Towards Execution of the Synchronous Functional Data-Flow Language SIG

[Draft Paper]

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

Markus Lepper
semantics GmbH

Abstract

SIG is the prototype of a purely declarative programming language and system for the processing of discrete, clocked synchronous, potentially real-time data streams. It aspires to combine good static safety, scalability and platform independence, with semantics that are precise, concise and suitable for domain experts. Its semantical and operational core has been formalized. Here we discuss the general strategy for making SIG programs executable, and describe the current state of a prototype compiler. The compiler is implemented in Java and targets the JVM. By careful cooperation with the JVM JIT compiler, it provides immediate executability in a simple and quickly extensible runtime environment, with code performance suitable for moderate real-time applications such as interactive audio synthesis.

1. Introduction

SIG is the prototype of a purely declarative programming language and system for the processing of discrete, clocked synchronous, potentially real-time data streams. It is designed to support both visual (data-flow diagram) and textual (functional) programming styles, to be scalable to complex tasks, and to be interoperable with a wide variety of execution platforms and legacy code bases.

The potential application fields for SIG are in science, such as modelling and simulation of system dynamics, in engineering, such as sensor data processing and control in embedded systems, as well as in art, such as audio synthesis and computational music.

The strategic vision of the SIG project is to leverage the safety and productivity of modern language technology in a system that can be used effectively, and its actual semantics understood, by domain experts. We believe that this could constitute a significant improvement over the state of the art, which is plagued by twin evils: Application development that uses the established domain-specific tools must deal with their outdated technology and ill-defined semantics; while development that avoids them exposes domain experts as programming laymen to low-level general-purpose programming languages with inadequate expressivity.

The full realization of this vision is of course a long-term goal, and would require substantial effort in order to implement an infrastructure consisting of development tools, runtime environments, algorithmic libraries, bindings for indispensable legacy code, etc. A first major step has been reported on in [7], where the computational framework of SIG (i.e. denotational semantics, core operations, intermediate code representation, and their precise relationships) are discussed in due technical detail.

In the present paper, we report on the next step: a prototype SIG runtime environment that emphasizes integrated tool chains, and immediate and transparent execution of code in various phases of the interpreted–compiled spectrum. This allows us to demonstrate SIG in application areas with interactive systems and moderate real-time requirements, simultaneously showcasing the expressivity and practical feasibility of the language. A running demo package for audio synthesis has recently been published [8].

2. SIG at Work

2.1 Design Considerations

With regard to notation, the data-stream programming world is divided into a visual and a textual camp.

The visual approach, employing data-flow diagrams as the main notation for algorithms, is traditionally favoured by domain experts. Typical programming systems include Simulink for engineering applications, Max/MSP for audio and artistic performance, or the “system dynamics” school of computational modelling of complex systems. Programs are graphs built from *boxes* that specify computations, and *wires* that carry data flow. In spite of the appealing ability to visualize the routing of data flow very intuitively, the diagram approach is known to suffer from poor scalability, frequent confusion of layout and semantics, and lack of support for other essential aspects of algorithms: data types, case distinctions, abstraction and reuse, state and initialization.

These weaknesses are conspicuously absent in functional programming, which features well-understood remedies such as type inference, algebraic data types and pattern matching, anonymous and higher-order functions, and purely declarative semantics. It is therefore no surprise that *functional reactive programming* (FRP) is hailed as an elegant foundation for data-stream programming by the more semantically-minded. Diagrams can be expressed in this framework in terms of *arrows* [2].

The SIG approach aims at neutrality between visual and textual frontend representations, and consequently has been designed around a functional core representation that can represent both naturally; see [7]. In comparison with FRP, SIG takes a characteristically different route: On the one hand, the model of time as discretized by clock ticks at one or several constant rates, is much

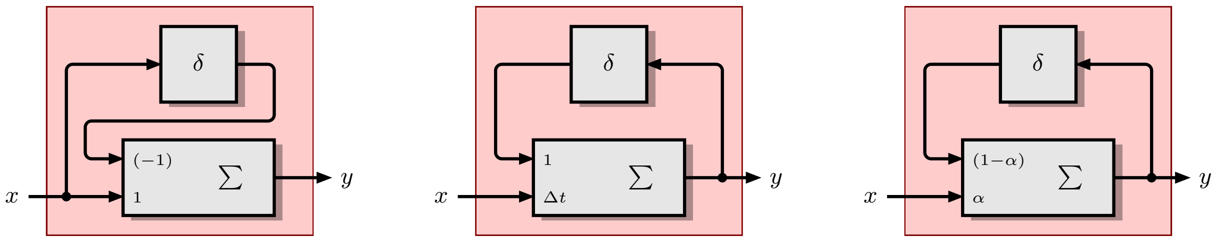

Figure 1. Linear stream programming with delay: *left to right* – backward difference; discretized integral; first-order low-pass filter.

less generic and abstract than in general FRP, which supports also spontaneous events and continuous signals. On the other hand, this restrictiveness is exploited in a computational model that brings denotational semantics and low-level implementation techniques to a very close congruence, and is more orthogonal to other features of functional programming, most notably pattern matching, than current arrow-based FRP frameworks.

2.2 Frontend

True to the tradition of synchronous data-flow programming, SIG programs are represented in a style that abolishes all kinds of explicit sequential control flow, such as blocks, loops or recursion. All computations are specified as if operating on instantaneous data at a single clock tick, and are understood to be implicitly lifted to whole streams by iteration at their respective clock rate, without spontaneous events or termination. All data flow is conceptually instantaneous, unless explicitly delayed. Nontrivial behavior in general (anything other than a function mapped over a stream) arises from delayed interference, and state in particular arises from delayed feedback. Instantaneous feedback (i.e. circular data flow *not* passing through a delay operator) is forbidden. Hence no causal singularities arise; scheduling can be decided modularly and statically, and no fixpoints need be considered. For simplicity, we consider only one primitive delay operator δ, which delays an arbitrary stream for exactly one clock tick (i.e. prepends some specified or default initial value).

A great variety of important building blocks for stream processing algorithms can already be specified in the simplest form of this style; see Figure 1 for a gallery of ubiquitous components built from elementary arithmetics and delay.

Evidently, the diagram approach shines where data has *product* structure and routing is static: a tuple of values is nicely visualized as a bus of wires. By contrast, data with *coproduct* structure, where routing depends on dynamic case distinctions, is handled rather awkwardly. It is hence no surprise that automata (a principal algorithmic manifestation of coproduct-oriented computation) are supported by a *different* diagram language in visual approaches (e.g. Stateflow for Simulink), if at all.

As an archetypal running example, consider the *sample and hold* (S&H) operator, which either forwards its current input x or retains its previous output y, depending on an external trigger t taking the values $\{\mathsf{S}, \mathsf{H}\}$. This functionality can be specified conveniently in a diagram as depicted in Figure 2, using an ad-hoc multiplexer component. Note that we refrain from the "engineering practice" of encoding the range of t numerically, for obvious reasons of clarity and safety. An equivalent specification can be given textually as depicted in Figure 3, using an enumerated type and the SIG box notation. Note that this notation differs from lambda terms by naming both inputs and outputs explicitly and symmetrically.

The multiplexer approach to control flow, while handy for simple situations, has rather poor expressivity and scalability. For instance, consider the evident refactoring of the S&H component

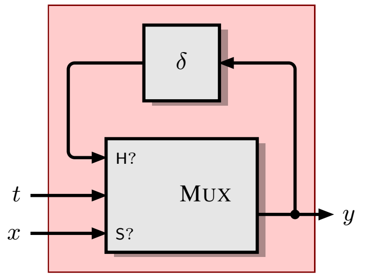

Figure 2. Triggered S&H; diagram with multiplexer

```
type trigger = { S, H }

[
  in x : real , t : trigger
  out y : real
where
  y := case t of {
         S → x
         H → delay(y)
       }
]
```

Figure 3. Triggered S&H; SIG notation (box)

from a functional programmer's viewpoint: Since the input x is irrelevant in the *hold* case, a more economic interface would fuse the two inputs, using a well-known algebraic datatype as depicted in Figure 4. Note that Scala vernacular is used, Haskell enthusiasts may substitute *Maybe*. Whereas this encoding is easily processed with pattern matching clauses, there is no obvious viable generalization of multiplexing to do the job. Apparently the challenging feature is the combination of case distinction and data unpacking, as effected by pattern constructors, as a single atomic operation.

To bring the S&H example even closer to traditional functional programming style, a lambda-style asymmetric function abstraction and named access pattern may be used, as depicted in Figure 5. Note that delayed feedback from the output, a ubiquitous pattern in stream programming, practically prevents the function body expression from being anything than a locally bound variable, hence the gain in conciseness over the box notation is not quite as great as in noncircular cases.

However, the **where** clause gives an impression of the unification of the diagram and expression paradigms that SIG aspires to. Ideally, the programmer should be free to combine the notations orthogonally, each where it shines: Expressions for tree-shaped flow with irrelevant intermediates and coproduct-structured data; diagrams for irregular and circular flow and product-structured data.

```
type option(t) = { some(t), none }

[
  in x : option( real )
  out y : real
where
  y := case x of {
         some(v)  → v
         none     → delay(y)
       }
]
```

Figure 4. Triggerless S&H; SIG notation (box)

```
{
  x : option( real ) → y
  where y := getOrElse(x, delay(y))
}
```

Figure 5. Triggerless S&H; SIG notation (lambda)

The SIG language addresses these issues by program reduction to a core layer with primitive operations that can implement multiplexers and pattern matching equally naturally, and deal with delay in a semantically clean and operationally useful way.

2.3 Core

The key insight behind the semantic framework of SIG is that three essential description formats can be made to coincide [7]:

1. adjacency-based algebraic hypergraph representation of data-flow diagrams (with wires as nodes and boxes as hyperedges, respectively);
2. administrative normal form of functional program expressions, or rather the equivalent static single-assignment (SSA) form;
3. intensional definition of local, elementwise semantics given as a Mealy-style combined I/O-and-transition relation (giving rise to global, stream function semantics by coinduction).

The full details of the theoretical foundation of (3.) and the algorithmic derivation of (2.) from a functional frontend notation can be found in [7]. In the present section, we summarize the key points. The following sections give the main technical contribution of the present paper, by discussing the further use of (2.) in a compiler pipeline.

2.3.1 Delay Elimination

The notation of stream computations in terms of per-element and delay operators, while intuitively convenient, is awkward to reason with directly in a declarative language processing framework. Stream-level behavior is not specified fully by element-level input/output relations, as delay operators break referential transparency.

Hence SIG eliminates delay operators en route to the core layer, by introducing a matching pair of pre- and post-state variables for each occurrence of δ, which then becomes a pair of independent simple equations, forwarding input to post-state and pre-state to output, respectively. Apparently circular data flow is admissible if and only if the circles are eliminated by the splitting of all delay operators.

It is implied that the post-state values of each clock cycle flow to the corresponding pre-state variables of the next cycle. That is, the quaternary relation of input, output, pre- and post-state specifies a stream-transducing Mealy machine.

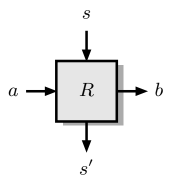

Figure 6. Stateful single-step computation model

```
[
  in x : real , t : { S, H }
  out y : real
  state z : real                 // implies z' : real
where
  y := case t of {
         S  → x
         H  → z
       }
  z' := y
]
```

Figure 7. Triggered S&H; SIG notation (no delay)

The approach can be visualized as depicted in Figure 6. Explicit data flow in the sense of the functional composition of computations, proceeds left to right. Temporal data flow proceeds top to bottom. The stream-level global semantics of a program is given by replicating its element-level relation ω times along the vertical axis, up to initial values for the top end. If the element-level relation is a total function, as dictated for complete SIG component definitions, then the corresponding denotational semantics is captured neatly by coinduction, as elaborated in [7].

The reduction of delay to state can also be notated textually. Figure 7 depicts the result of delay elimination from the program in Figure 3. Note that reduction to the core layer also implies the naming of all intermediate values, as customary for administrative normal or SSA form, although the S&H example does not exhibit this feature.

2.3.2 Control Elimination

Control flow is an awkward feature from a data flow-centric perspective. The SIG approach reduces control flow to data flow for the purpose of maximally parallel core-layer semantics. Backends are free to implement these directly, as in hardware, or to emulate them by reconstructed control flow, as on sequential machines.

The rationale here is that the automatic sequentialization of parallel programs is conceptually much simpler, and effectively achieved with standard compiler technology, than the reverse problem, which remains the elusive holy grail of traditional high-performance computing.

The elimination of control is achieved by creatively abusing the φ operator introduced by SSA, and complementing it with a novel, dual γ operator, to be introduced below. In its original sense, φ multiplexes a number of inputs, understood as alternative values of the same variable produced by different control predecessors.

Of course, there is to be no such thing in SIG; the very purpose of the core layer is to gather all computations in a single basic block. Instead, the SIG-style φ operator multiplexes values from (the right hand sides of) different clauses of a case distinction, depending on the success of pattern matching (of their respective left hand sides).

To this end, all *internal* variables of a component are tacitly augmented to admit an additional value $\perp$. Note that $\perp$ merely signifies that no value is currently available. This is a decidable situation, since program divergence, the usual meaning of $\perp$ in the

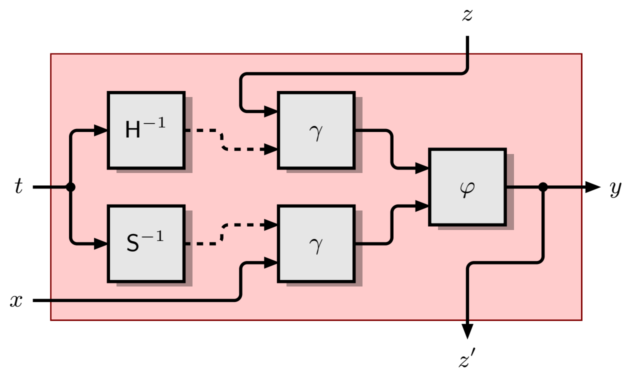

Figure 8. Triggered S&H; diagram (core)

```
[
  in x : real , t : { S, H }
  out y : real
  state z : real                  // implies z' : real
where
  local c, d : control
  local v, w : real
  c := S⁻¹(t)
  v := guard(x, c)
  d := H⁻¹(t)
  w := guard(z, d)
  y := choose(v, w)
  z' := y
]
```

Figure 9. Triggered S&H; SIG notation (core SSA)

semantics of recursive functions, is excluded. Ordinary elementary operations are lifted strictly; if any input is $\perp$, then so are all outputs.

Each partial computation, such as a single clause of a case distinction, can be represented uniformly as a "left-top-total" relation in the sense of Figure 6, where missing cases are mapped to $\perp$. A φ node then simply chooses nondeterministically among its non-$\perp$ inputs, or yields $\perp$ if there is none.

The success of pattern matching is communicated by adding to each pattern constructor an additional "control" output indicating success. The type of these is nominally a singleton $\{\top\}$, augmented to a Boolean control type $\{\top, \perp\}$. If the pattern succeeds, then regular outputs unpack the data constructor argument values, which are non-$\perp$ by strictness, and the control output is $\top$. If the pattern fails, then all outputs are $\perp$. Note that this encoding may appear redundant for data constructors with arguments, but it is *not* for the common case of nullary constructors, where the corresponding pattern is a Boolean test.

The selection of computations is expressed by a guard operator γ. It takes a single data input and arbitrarily many control inputs. The data is forwarded if no control is $\perp$. Otherwise, the output is $\perp$ as well. A clause from a case distinction is then selected by guarding each result of its right hand side with all control values issued by its left hand side.

The elimination of control can be specified formally by tedious but straightforward syntax-directed rewrite rules, see [7]. The application to the S&H example is depicted as a diagram in Figure 8. Data and control wires are indicated by solid and dashed lines, respectively.

A textual representation is depicted in Figure 9. As stated in the beginning of section 2.3, the set of assignments can be read consistently in several ways: as the adjacency list of the hypergraph corresponding the diagram in Figure 8; as a normalized functional program in SSA form consisting of a monolithic basic block; as the intensional definition of an element-level semantic relation by set comprehension in the style of the Z notation.

With respect to the former two, note that the textual single-assignment constraint coincides with the usual diagram constraint that distinct outputs must not collide on a shared wire.

Note that γ and φ nodes act as data-carrying conjunction and disjunction operators, respectively. They can be reduced further to logical expressions in conjunctive normal form. Hence interesting static properties such as definite single assignment of outputs can be checked using off-the-shelf SAT solver technology.

2.4 Backend

While SIG is designed with maximal platform independence in mind, there are a number of general assumptions that constitute a loose execution model.

2.4.1 Composition of Components

A key feature of SIG for scalability and efficient use is full compositionality. The computational box abstraction unifies primitive computations and user-defined subprograms. Thus complex stream-processing systems are constructed and scoped hierarchically. A reference to a defined component can be inlined (i.e. the box replaced by its innards) without affecting program semantics.

This seems like an obvious, almost trivial, property of a functional language, but has decidedly nontrivial consequences in a time-aware setting. Most importantly, the wire abstraction of data flow must not have intrinsic delay, as this would break the scale-free semantics and disallow the optimizations that routinely go with inlining, such as copy propagation. All data flow, except for explicit delay, must be undistinguishable from instantaneous transport.[1] This places strict bounds on the depth of computational networks that can be implemented with given real-time constraints.

2.4.2 Global Control

The realization of a component performs a single step that processes one element of each connected data stream. This involves the updating of pre-state from the preceding post-state, the consumption of inputs, and the production of intermediate values, post-state and outputs, with no particular order of the subtasks specified. During the execution of a step, each component is responsible for having its subcomponents executing a step of their own, respecting data flow constraints.

On a sequential platform, this means that the producer of each stream must execute before its respective consumers. SIG is designed such that a schedule can be devised compositionally and ahead of time. Note that the order of assignment statements depicted in Figure 9, while semantically irrelevant, is a valid sequential schedule of the component; each variable is written before it is read. A sequential implementation is free to choose this or any other valid order, as long as the choice remains transparent to the external observer.

In many cases, ahead of time means at compile time, but various advanced but typical applications require the reconfiguration of computations by parts of the program running at a slower rate. Support for such hierarchically dynamic systems in SIG is a matter for future research.

[1] To use a physical metaphor, the SIG model of spacetime is the Newtonian $c \to \infty$ limit of relativity.

Parts of a SIG program may operate at the same or at different clock rates. The complete program is sliced into its synchronous parts (i.e. each operating at a single rate) and re-sampling connectors. The details are a matter of future research. The runtime environment triggers the execution steps of each root component centrally, with the prescribed rate, in a conceptually infinite loop. Components may not choose to terminate this loop spontaneously.

2.4.3 Inter-Component Communication

Communication between components (i.e. how wires work conceptually) in SIG is characterized by pull-based shared memory. Actual implementations may use arbitrary mechanisms to achieve the specified behavior.

Each component has the exclusive ownership of a distinct writable storage location for each of the output streams it produces. Consumers can access the current value of a stream by reading from this location. All activity is driven by the external clock; neither production nor consumption constitutes an observable event.

On the one hand, the current element of each stream is defined by the value of its location at the clock tick. The producer must be given the opportunity to write an up-to-date value in time. Otherwise, the previous value is tacitly retained. By writing to a location, the previous value is generally made inaccessible. If needed, it must be retained elsewhere, typically using delay.

All outputs of a component change apparently simultaneously. Inconsistent states, such as temporarily arising from implementation by a sequence of write operations, must not be observed. Spontaneous events of the execution environment must be quantized at some clock rate, and reacted on by polling.

On the other hand, each component is oblivious to the consumers of the outputs it produces. Reading the current element of a stream from a location does not notify the producer. Demand for a value does not trigger its computation, nor does absence of demand prevent it. Weird effects such as the infamous *time leaks* of lazy FRP do not arise.

2.4.4 Total Computations

The shared-memory communication model implies that, without additional out-of-band information, is is conceptually impossible *not* to yield a result. In embedded systems, this is often a very practically the case.[2] SIG components are generally implementations of total functions; they must not fail to define their outputs for any valid combination of input and pre-state.

By contrast, arbitrary networks of components have more freedom. They can produce $\perp$ values, and even be nondeterministic. In the disciplined textual frontend language of SIG, the former arises from partial computations such as incomplete case distinctions, and the latter arises from overlapping cases, since SIG has no implicit first- or best-fit disambiguation rules. By liberal use of the core operations γ and φ, a wider variety of similar situations can be created.

Only when a network of components is explicitly designated as the definition of a component by the programmer, a proof obligation for totality and determinism is entailed. Since the question is evidently undecidable in general for all nontrivial collections of primitive operations, a statically checkable approximation is needed. For the disciplined approach (where unsafe subcomputations arise from pattern matching), the requirement that case distinctions be complete and non-overlapping is a natural candidate, and can be checked effectively using standard compiler technology. Possible relaxations, as well as the general case of arbitrarily mixed core operations, are left for future research.

[2] As has been demonstrated drastically by the botched first launch of the Ariane 5 rocket.

3. Compiler

3.1 Architecture and Environment

The current SIG compiler and execution environment is written in Java. The parser is generated by a variant of the ANTLR[3] tool. Syntax trees are mapped to an intermediate representation (IR) as specified in [7], and the various subsequent program transformations towards the SSA form are implemented using a visitor style pattern approach. The IR data model and the visitor machinery are generated from a very concise ($\sim$200 lines) specification by the UMOD tool [3].

Programs in IR can be executed on the fly by an interpreter, or translated to JVM bytecode for better performance. A joint communication API makes the choice of the execution strategy transparent, on a per-component basis. Bytecode is produced in a closed loop and fed directly to the JVM class loader, without the need to call external tools. Alternatively, the bytecode can be stored and compiled to machine code by an external static code generator.

Theoretically, SIG programs can be modularized and compiled separately, although the frontend notation has no module system yet. However, for real-time applications, we expect that satisfactory results require whole-program compilation, in particular since many important analyses (e.g. worst case execution time) work best globally. The non-recursive nature of SIG data flow networks ensures that conceptual boundaries which exist in well-structured source code can be eliminated during compilation by aggressive inlining. Performance-critical application tend to be small enough for whole-program compilation to be feasible.

3.2 Runtime Interface

3.2.1 Type Specialization

Several basic data types of the SIG frontend are mapped directly to their Java/JVM counterparts, such that primitive operations can be used and the dynamic allocation of boxing objects can be avoided. Computations that declare only variables of such types are guaranteed to run without the use of the JVM allocator, and hence without triggering the garbage collector, which greatly enhances real-time responsiveness.

In particular, the Java/JVM types `int` and `double` are supported. The Java frontend type `boolean` is supported as well, which is encoded as the subset $\{0, 1\}$ of `int` on the JVM. Following this example, arbitrary user-defined enumerated types (i.e. algebraic data types with nullary constructors only) are encoded as subsets $\{0, \dots, n-1\}$ of `int`. The extra value $\perp$ is encoded by pairing each variable of primitive type with a `boolean` control variable. Types that have no primitive mapping are encoded as objects.

3.2.2 Data Interfaces

For the sake of abstraction, the interfaces of components admit two different perspectives. The internal perspective is symmetrical with respect to input and output. It identifies variables of both kinds formally by locally scoped names, and operationally by self-owned storage locations. This view has been demonstrated in the examples of the SIG box notation.

By contrast, the external view treats input and output asymmetrically. Each component publishes its outputs passively by implementing an API Source for querying their current values. Conversely, inputs are supplied by reference to another instance of the API Source, which the component may query actively. Variables of either kind are identified by their position in the list of respective parameter declarations, regardless of their internal names. Thus the principle of alpha equivalence carries over from conventional functional programming.

[3] http://www.antlr.org

```
interface Source {
  int      getInt     (int index);
  double   getDouble  (int index);
  boolean  getBoolean (int index);
  Object   getValue   (int index);
}
```

Figure 10. Data API

The API needs to strike a pragmatic balance. On the one hand, static safety and efficiency of data flow demand a high degree of specialization. On the other hand, ease of use and efficiency of caller logic demand a uniform access pattern. In the current implementation, we have chosen a middle road. The uniform interface is depicted in Figure 10. It specializes access according to implementation data types, but not according to parameter position. A critical evaluation of the actual performance and comparison with alternative approaches is a matter for future research.

Note that the API is mainly employed at system boundaries. Globally, instances are supplied by the runtime environment and the compiled program for system inputs and outputs, respectively. Locally, API encapsulation arises at metaprogramming stage boundaries, where one part of the running system configures another, to be run at a faster rate. Within relatively static component networks, the SIG compiler is expected to perform whole program optimization, resulting in the elimination of intermediate interfaces by inlining.

3.2.3 Component Instantiation

Metaprogramming capabilities are essential to the SIG approach, because the greatly amplify the scalability of the program development process. We follow the *staged* metaprogramming paradigm à la MetaML[6], where code fragments can be *quoted* and *spliced*, the meta equivalent of function abstraction and application, respectively, in ordinary higher-order functional programming. The current implementation has prototypic support. The details of notation and semantic constraints of SIG metaprogramming are a matter for future research.

Care must be taken when lifting a first-class notion of computation as data to the scenario of elementwise stream processing. There is no evident canonical explication of, say, a stream of stream functions. The dilemma is rather subtle: On the one hand, application to single elements is not functional application, due to hidden state transitions. On the other hand, application to whole streams, which is perfectly functional, is never expressed directly in the language.

How staged metaprogramming can help to clarify these issues is a matter of ongoing research. The full details are to be discussed in a forthcoming companion paper. For the present, it suffices to state informally that stages break synchronization. From the perspective of earlier stages, later stages are code objects to be configured to run independently, at a faster rate. Conversely, from the perspective of later stages, earlier stages are represented as creation-time snapshots only. This asymmetricity allows to keep violations of referential transparency, incurred by the use of delay/state, under the hood of the implementation.

The technical realization of this scheme in the Java-based runtime environment uses a three-tiered factory model, with one layer of abstraction each above and below the representation of components.

The highest level of abstraction, and the unit of *implementation*, is the Template. It corresponds to the defining expression of a component object (i.e. a quotation in the frontend language), out of context. In higher-order functional terminology, templates can be thought of as *lambda-lifted* local functions. A template can be instantiated with an environment snapshot of the current values of its free (cross-stage) variables to produce a Component, see Figure 11. This is done implicitly by the quotation operator. Different implementation strategies can coexist transparently through different subclasses of Template.

```
interface Template {
  Component newInstance (Source environment);
}
```

Figure 11. Runtime factory; upper level

```
interface Component {
  Session newSession ();
}
```

Figure 12. Runtime factory; middle level

The middle level of abstraction, and the unit of *configuration*, is the Component. Components represent referentially transparent stream functions. In higher-order functional terminology, templates can be thought of as *closure-converted* local functions. In order to make components reentrant in spite of local state, they must be instantiated for each stream-level application to produce a Session, see Figure 12. This is done implicitly at initialization time of the containing computation.

The lowest level of abstraction, and the unit of elementwise *computation*, is the Session. Sessions represent intermediate states of stream computations, and are thus not referentially transparent. They are never exposed to the user, but handled only internally. A session need to be initialized (init), and subsequently invoked (step) once per clock tick to execute a step. This is done implicitly at initialization time of the containing computation, and by the splicing operator, respectively. See Figure 13.

Sessions communicate by the Source API. Each session must be connected to a source from which it can pull the current elements of its inputs streams at each step. Conversely, each session implements the Source interface to provide public access to the current elements of its output streams. The wiring is performed implicitly at initialization time of the containing computation; the actual pulling of outputs is done by the splicing operator.

In principle, sessions can be reused sequentially by reconnection to new inputs and reinitialization, although concurrent reuse is obviously unsafe and must be avoided.

Each step of a session consists of three subtasks that update prestate (tick), inputs (input), and post-state and outputs (action), respectively. Subclasses of Session must override all abstract methods to implement the computation of the represented component, as well as allocate exclusive storage for all local variables. The current implementation mandates that a copy of the pulled values be stored during the input phase. Thus, all variables in the scope of the component body can have the same storage and access pattern, and there is no need for distinct "operand modes" of primitive operations.

The API has been designed consciously such that no advanced features of Java are used, hence it could be mapped with little effort and no significant overhead to more low-level languages such as C. Thus, by the implementation of a C code generator, SIG components could be made usable as libraries in a very wide variety of systems.

```
abstract class Session implements Source {
  private Source inSource;
  public void setInSource (Source inSource) {
    this.inSource = inSource;
  }

  public abstract void init ();

  public void step () {
    tick ();
    input(inSource);
    action () ;
  }

  protected abstract void tick    ();
  protected abstract void input   (Source source);
  protected abstract void action  ();
}
```

Figure 13. Runtime factory; lower level

3.3 Code Generation

3.3.1 State Transition

From the perspective of the SIG core layer, each delay operator gives rise to a pair of distinct variables x and x' for pre- and post-state, respectively. The code for a single step of a component relies on the calling environment to update its pre-state, namely with initial values on the first call, and with the previous value of the corresponding post-state on each subsequent call. How this state transition is actually effected is up to the particular implementation. There are several reasonable tactics with different usage profiles:

Transport The pair of conceptual variables can be taken literally, and an actual move operation can be used to copy values from each post-state variable to its pre-state counterpart. This is a semantically safe fallback tactic that works in all cases, but not particularly efficient. It is used by the current compiler implementation by default.

Double Buffering The behavior of the step code can be made to alternate between two variants, either by a global Boolean indirection switch, or by flipping between two clones of the code where the respective roles of pre- and post-state are mirrored. This tactic is likely more efficient than literal transport if there are many state variables. It is supported by the current compiler implementation as a configurable alternative to the default.

Overlay During code generation for a sequential machine, the SSA variables of the core representation are likely allocated to pseudo-registers anyway, such that in general values with non-overlapping life times can share a storage location. Additional constraints can be placed on the instruction schedule, such that all operations reading a pre-state variable must occur before the operation writing the corresponding post-state variable. Then pre- and post-state are non-overlapping, and may share a storage location. This tactic can save space as well as time, but does not work in all cases; see Figure 14 for a counterexample. It is used by the current compiler implementation heuristically on an all-or-nothing basis; selective use is planned for a future revision.

Indirect Buffering Multi-step delay of data must be expressed as a chain of single-step delay operations. No matter which of the preceding tactics is used, this yields a naive FIFO buffer implementation in terms of state variables, where values are actually transported from the input to the output end, see Figure 15. Except for near-trivial cases, an indirectly addressed (ring buffer) implementation is preferrable, where the current position of the input and output ends move, rather than the stored data. This tactic needs to be applied selectively for suitably long delay chains in order to pay off. Support is planned for a future revision of the compiler.

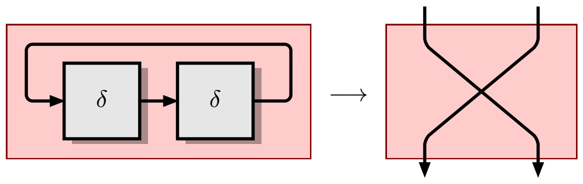

Figure 14. Delay reducing to non-overlayable state variables

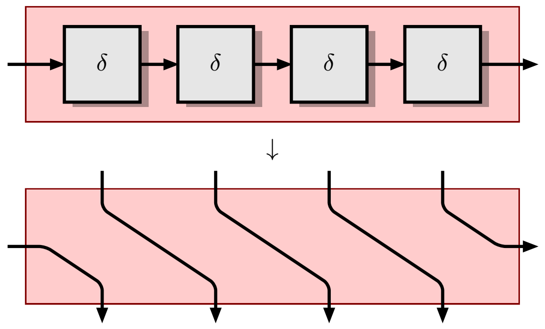

Figure 15. Delay chain reducing to FIFO

3.3.2 Parallel and Sequential Evaluation

The semantics and core operations of SIG have designed carefully to allow for maximal parallelism, constrained only by explicit data flow. The encoding of control flow into data flow that embodies this principle, and is achieved by means of γ and φ operations as described above, seems unnatural from the perspective of execution on a conventional sequential machine: rather than choosing proactively between alternative branches, all branches are evaluated independently, and unneeded results are only discarded after the fact. Compare this behavior to the eager operators `&` and `|` in the C language family, as opposed to the short-circuiting operators `&&` and `||`, respectively. Several arguments need to be considered in favor of either operational approach:

In a side-effect free language, the two variants are behaviorally indistinguishable. Implementations may choose either on the grounds of convenience and efficiency. On a simple sequential execution platform, avoiding unneeded computations by conditional *branches* is virtually always a win. On modern CPUs with deep pipelines, branchless solutions that overlap alternatives and select results by conditional *moves* may be preferrable, as long as alternatives are few in number and not disproportionately expensive. Opportunistic choices need to be made, based on accurate cost models for the specific processor architecture, for good performance. By contrast, on non-sequential platforms such as field-programmable gate arrays (FPGAs), a literally parallel layout of alternatives followed by multiplexers is the canonical solution.

The current implementation of the SIG compiler takes the parallel semantics of control at face value, and translates γ and φ operators to code as they appear. Clearly, this solution scales badly on its target platform, the strongly sequential JVM. Fortunately, the sequentialization of parallel programs is turning out to be a much more tractable problem than its converse. A compiler pass that identifies conditionally needed code in the SSA form and substitutes conditional branches for γ and φ nodes is being developed.[4]

[4] Andreas Loth. Master's thesis.

```
abstract class Action {
  public abstract void run (State state);
  // ...
}
```

Figure 16. Threaded code substep

```
class State {
  public Action pc ;

  public final Object[]   registers   ;
  public final double[]   registers_double  ;
  public final int []     registers_int   ;
  public final boolean[]  registers_bool  ;

  public final boolean[]  registers_control   ;
}
```

Figure 17. Interpreter state

3.3.3 Interpreter

The interpreter variant of the current SIG execution environment operates almost directly on the SSA core form. Operations are scheduled statically in some valid sequential order, variables are allocated to numbered reusable "registers", and frequently occuring generic operations are specialized for their operand cound (if variadic) and/or type (if polymorphic), respectively. Otherwise, there is a one-to-one relationship between SSA statements and substeps of the actual execution.

The substeps are reified as individual Action objects, see Figure 16, organized in an object-oriented form of the traditional *threaded code* approach. The allocated virtual registers are realized as a family of equally-shaped arrays of the various supported primitive types, bundled together with a program counter (i.e. reference to the next substep) in a State object; see Figure 17. The interpreter invokes each substep in turn (run), allowing it to modify the current state. How the program counter is updated has been omitted for simplicity. In the current implementation, a linear list of substeps according to the predetermined schedule is traversed. However, the mechanism generalizes to nontrivial control flow with branching instructions, which shall be supported in a future revision.

The threaded code implementation has been designed to maximize the use of primitive data and array features of the JVM, as opposed to "clean" high-level object-oriented APIs. Consequently, actual operations coded as subclasses of Action invoke few JVM instructions with little execution overhead each, thus encouraging the JVM JIT compiler to compensate the interpretative overhead by aggressive inlining and specialization.

The interpreter, instantiated with the preprocessed code and register layout of a component, is encapsulated behind the Template interface. It can be mixed transparently with other means of implementation, as long as they use the Source API for communication.

The threaded code approach fulfills the requirement for extensible instruction sets nicely. All that is needed to add a new instruction is a new subclass of Action that mutates a State object accordingly, and a corresponding rule in the instruction selection procedure of the interpreter. The Action abstraction also allows for easy unit testing, tracing and profiling of instruction set extension candidates.

3.3.4 Compiler

The threaded code interpreter, while reasonably fast and very flexible, contains two indirections that cause runtime overhead on *every* instruction: dynamic array-based access to local variables, and virtual method invocation of Action.run.

```
public class ... extends Session {
  double in0;                     // x
  int in1;                        // t
  double out0;                    // y
  double pre0;                    // z
  double post0;                   // z'

  // Session method implementations
}
```

Figure 18. Triggered S&H; compiled class

```
abstract class Action {
  // ...
  public void compile(CompilationContext ctx);
}
```

Figure 19. Threaded code substep

We have added an "afterburner" code generation phase that compiles threaded code objects to JVM bytecode. Dedicated subclasses of Session, and their factory progenitors Template and Component, are created for each compiled SIG component. Local variables are mapped to individual member fields of the appropriate primitive type; see Figure 18 for the S&H example. Instructions are compiled to JVM bytecode fragments, which are then glued together to implement Session.action; see Figure 20 in comparison to Figure 9. The resulting code can be loaded directly into the host JVM by a ClassLoader, or stored as class files for external use.

Compilation is implemented in a distributed fashion by Action subclasses. Namely, a method compile receives a CompilationContext object that can resolve variables to JVM constant pool entries, and act as a sink for bytecode instructions. This design retains as much extensibility and traceability of the instruction set as possible, even if fragmented bytecode generation is somewhat harder to test and debug than threaded code. The downside is that, because instruction selection is performed in isolation, the resulting bytecode contains a number of redundancies, easily seen in Figure 20.

Theoretically, an extra optimization pass on the JVM bytecode format could be used for cleanup. But we have found that JVM JIT compilers do that job well already. For the S&H example, the machine code produced by Oracle's Hotspot JVM 1.8.0_20, on a test machine specified in the following subsection, is depicted in Figure 21.

The redundancy that remains in the depicted code, namely that patterns are matched twice, stems from the incongruency of control flow which is parallel in SIG and sequential on the JVM. A transformation-based systematic solution notwithstanding, we have found that existing compilers are quite capable of eliminating the redundancy in simple cases. In particular, the machine code produced by GCJ 4.8.2 with the -O3 option, invoked with the same bytecode on the same target machine, is depicted in Figure 22. Comparison of Figures 21 and 22 illustrates the typical tradeoff between just-in-time and ahead-of-time compilation: more aggressive use of processor-specific capabilities (here, SSE2 extensions) for the former, and more thorough (here, optimal) application of expensive optimizations (here, sparse conditional constant propagation) for the latter.

3.4 Experimental Evaluation

We have tested the performance of both interpreted and compiled code with a simple but nontrivial sound synthesis application. It

```
protected void action();
  Code:
     0: aload_0
     1: getfield    #37                // t
     4: iconst_1
     5: isub                           // S?
     6: ifne        14
     9: iconst_1
    10: istore_1
    11: goto        16
    14: iconst_0
    15: istore_1

    16: iload_1
    17: ifne        28
    20: dconst_0
    21: dstore_2
    22: iconst_0
    23: istore      4
    25: goto        36
    28: aload_0
    29: getfield    #31                // x
    32: dstore_2
    33: iconst_1
    34: istore      4

    36: aload_0
    37: getfield    #37                // t
    40: iconst_0
    41: isub                           // H?
    42: ifne        50
    45: iconst_1
    46: istore_1
    47: goto        52
    50: iconst_0
    51: istore_1

    52: iload_1
    53: ifne        65
    56: dconst_0
    57: dstore      5
    59: iconst_0
    60: istore      7
    62: goto        74
    65: aload_0
    66: getfield    #47                // z
    69: dstore      5
    71: iconst_1
    72: istore      7

    74: aload_0
    75: iload       4
    77: ifeq        84
    80: dload_2
    81: goto        102
    84: iload       7
    86: ifeq        94
    89: dload       5
    91: goto        102
    94: // abort (t out of range)
   102: putfield    #39                // y

   105: aload_0
   106: aload_0
   107: getfield    #39                // y
   110: putfield    #44                // z'
   113: return
```

Figure 20. Triggered S&H; bytecode

```
action:
        mov     0x38(%rsi), %r11d # t
        mov     %r11d, %r10d
        dec     %r10d
        xorpd   %xmm0, %xmm0, %xmm0
        test    %r10d, %r10d      # S?
        je      .Le
        xorpd   %xmm1, %xmm1, %xmm1
.La:
        test    %r11d, %r11d      # H?
        jne     .Lb
        movsd   0x28(%rsi), %xmm0 # z
.Lb:
        test    %r10d, %r10d      # S?
        je      .Ld
        test    %r11d, %r11d      # H?
        jne     .Lf
.Lc:
        movsd   %xmm0, 0x20(%rsi) # y
        movsd   %xmm0, 0x30(%rsi) # z'
        ret
.Ld:
        movapd  %xmm1, %xmm0
        jmp     .Lc
.Le:
        movsd   0x18(%rsi), %xmm1 # x
        jmp     .La
.Lf:
        # abort (t out of range)
```

Figure 21. Triggered S&H; machine code (JRE)

```
action:
        movl    48(%rdi), %eax    # t
        testl   %eax, %eax        # H?
        je      .L17
        cmpl    $1, %eax          # S?
        jne     .L26
        movsd   40(%rdi), %xmm0   # x
        movsd   %xmm0, 56(%rdi)   # y
        movsd   %xmm0, 72(%rdi)   # z'
        ret
.L17:
        movsd   64(%rdi), %xmm0   # z
        movsd   %xmm0, 56(%rdi)   # y
        movsd   %xmm0, 72(%rdi)   # z'
        ret
.L26:
        # abort (t out of range)
```

Figure 22. Triggered S&H; machine code (GCJ)

implements a digital organ with a range of four chromatic octaves. Each of the 49 notes consists of two SIG components, namely a sine wave generator and an ADSR envelope generator, running at the audio rate of 44.1 kHz and the 64 times slower control rate, respectively. The precise algorithms are specified in [8]. They translate to 4 and 54 core operations, respectively.

A hand-coded environment runs all 49 notes in quasi-parallel for full polyphony, and mixes them together according to input from a MIDI keyboard, for interactive real-time CD quality output. The resulting audio stream is fed to the push-based Java audio system. Hence the audio and control rate clocks operate in pseudo-real time: the control loop runs at full speed when there is sufficient space in the audio output buffer, and blocks when the buffer is full. By limiting the buffer size, latency is bounded to 10–100 ms.

The actual time spent in computation (i.e. component execution and mixing) is recorded with the precision and accuracy of Java `System.nanoTime()`. Optimizations that turn off silent voices have been deactivated for the sake of regular load and stable measurements. On our test system, with a Core i5-3317U CPU at 1.7 GHz, Ubuntu 14.04 OS, and Oracle JDE 1.8.0_20, we have observed an effective rate (number of samples produced divided by time spent computing) of 229±3 kHz for interpreted code, and 2740±60 kHz for compiled code, respectively.[5] These figures translate to an average effort of about 152 and 13 CPU cycles per voice-sample, or to a load of 19.6 % and 1.6 %, respectively. The speedup by compilation is a factor of 12. All experiments use only a single CPU core for SIG computations, although JVM system threads may run concurrently on other cores.

In summary, the interpreted version, on stock hardware and without JVM tweaking, performs fast enough for a real-time demonstration by a comfortable margin. The compiled version has enough computational reserves that it can be expected to scale up to audio synthesis tools of artistically acceptable quality.

4. Conclusion

The SIG language is highly domain-specific, and hence poses specific problems for effective and efficient execution. On the one hand, the purely and totally functional approach, and the rigid control flow enable or simplify a great number of analyses and optimizations. On the other hand, the prototype nature of the current implementation and applications, and the fact that type system and instruction sets are far from fixed, calls for a compiler design that is more a laboratory environment than a closed tool.

As a notable practical lesson from the construction of the SIG compiler, we have corroborated the hypothesis that bytecode platforms are suitable backends for rapid prototyping. Many errors in the code generator have been detected statically by standard JVM bytecode verification tools. In other cases that fail at runtime, debugging is fairly convenient, even without a working generator for symbol tables or source location metadata.

The Java platform has extensive support for real-world interaction, such as GUIs, sampled audio output and MIDI audio input, in terms of on-board libraries that work out-of-the box and with decent efficiency/safety tradeoffs. Using these, tangible demonstrations of SIG programs in a live loop, as in [8], can be constructed with very moderate effort.

The JVM JIT compiler allows to explore the interpreter–compiler continuum in search for a sweet spot for the prototype implementation of a novel language rather freely, by keeping the performance penalties for higher levels of backend abstraction within reasonable limits.

4.1 Related Work

- FRP [2, 4, 9]
- Hume [1]
- Faust [5]
- Trace-based compilation techniques

4.2 Future Work

- Branch-based implementation of φ nodes
- C backend
- FPGA backend
- Worst-case execution time analysis

[5] Reported errors are mean absolute deviations.

References

[1] Kevin Hammond and Greg Michaelson. The design of Hume: A high-level language for the real-time embedded systems domain. In *Domain-Specific Program Generation*, volume 3016 of *Lecture Notes in Computer Science*, pages 127–142. Springer-Verlag, 2003.

[2] Paul Hudak, Antony Courtney, Henrik Nilsson, and John Peterson. Arrows, robots, and functional reactive programming. In *Advanced Functional Programming (AFP 2002)*, volume 2638 of *Lecture Notes in Computer Science*, pages 159–187. Springer-Verlag, 2003.

[3] Markus Lepper and Baltasar Trancn y Widemann. Optimization of visitor performance by reflection-based analysis. In J. Cabot and E. Visser, editors, *Proceedings 4th International Conference on Theory and Practice of Model Transformations (ICMT 2011)*, volume 6707 of *Lecture Notes in Computer Science*, pages 15–30. Springer-Verlag, 2011.

[4] Henrik Nilsson, Antony Courtney, and John Peterson. Functional reactive programming, continued. In *Haskell Workshop*, pages 51–64. ACM, 2002.

[5] Yann Orlarey, Dominique Fober, and Stephane Letz. Syntactical and semantical aspects of Faust. *Soft Comput.*, 8(9):623–632, 2004.

[6] Walid Taha and Tim Sheard. Metaml and multi-stage programming with explicit annotations. *Theor. Comput. Sci.*, 248(1-2):211–242, 2000.

[7] Baltasar Trancón y Widemann and Markus Lepper. Foundations of total functional data-flow programming. In *Mathematically Structured Functional Programming (MSFP 2014)*, volume 154 of *Electronic Proceedings in Theoretical Computer Science*, pages 143–167, 2014.

[8] Baltasar Trancón y Widemann and Markus Lepper. Sound and soundness – practical total functional data-flow programming. In *2nd International Workshop on Functional Art, Music, Modeling and Design (FARM 2014)*. ACM Digital Library, 2014.

[9] Zhanyong Wan and Paul Hudak. Functional reactive programming from first principles. *SIGPLAN Not.*, 35(5):242–252, 2000.