn{2}{l}{s_x, s_y, s_a \;/\; () \to x \;/\; s'_x, s'_y, s'_a} \\ \hline x := s_x & s'_x := \iota(1, s_y) \\ & s'_y := \iota(s'_a \,/\, 2, s_a \cdot s_y - s_x) \\ & s_a := \iota(2 - \alpha \cdot \alpha, s_a) \end{array}\right]$$

In the final step, two slices are computed:

– an *initialization* slice, which retains just the statements affecting post-state, and replaces each ι operator by its first operand;
– a *loop* slice, which retains all statements affecting post-state and/or output, and replaces each ι operator by its second operand.

For the example, we obtain:

$$\left[\begin{array}{l} \underline{s_x, s_y, s_a \;/\; () \to x \;/\; s'_x, s'_y, s'_a} \\ s'_x := 1 \\ s'_y := s'_a \;/\; 2 \\ s'_a := 2 - \alpha \cdot \alpha \\ \hdashline x := s_x \\ s'_x := s_y \\ s'_y := s_a \cdot s_y - s_x \\ s'_a := s_a \end{array}\right]$$

where initialization and loop are above and below the dashed line, respectively.

The execution model assumes that the initialization slice is evaluated once; afterwards and for conceptually infinitely many clock ticks, post-state is fed back to pre-state and the loop slice is evaluated. Note that, at least for common simple cases, suitable efficient implementations of state feedback can be suggested by peephole optimizations: for the example,

– the statement $s'_a := s_a$ witnesses that a is constant, and can be eliminated in the obvious way by allocating s_a and s'_a to the same storage location;
– the statement $s'_x := s_y$ witnesses that the pair y, x is a buffer queue. While this particular instance is of trivial size and needs no special attention, longer queues that are candidates for a ring buffer implementation can be found by simple flow graph pattern matching.

5 Slicing for Conditional Execution

Sig has no concept of true user-defined control flow, such as jumps, loops or recursion; the infinite unfolding of output streams is the only means of iteration. However, the language does have *pattern matching* constructs as expressive and general means of dynamic case distinction. Because all Sig expressions are totally productive (may not diverge element-wise), case distinction can be seen as a data-flow, rather than control-flow issue: the semantics allows for all alternative rules to be evaluated concurrently. Such speculative evaluation may *fail* on a matching rule because of a refuted pattern on the left hand side. In that case, the right hand side is taken to evaluate to the special value $\bot$, which can be conceived of as an exception.

In the core language, a special primitive γ is used to guard the actual result of the right hand side, conditional on the success of matching. From the set of alternatives, a second special primitive φ selects a successful rule, conceptually

catching the exceptions. All other operations are strict with respect to $\bot$. Our usage of φ differs from its classical namesake ϕ in SSA in the sense that choice is not based on incoming control flow; instead it is generally nondeterministic but avoids $\bot$ whenever possible. If rules are mutually exclusive and jointly total, as they are in a normalized well-defined pattern-matching expression, then the final result is total and deterministic. A static analysis enforces that $\bot$ is never leaked from a component.

As in the previous section, we interleave general descriptions of compilation tasks, and particular illustrations. For example, consider the following Sig program, which defines a wave generator component with an additional switch to silence the output, a *gate* in audio parlance:

$$\left[\begin{array}{l} \mathit{gate} \to \mathit{wave} \\ \hline \;\mathit{wave} := \textbf{if } \mathit{gate} \textbf{ then } \sin(\mathit{phase}) \textbf{ else } 0 \\ \mathit{phase} := 0 \; ; (\mathit{phase} + \alpha) \end{array}\right]$$

The if-then-else construct is syntactic sugar for pattern matching on the enumerated datatype $\{\mathrm{true}, \mathrm{false}\}$. Pattern matching is eliminated according to a nontrivial syntax-directed program transformation duly specified in [13], and replaced by a network of φ and γ operators. For the example, performing also statification of delay as described in the previous section, we obtain:

$$\left[\begin{array}{ll} s \;/\; \mathit{gate} \to \mathit{wave} \;/\; s' & \\ \hline \;\mathit{wave} := \varphi(a, b) & \\ \qquad\quad a := \gamma(\sin(\mathit{phase}), \mathit{gate} = \mathrm{true}) & \\ \qquad\quad b := \gamma(0, \mathit{gate} = \mathrm{false}) & \\ \mathit{phase} := s & s' := \iota(0, \mathit{phase} + \alpha) \end{array}\right]$$

The φ operator selects the output value from either of the complementary branches a and b. Each of these is given by a γ operation that yields the first operand if the constraint expressed by the other operand(s) is satisfied, or $\bot$ otherwise. In a well-typed context, exactly one branch is defined (non-$\bot$) at all times. For the example, by copy propagation and initialization slicing as described in the previous section, we obtain:

$$\left[\begin{array}{l} s \;/\; \mathit{gate} \to \mathit{wave} \;/\; s' \\ \hline s' := 0 \\ \hline \mathit{wave} := \varphi(a, b) \\ \qquad\quad a := \gamma(\sin(s), \mathit{gate} = \mathrm{true}) \\ \qquad\quad b := \gamma(0, \mathit{gate} = \mathrm{false}) \\ \qquad\quad s' := s + \alpha \end{array}\right]$$

On massively parallel execution platforms, such as FPGAs, it is perfectly reasonable and efficient to evaluate both branches independently and implement φ as a multiplexer. From this perspective, the whole body of a component is just a single basic block. By contrast, on more conventional sequential platforms, such

as ordinary CPUs and virtual machines, it is often desirable to save time by evaluating only relevant branches. To this end, statements should be organized in smaller basic blocks guarded by conditional branching instructions, as usual for conventional sequential imperative languages.

The Sig compiler, which currently targets the Java virtual machine platform, has an experimental pass for automatic sequentialization of programs with φ and γ operations, thus converting data flow into control flow. It works roughly as follows [9]:

- For each variable, determine the condition under which it is defined. Since the only sources of undefinedness are failed pattern-matching operations, this is a straightforward backwards data-flow analysis. The result is a positive propositional formula, where pattern matching primitives, γ and φ contribute literals, conjunctions and disjunctions, respectively.
- Group statements according to the definition conditions of their results.
- Nest groups according to logical implication of their conditions. Simplify nested conditions relative to their parents.
- Split each group into as few basic blocks as possible, such that inter-group data-flow dependencies do not connect blocks circularly.
- Guard the entry into each basic block by conditional branching.
- Re-interpret φ operations as their traditional SSA ϕ counterparts.
- Optionally, follow up with a standard SSA cleanup pass, which attempts to allocate all operands of a ϕ operation to the same storage location, thus eliminating it completely; otherwise, the set of ϕ operations at the beginning of a basic block is transposed to a set of simultaneous moves at the end of each predecessor block [5].

Note that it may appear simpler to sequentialize case distinctions directly as they appear in the front-end syntax. But code transformations such as common subexpression elimination and algebraic simplification are dramatically effective on the pure data-flow form, and can disrupt the original block structure.

For the example, on the whole we obtain a sequential program that can be described by the following pseudocode with conditional execution:

$$\left[\begin{array}{ll} \multicolumn{2}{l}{s \;/\; \mathit{gate} \to \mathit{wave} \;/\; s'} \\ \hline s' := 0 & \\ \hdashline \mathit{wave} := \sin(s) & \textbf{if } \mathit{gate} = \mathrm{true} \\ \mathit{wave} := 0 & \textbf{if } \mathit{gate} = \mathrm{false} \\ \quad s' := s + \alpha & \end{array}\right]$$

Note that both a and b have been re-allocated to *wave*, and all administrative moves have been eliminated thereby.

It follows that the, possibly expensive, expression $\sin(s)$ is only evaluated when necessary. However, this example is misleading about the generality of the approach as outlined above: it depends crucially on all conditional statements being stateless primitives; if calls to dynamically bound, potentially stateful subcomponents are allowed, as they are in full Sig, then naïve conditional execution is no longer safe.

Consider a variation of the preceding example, where the waveform shape is no longer a pure function, but a statically non-fixed stream generator component reference:

$$\left[\begin{array}{l} gate \to wave \\ \hline wave := \textbf{if}\ gate\ \textbf{then}\ shape()\ \textbf{else}\ 0 \end{array}\right]$$

This can be seen as a modularization, where the concerns of wave generation, including private phase state, and of gate operation are separated. The original form is restored by plugging in the following simple generator:

$$\left[\begin{array}{l} () \to wave \\ \hline \ wave := \sin(phase) \\ phase := 0\,;\,(phase + \alpha) \end{array}\right]$$

The problem with this 'morally equivalent' situation is that, by design, the component instance state referred to by *shape* is private, and the output and transition operations are tied up together in a monolithic quaternary machine relation, as discussed in the preceding section. Hence the call to the subcomponent cannot be assumed to be side-effect-free, and hence it cannot be simply omitted at some clock ticks, lest spurious local time-freezes arise. For the example, the phase of the generator would simply stop whenever the gate is shut down, which may or may not be pragmatically acceptable depending on context, but is certainly not in accordance with the principle of least surprise.

A number of partial or universal solutions to this dilemma are conceivable:

- Simply call subcomponents unconditionally, wasting any opportunity for inter-component work saving.
- Record statelessness in the type system; call only manifestly stateless subcomponents conditionally.
- The original version that has no issues can be restored by inlining; simply defer code generation until parameters are bound.
- Generally separate state transition and output production, delegate the former to a central scheduler. However, separation can be difficult because of arbitrary data-flow inter-dependencies, and decentral solutions are decidedly more light-weight and elegant.
- Create an alternative and more efficient, *tacit* variant of each component, to be used under conditions where the outputs are not needed.

The last solution has little impact on our execution model as described before, and good modularity. Furthermore, it can be implemented by straightforward program slicing, namely as the backward slice of all statements affecting post-state, with no outputs. For the simple wave generator, after initialization slicing we obtain the component on the left, and its tacit variant on the right:

$$\left[\begin{array}{c} s\ /\ () \to wave\ /\ s' \\ \hline s' := 0 \\ \hdashline wave := \sin(s) \\ s' := s + \alpha \end{array}\right] \qquad \left[\begin{array}{l} s\ /\ () \to ()\ /\ s' \\ \hline s' := 0 \\ \hdashline s' := s + \alpha \end{array}\right]$$

Note that, just like in the monolithic original example, the tacit variant saves work by not computing $\sin(s)$. The two component definitions are understood to be instantiated together, and share state.

The sequentialization algorithm can be fixed to support stateful subcomponents by adding the following clause:

- After grouping statements by condition, for each statement that calls a subcomponent c, add a corresponding call to the tacit variant c_0 under the complementary condition.

Note that, since all condition literals are about closed algebraic datatypes, the complement of a condition is again a finite positive formula. The rule is sound but redundant also for unconditional calls; the additional tacit call is unreachable by construction.

For the modularized example, we obtain:

$$\left[\begin{array}{ll} \multicolumn{2}{l}{\mathit{gate} \to \mathit{wave}} \\ \hline \mathit{wave} := \mathit{shape}() & \textbf{if } \mathit{gate} = \text{true} \\ \quad () := \mathit{shape}_0() & \textbf{if } \mathit{gate} = \text{false} \\ \mathit{wave} := 0 & \textbf{if } \mathit{gate} = \text{false} \end{array}\right]$$

In general, components may have more than one output variable. In that case, there are candidates for intermediate variants between the full and the tacit version; namely one for each subset of outputs. Each of these variants can be obtained by the same straightforward slicing policy. But because of exponentially growing number of combinations and diminishing relative gains, we do not envisage a static exploration. Nevertheless, intermediate slices may occur dynamically on execution platforms supporting run-time program specialization.

6 Conclusion

6.1 Related Work

The design of Sig draws on inspirations and results from many paradigms. Several characteristic features are inherited from the ‘French school’ of synchronous languages; in particular we are indebted to Lucid Synchrone [2] for the initialized delay operator and implicit handling of (multi-)rate. However, Lucid suffers from being based on an impure functional core language. Faust [10] has pioneered the virtue of purity and its benefits in powerful code transformations. Functional reactive programming (FRP) based on the theory of arrows [6], provides many of the elegant core properties we aim at, although in a more abstract and general, and less down-to-earth setting. Extensions of its axiomatic theory exists to deal with advanced features such as complex initialization and control flow, namely as *causal commutative arrows* [8] and *arrows with choice* [7], respectively, although a grand unifying picture remains elusive.

There are remarkably few attempts to apply slicing to functional or dataflow languages: Ganapathy and Ramesh apply slicing to a statechart variant [4].

Their definition of a slice is based on behavioral equivalence with respect to one selected output signal and thus quite different from ours. Clarke et.al. [3] apply slicing to VHDL, a declarative hardware description language. Astonishingly, they take the detour of converting VHDL into an imperative language (C) and then apply a commercial slicing tool, instead of exploiting the functional aspects of VHDL. A positive example of applying slicing to pure functional programs by pure functional means is due to Rodrigues and Barbosa [11]; while they operate very elegantly on an abstract semantic formalism (Bird–Meertens), they cannot promise that their approach does scale, what can easily be shown for our approach, which is syntax-directed and thus more directly applicable.

For the embedding of explicit case distinctions in functional data-flow formalisms, see [**?**] and [7] for monads and arrows, respectively.

6.2 Summary and Outlook

Because of the rigorously puristic and mathematically elegant design of the Sig semantics framework, very basic and easily implemented, data-flow-based program analyses and transformations go a long way. We have demonstrated the wide applicability of slicing transformations to a number of conceptually independent issues: advanced support for high-level front-end features, such as multi-rate networks with resampling; necessary support for orthogonality of basic features, such as complex expressions and initialized delay; optimized support for resource-efficient, state-transparent embedding of modular subcomponents.

The whole Sig compiler pipeline is a prototype and subject to ongoing research and development at various ends. Of the slicing applications we have presented here, initialization slicing is stably implemented in the current version of the compiler. Tacit slicing is implemented in principle, but its practical use is deferred pending the integration of the very recent, externally developed sequentialization pass. Multi-rate slicing is in the process of implementation, and the current main focus of our efforts, since it would allow us to raise our demonstrations in the application domain of audio and digital music [14] to a significantly more sophisticated level. Encouraging preliminary results and a first non-trivial case study are discussed in [15].

References

1. J. E. Bresenham. Algorithm for computer control of a digital plotter. *IBM Systems Journal*, 4(1):25–30, 1965.
2. P. Caspi and M. Pouzet. Lucid Synchrone, a functional extension of Lustre. Technical report, Université Pierre et Marie Curie, Laboratoire LIP6, 2000.
3. E. M. Clarke, M. Fujita, S. P. Rajan, T. Reps, S. Shankar, and T. Teitelbaum. Program slicing of hardware description languages. Technical report, Carnegie Mellon University, 1999.
4. V. Ganapathy and S. Ramesh. Slicing synchronous reactive programs. In *SLAP 2002*, *ENTCS* 65(5):50–64, 2002. DOI: 10.1016/S1571-0661(05)80440-2

5. S. Hack. Register Allocation for Programs in SSA Form. Phd Thesis, Universität Karlsruhe, 2007. URL: http://digbib.ubka.uni-karlsruhe.de/volltexte/documents/6532
6. P. Hudak, A. Courtney, H. Nilsson, and J. Peterson. Arrows, robots, and functional reactive programming. In *Advanced Functional Programming*, LNCS 2638, pp. 159–187. Springer, 2003. DOI: 10.1007/978-3-540-44833-4_6
7. J. Hughes. Programming with arrows. In *Advanced Functional Programming*, LNCS 3622 , pp. 73–129. Springer, 2005.
8. H. Liu, E. Cheng, and P. Hudak. Causal commutative arrows. *J. Functional Programming*, 21:467–496, 2011. DOI: 10.1017/S0956796811000153
9. A. Loth. Synthese von Kontrollfluss für eine Synchrone Datenflusssprache. Master's thesis, Technische Universität Ilmenau, 2015.
10. Y. Orlarey, D. Fober, and S. Letz. Syntactical and semantical aspects of Faust. *Soft Comput.*, 8(9):623–632, 2004. DOI: 10.1007/s00500-004-0388-1
11. T. Petricek, A. Mycroft, and D. Syme. Extending monads with pattern matching. In *Haskell Symposium 2011*, pp. 1–12. ACM, 2011. DOI: 10.1145/2034675.2034677
12. N. F. Rodrigues and L. S. Barbosa. Slicing functional programs by calculation. In *Beyond Program Slicing*. Dagstuhl Seminar 05451, 2005.
13. B. Trancón y Widemann and M. Lepper. tSig : Towards semantics for a functional synchronous signal language. In *KPS 2011*, Arbeitsbericht des Instituts für Wirtschaftsinformatik 132, pp. 163–168. Westfälische Wilhelms-Universität Münster, 2011. URL: https://www.wi.uni-muenster.de/sites/default/files/publications/arbeitsberichte/ab132.pdf
14. B. Trancón y Widemann and M. Lepper. Foundations of total functional data-flow programming. In *MSFP 2014*, *EPTCS* 154, pp. 143–167, 2014. DOI: 10.4204/EPTCS.153.10
15. B. Trancón y Widemann and M. Lepper. Sound and soundness: Practical total functional data-flow programming. In *FARM 2014*, pp. 35–36. ACM, 2014. Demo abstract. DOI: 10.1145/2633638.2633644
16. B. Trancón y Widemann and M. Lepper. The Shepard tone and higher-order multi-rate synchronous data-flow programming in Sig. In *FARM 2015*, pp. 6–14. ACM, 2015. DOI: 10.1145/2808083.2808086
17. M. Weiser. Program slicing. In *ICSE 1981*, pp. 439–449. IEEE, 1981.