On-Line Synchronous Total Purely Functional Data-Flow Programming on the Java Virtual Machine with Sig

Baltasar Trancón y Widemann
Ilmenau University of Technology
Ehrenbergstraße 29, 98693 Ilmenau
Germany
baltasar.trancon@tu-ilmenau.de

Markus Lepper
semantics GmbH
Berlin
Germany

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 just-in-time 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.

CCS Concepts

•**Software and its engineering → Runtime environments; Domain specific languages;** *Interpreters; Just-in-time compilers;*

Keywords

Functional language; Synchronous data flow

1. INTRODUCTION

Sig is a prototypic purely declarative programming language and system for the processing of discrete, clocked synchronous, potentially real-time data streams. It is parallel in the sense that ordered and conditional execution of operations are abstracted from in the semantics, which can be understood in terms of elementary mathematics (relations, automata) and a hardware circuit metaphor.

Sig 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 modeling and simulation of symbolic system dynamics, in engineering, such as sensor data processing and control in embedded systems, and last but not least in art, such as audio synthesis and computational music.

PPPJ '15, September 08–11, 2015, Melbourne, FL, USA
© 2015 Copyright held by the owner/author(s). Publication rights licensed to ACM. ISBN 978-1-4503-3712-0/15/09…$15.00
DOI: http://dx.doi.org/10.1145/2807426.2807430

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 in our fields of application. We believe that this could constitute a significant improvement over the state of the art, which is plagued by twin evils: Application development *using* the established domain-specific tools must deal with their outdated language technology and ill-defined semantics; while development *avoiding* them bothers domain experts as programming laymen with low-level general-purpose programming languages, and their inadequate safety and expressivity.

A first major step towards our ambitious long-term goals has been reported on in [26], where the computational framework of Sig (i.e. denotational semantics, core operations, intermediate code representation, and their precise relationships) is discussed in due formal detail.

In the present paper, we report on the next step: a prototype Sig execution environment that emphasizes integrated tool chains, and immediate and transparent execution of code in various regimes 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 basic audio synthesis has recently been presented [27].

Our implementation is also a case in point of the wide-spectrum applicability of the JVM platform. The Sig language incarnates a programming paradigm extremely different from the one underlying the Java language: Objects with imperative behavior and sequential and/or concurrent control flow on the one hand; infinite data streams and purely declarative, periodically recurring, data flow-driven computations on the other. And yet, we have found the JVM quite hospitable in terms of runtime environment interface design and implementation, and efficient execution of data flow-oriented programs.

1.1 Outline and Contributions

The following section summarizes the design of Sig, its frontend and core representation from [26], as far as needed for the present discussion. The remainder are novel contributions of this paper: Section 3 completes the picture with an operational model of Sig backends. Sections 4 and 5 present the Sig compiler architecture, and its code generation strategy and basic runtime API, respectively. Section 6 gives some experimental results from JVM-based code generation and execution. Section 7 ties up the loose ends.

Evidently, the current Sig system as presented in this paper falls short of our long-term vision in many ways, which are clearly pointed out at the corresponding points in the discussion. Some of these are standard compiler construction chores, while others are old or new challenging problems of theory or design. Nevertheless, we comprehensively document here a substantial amount of practical design and implementation problems solved. The present state of the tools allows us to experiment conveniently with reasonably efficient executable Sig programs, to build demonstrations and gather feedback for the evaluation of language design issues, and to reflect on the JVM as an intermediate language technology.

2. SIG AT WORK

2.1 Design Considerations

With regard to notation, the domain-specific data-stream programming world is presently 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[1] for engineering applications, Max/MSP[2] for audio and artistic performance, or the "system dynamics" school of computational modeling of complex systems[3]. 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 [8], 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 [23].

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 [13], and purely declarative semantics [2]. It is therefore no surprise that *functional reactive programming* (FRP) [31] has been 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* [12]. However, the axiomatic presentation makes it difficult to relate the theory to the practice of domain experts on one hand, and to the technological legacy of established execution models and code bases on the other.

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 [26].

2.2 Frontend

True to the tradition of declarative 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 infinite streams by iteration at their respective clock rate, without spontaneous "interrupt" events or termination. All data flow is conceptually instantaneous, unless explicitly delayed. Nontrivial behavior in general (anything other than a memoryless 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[4], 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 element).

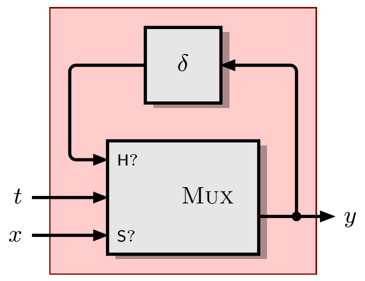

Figure 2: Triggered S&H; diagram with multiplexer

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. The operator $\left(\begin{smallmatrix}*\\*\end{smallmatrix}\sum\right)$ is a linear combinator with given weights.

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 "primordial 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 expressions 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 scal-

[1] http://www.mathworks.com/products/simulink/
[2] https://cycling74.com/products/max/
[3] http://www.systemdynamics.org/

[4] For a counterexample arising from instantaneous feedback see [5].

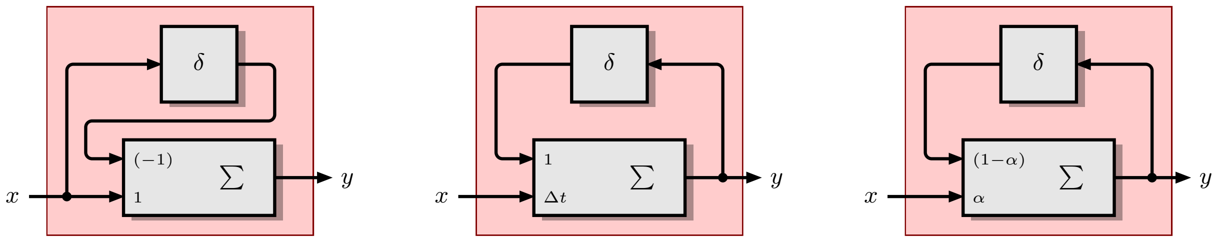

Figure 1: Linear stream programming with delay: ***left to right*** **– backward difference; discretized integral; first-order low-pass filter.**

```
type trigger = { S, H }

sah = [
  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)

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

sah2a = [
  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)

ability. For instance, consider the evident refactoring of the S&H component 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 (de)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 forces the function body expression to be a locally

```
// getOrElse : (Option(T), T) → T

sah2b = {
  (x : option( real )) → y
  where y := getOrElse(x, delay(y))
}
```

Figure 5: Triggerless S&H; Sig notation (lambda)

bound variable, hence the gain in conciseness over the box notation is not quite as great as in non-circular 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.

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 [26]:

1. algebraic hypergraph representation of data-flow diagrams (with wires as nodes and boxes as hyperedges, respectively);
2. static single-assignment (SSA) form of functional program expressions;
3. intensional definition of local, elementwise semantics given as a Mealy semiautomaton, i.e., combined I/O-and-transition relation (giving rise to global, stream function semantics by coinduction, along the lines of [6]).

The full details of the algorithmic derivation of (2.) and the theoretical foundation of (3.) from a functional frontend notation can be found in [26]. In the remainder of the present section, we summarize the key points. The following sections then give the main technical contributions of the present paper, by discussing the further use of (2.) in a compiler pipeline.

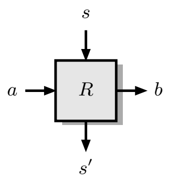

Figure 6: Stateful single-step computation model

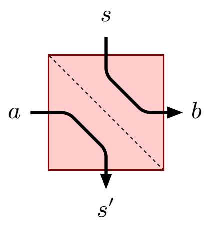

Figure 7: Delay operator functionality

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

Figure 8: S&H; Sig notation (delay eliminated)

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 framework. Stream-level behavior is not specified fully by element-level input/output relations, as delay operators appear to break referential transparency.

Sig eliminates delay operators en route to the core layer, by introducing a matching pair z, z' of pre- and post-state variables for each occurrence of δ, which then becomes a pair of *independent* simple assignments, forwarding input to post-state and pre-state to output, respectively. Thus the apparent data-flow connection between input and output of the delay node, which does not imply actual flow at the single element level, is broken. 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; possibly a nondeterministic one, see section 2.3.2 below.

The approach can be visualized as depicted in Figures 6 and 7. Explicit data flow, in the sense of the functional composition of operations, proceeds left to right. Temporal data flow, between clock ticks, proceeds top to bottom. The stream-level global semantics of a program is given by replicating its element-level relation infinitely often along the vertical axis, with the post-state of each step equal to the pre-state of its successor, and specifying initial state values.

The reduction of delay to state can also be notated textually. Figure 8 depicts the result of delay elimination from the program in Figure 3. Note that reduction to the core layer also entails the explicit 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 non-sequential 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; see section 5.2 below. The rationale here is that the automatic sequentialization of parallel programs[5] 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 [1].

The semantic equivalence of control-flow structure (branches either taken or not) and data-flow structure (selection from branch results) is of course only possible by the absence of both side effects and nontermination in the language.

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 specified 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 are tacitly augmented to admit an additional value $\bot$. Note that $\bot$ merely signifies that no value is currently available. This is a decidable situation, since program nontermination, the usual meaning of $\bot$ in the semantics of recursive functions, is excluded. Ordinary elementary operations are *strict*; if any input is $\bot$, then so are all outputs.

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

$$x_1, \dots, x_n \overset{\varphi}{\longmapsto} x_i \iff x_i \neq \bot \vee \{x_1, \dots, x_n\} = \{\bot\}$$

The success of pattern matching is communicated by adding to each data destructor (implementation of pattern constructor) an additional "control" output indicating success. Its type is nominally a singleton $\{\top\}$, augmented to a Boolean control type $\{\top, \bot\}$. If the pattern succeeds, then regular outputs unpack the data constructor fields, which

[5]Note the distinction of parallel *programs* from parallel *executions*.

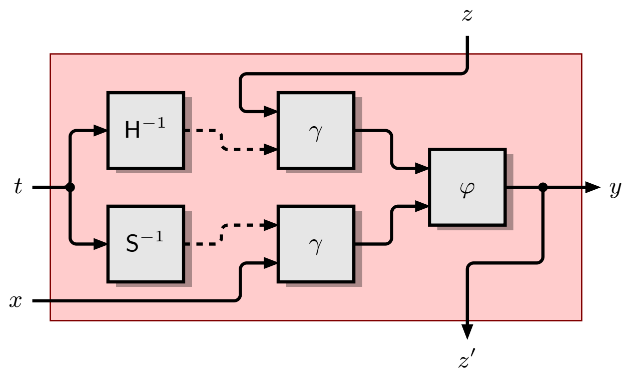

Figure 9: Triggered S&H; diagram (core)

are certainly non-$\bot$ by strictness, and the control output is $\top$. If the pattern fails, then all outputs, including control, are $\bot$. Note that this encoding seems 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 partial computations is expressed by a guard operator γ. It takes a single data input and arbitrarily many control inputs. The data is forwarded if and only if no control is $\bot$:

$$x, c_1, \dots, c_n \xmapsto{\gamma} \begin{cases} x & \{c_1, \dots, c_n\} = \{\top\} \\ \bot & \text{otherwise} \end{cases}$$

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 [26]. The application to the S&H example is depicted in Figure 9. Data and control wires are indicated by solid and dashed lines, respectively.

A textual representation is depicted in Figure 10. 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 9; 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 [24]). With respect to the former two, note that the textual single-assignment constraint coincides with the usual diagram constraint that distinct box outputs must not collide on a shared wire.

Note that γ/φ nodes act as data-carrying con-/disjunction operators, respectively. They can be reduced further to logical formulae in conjunctive normal form. Hence several static properties, such as definite single (non-$\bot$) assignment of outputs, can be checked using off-the-shelf SAT solver technology, as noted in [26].

3. BACKEND

While SIG is designed with maximal platform independence in mind, there are a number of general assumptions that constitute a loose execution model. Its implementation on the JVM is straightforward; see the next section.

```
sah = [
  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 10: S&H; Sig notation (core SSA)

3.1 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.[6] This places hard global bounds on the depth of computational networks that can be implemented with given real-time constraints.

A minor downside is a blanket ban on instantaneous feedback from outputs of a complex subcomponent to its own inputs. Such feedback could possibly be perfectly safe if the corresponding internal data path that completes the loop is delayed, but that fact is not readily reflected in the interface specification (type) of the component, and must hence be disregarded at least in modular compilation, for reasons of proper encapsulation. As a consequence, inlining could conceivably turn an illegal program into a legal one. Conversely and more unfortunately, a subexpression with external feedback cannot be factored out straightforwardly.

3.2 Global Control

The realization of a component performs a single step that processes one element of each connected data stream. This involves the loading of pre-state from the preceding post- (or initial) 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 causality constraints.

On a sequential platform, this means that the producer of each stream must execute before its respective consumers.

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

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 10, while semantically irrelevant, is a causally 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' in the last paragraph means at compile time, but various advanced (yet typical) applications require the reconfiguration of computation subnetworks by parts of the program running at a slower rate.

Parts of a Sig program may operate at the same or at different clock rates [28]. The complete program is to be sliced into its synchronous parts (i.e. each operating at a single rate) and re-sampling connectors. 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. The current state of the compiler creates passive component code only. Hence the main loops that drive components at their respective (interleaved) clock rates have to be programmed manually; see for instance Figure 15 below. The part of the tool chain that automatizes inference of clock rates, slicing of multi-rate programs and generation of driver loops is a matter of future research and development.

3.3 Inter-Component Communication

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

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

On the one hand, the current element of each stream is defined by the value at the corresponding 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 may be tacitly retained; failure to meet real-time requirements need not be detected at runtime. By writing to a location, the previous value is generally made inaccessible. If needed, a delayed copy must be retained.

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 its outputs. 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 lack of demand prevent it.[7]

3.4 Total Computations

The shared-memory communication model implies that, without additional out-of-band information, it is conceptually impossible *not* to yield a result. In embedded systems, this is often very practically the case.[8] 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 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 incurred. 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 analysis technology. This is implemented in the current state of the compiler. Possible relaxations, as well as the general case of arbitrarily mixed core operations, are left for future research.

4. COMPILER ARCHITECTURE

The current Sig compiler and execution environment are written in Java. The parser is generated by the ANTLR[9] tool. Syntax trees are mapped to an intermediate representation (IR) as specified in [26], and the various subsequent program transformations towards the SSA form are implemented using a *visitor* pattern approach. The IR data model and the visitor machinery [16] are generated from a very concise (∼200 lines) specification by the umod[10] tool.

Programs in IR can be interpreted on the fly, or translated to JVM bytecode for better performance. A common component API makes the choice of the execution strategy transparent (including manual implementations in any JVM language of choice), on a per-component basis. Bytecode is produced in a closed loop and fed directly to the JVM class loader, without a need for external storage or tools. Alternatively, the bytecode can be stored for persistent deployment or ahead-of-time compilation to machine code.

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 abstraction barriers which exist in well-encapsulated source code can be eliminated during compilation by aggressive inlining. We expect performance-critical applications to be small (and our compiler efficient) enough for making whole-program compilation feasible.

[7]Weird effects such as the infamous *time leaks* of lazy FRP do not arise.

[8]As has been demonstrated drastically by the botched first launch of the Ariane 5 rocket [9].

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

[10]http://www.bandm.eu/metatools/docs/usage/umod.html

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

Figure 11: Runtime Data API

4.1 Runtime Interface

4.1.1 *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 involve only such types are guaranteed to run without heap allocation, 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 $\bot$ is encoded by pairing each variable of primitive type with a `boolean` control variable. Types that have no primitive mapping, such as complex algebraic data and function types, are encoded as objects.

4.1.2 *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.

The API needs to strike a pragmatic balance. On the one hand, static type 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 11. It specializes access according to implementation data types, but not according to position. Behavior is undefined if the selected position index is out of bound, or the stream not of the expected type. 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 (or module) boundaries. Globally, instances are supplied by the runtime environment and the compiled program for system inputs and outputs, respectively. Locally, API encapsulation arises at higer-order/meta-programming 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.

4.1.3 *Component Instantiation*

So far, we have presented first-order features of Sig. Obviously, the expressive power of a functional language is greatly enhanced by higher-order capabilities. However, the time-dependent variables of the data-flow paradigm come with some pitfalls with regard to referential transparency, as has been noted as a motivation for structured FRP [31]. For illustration, consider the following higher-order program of 'curried' multiplication:

$$\{\ \mathsf{x} : \mathsf{real} \rightarrow \{\ \mathsf{y} : \mathsf{real} \rightarrow \mathsf{x} * \mathsf{y}\ \}\ \}$$

In a data-flow setting, this describes a component that transforms a stream of numbers into a stream of multipliers. But since the resulting components need not *run* at the same clock rate they are *produced*, the free variable x in the inner component body cannot denote the same thing (stream) as in the outer body!

We resolve this paradox by tying nested functions to staged metaprogramming à la MetaML [25], where code fragments can be *quoted* and *spliced*, the meta equivalent of function abstraction and application. Hence Sig forces the inner component to be quoted:

$$\{\ \mathsf{x} : \mathsf{real} \rightarrow \#\{\ \mathsf{y} : \mathsf{real} \rightarrow \mathsf{x} * \mathsf{y}\ \}\ \}$$

The lexically scoped free variables of quotations, called *cross-stage persistent* in MetaML parlance, are a hallmark of staging, as opposed to macro programming, where they would be taken literally. In Sig, cross-stage variables are handled differently from other variables: they denote time-invariant constants, namely snapshots of stream elements taken at the clock tick of quoting. Confer the lexical scoping of final variables in nested Java classes.

Thus quoted components can be applied as stream functions, completely desynchronized from the context of their creation. The typical nontrivial usage pattern is that slow parts of a multi-rate system instantiate components for the faster parts for dynamic configuration. The usual well-formedness constraints on stages ensure that no weird causal loops arise. How our approach relates to theories of point-free higher-order stream programming such as the comonad distributive laws of [30], is an interesting open question.

The technical realization of the staging scheme in the Java-based runtime environment uses a three-tiered factory model, with one layer of abstraction each above and below the actual 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 persistent) variables to produce a Component, see Figure 12. This is done implicitly by the quotation operator. Different implementation strategies can coexist transparently through different subclasses of Template.

The middle level of abstraction, and the unit of *configuration*, is the Component. Components represent referentially transparent stream functions. In higher-order functional terminology, components can be thought of as *closure-converted* local functions. Confer the extra fields and constructor parameters of nested Java classes. 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 13. This is done implicitly at initialization time of the containing computation.

The Component interface can be instatiated by the user to link foreign (stream) functions into a SIG program, but care must be taken lest side effects break the non-sequential semantics.

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 must 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 14.

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. By contrast, independent SIG computations implemented as distinct components are guaranteed to have no implicit race conditions. Thus they may be executed in parallel on multi-core processors (or eventually FPGAs), as long as synchronization of observed outputs is maintained between the steps by a suitable barrier.

Each step of a session consists of three subtasks that update pre-state (tick), inputs (input), and post-state and outputs (action), respectively. Subclasses of Session must override all abstract methods to implement the behavior of the represented component, as well as allocate exclusive storage for 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 "addressing modes" of primitive operations.

Figure 15 shows a typical hand-coded main loop driving a component c with a single output stream and writing successive elements to the console. The loop is subject to external termination condition (halt) and synchronization (wait). The number and types of inputs from in and the possible existence of additional unused outputs are not manifest in the target program.

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

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

Figure 12: Runtime factory; upper level

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

Figure 13: Runtime factory; middle 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 native systems, although that would mean losing the benefits of the JVM platform; see section 6 below.

5. CODE GENERATION STRATEGIES

5.1 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 (tick), 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

```
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);        // inSource used only here
    action () ;
  }

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

Figure 14: Runtime factory; lower level

```
void mainLoop(Component c, Source in) {
  Session s = c.newSession();
  s.setInSource(in);
  s.init();
  while (!halt()) {
    s.step();
    System.out.println(s.getDouble(0));
    wait();                    // possibly real time
  }
}
```

Figure 15: Usage example

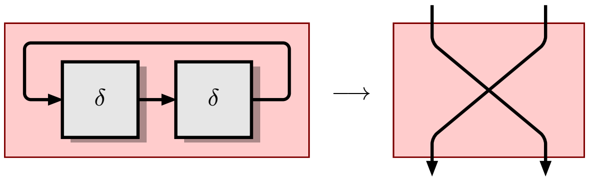

Figure 16: Delay reducing to non-overlayable state variables

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 16 for a counterexample. It is used by the current compiler implementation heuristically; optimal 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 naïve FIFO buffer implementation in terms of state variables, where values are actually transported from the input to the output end, see Figure 17. 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.

5.2 Parallel and Sequential Evaluation

The semantics and core operations of Sig have been designed carefully to allow for maximal potential 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 dis-

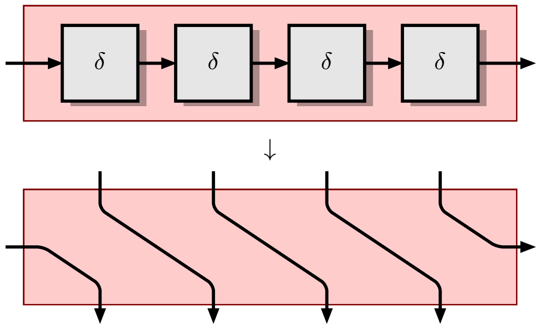

Figure 17: Delay chain reducing to FIFO array

carded after the fact. Compare this behavior to the eager operators `&` and `|`, 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.[11] 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 may scale 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 has recently been developed [18]. The strategy is roughly as follows:

1. Identify, for each node in the data-flow network, the static condition under which it can possibly contribute to the output. This is always a propositional formula over control variables, where destructors, γ and φ nodes on the path contribute positive literals, conjunctions and disjunctions, respectively.

2. Form a decision tree, grouping together nodes with identical conditions.

3. Simplify nested conditions, both relative to their ancestors, and by exploiting disjointness and completeness of alternative constructors in algebraic data types.

4. Group nodes into basic block, guarded by conditional branches that check their respective conditions. Make blocks as large as possible to avoid redundant checks,

[11]Note that Sig is only naturally side-effect free because communication is *not* event-based.

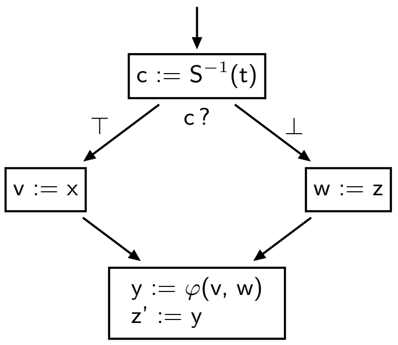

Figure 18: S&H; sequentialized control flow graph

but split and thread where demanded by data dependencies.

5. Since γ nodes end up in blocks that are conditional on the nodes' control inputs, they can be replaced by ordinary moves.

6. Replace data-flow φ nodes by classical, control-flow φ nodes.

The application to the S&H example from Figure 10 is depicted in Figure 18. Note the single decision on c and the dependency-induced split of unconditional statements between the top and bottom block. Note also that the test for $\mathsf{d} := \mathsf{H}^{-1}(\mathsf{t})$ is subsumed by the negation of c, given type information.

We shall illustrate the practical effect of this transformation on code generation for simple examples in section 5.4 below. The subsequent passes (register allocation etc.) are preliminary, and do not yet allow meaningful performance experiments with realistically complex programs.

5.3 Interpretion of SSA Form

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 count (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 19, 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 20. 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.

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 JIT compiler to compensate the interpretative overhead by aggressive inlining and

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

Figure 19: Threaded code substep; interpreter

```
class State {
  public Action pc;

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

  public final boolean[]  registers_control ;
}
```

Figure 20: Interpreter state

specialization. Interpreting sequentialized SSA code with control flow requires only a few small changes, most notably of course the addition of conditional branch operations.

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 (possibly even at runtime) 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.

5.4 Compilation of SSA Form

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.

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 21 for the S&H example. Instructions are compiled to JVM bytecode fragments, which are then glued together to implement Session.action. See Figure 23 in comparison to Figure 10; the six fragments of the latter correspond to the six statements of the former. The resulting code can be loaded directly into the host JVM by a ClassLoader, or stored as class files for external use. The corresponding JVM bytecode produced after the experimental SSA sequentialization pass, for the control flow graph depicted in Figure 18, is not shown. It contains roughly half as many bytecode instructions, and leads to very efficient just-in-time compiled machine code; confer Figure 26 below.

Compilation rules are distributed over the subclasses of Action. Namely, a method named compile is invoked with

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

  // Session method implementations
}
```

Figure 21: S&H; compiled class

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

Figure 22: Threaded code substep; compiler

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, for instance redundant store–load pairs across fragment boundaries, marked with [*] in Figure 23.

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 24.

The redundancy that remains in the depicted machine code, namely that some 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 bytecode-to-native 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 25. Comparison of Figures 24 and 25 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 application of expensive optimizations (here, exhaustive redundant test elimination) for the latter.

Finally, contrast both versions with the machine code produced by the JVM JIT after SSA sequentialization, depicted in Figure 26. This just-in-time compiled code is mostly equivalent to the GCJ version, but trades one additional jump for more code sharing. More importantly, the awkward and superfluous out-of-range case is missing, thanks to type-aware condition simplification.

```
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 valid range)
   102: putfield      #39                 // y

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

Figure 23: 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?, again
       je     .Ld
       test   %r11d, %r11d       # H?, again
       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 valid range)
```

Figure 24: 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 valid range)
```

Figure 25: S&H; machine code (GCJ)

```
action:
       movl  0x38(%rsi), %r11d   # t
       decl  %r11d
       testl %r11d, %r11d        # S?
       jne   .Lb
       movsd 0x18(%rsi), %xmm0   # x
.La:
       movsd %xmm0, 0x20(%rsi)   # y
       movsd %xmm0, 0x30(%rsi)   # z'
       ret
.Lb:
       movsd 0x28(%rsi), %xmm0   # z
       jmp   .La
```

Figure 26: S&H; machine code (JRE, sequential)

6. EXPERIMENTAL EVALUATION

We have tested the performance of both interpreted and compiled code on the JVM with a simple but nontrivial sound synthesis application. It 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 [27]. They translate to 4 and 54 Sig core operations, respectively.

A hand-coded driver loop 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 maximal speed (generally much faster than real time) 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) has been recorded with the precision and accuracy of Java `System.nanoTime()`. Dynamic scheduling 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 effective rates (number of samples produced divided by time spent computing) of 229±3 kHz for interpreted code, and 2740±60 kHz for compiled code, respectively.[12] These figures translate to an average effort of about 152 and 13 CPU cycles per voice-sample, or to a real-time load of 19.6 % and 1.6 %, respectively. The speedup by compilation to bytecode is a factor of 12. By contrast, gains for sequentialized code are relatively insignificant, due to scarce use of control flow in the computationally critical components. All experiments use only a single CPU core for Sig computations, although JVM system threads may run concurrently on other cores.

In summary, the naïve 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.

7. CONCLUSION

The Sig language is specific, in the sense that it trades semantic regularities for algorithmic restrictions that are acceptable only for a particular class of computational patterns, and hence poses specific problems for effective and efficient execution. On the one hand, the purely and totally functional approach, and the rigid minimalist 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 language

[12]Reported errors are median absolute deviations.

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, in terms of on-board libraries that work out-of-the-box and with decent efficiency/safety tradeoffs, for GUIs (javax.swing), sampled audio output (javax.sound.sampled) and MIDI audio input (javax.sound.midi). With these, tangible live demonstrations of SIG programs, as in [27, 29], can be constructed with 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.

7.1 Related Work

The "French school" family of synchronous languages (Signal, Lustre, Esterel, etc.) have set the precedent by demonstrating how data-stream programming can be given sound semantics and expressive frontend notations, although the early family members are remarkedly low-level. Lucid Synchrone [7] has successfully added features of high-level functional programming, in particular algebraic data types, pattern matching and a form of higher-order programming. However, unlike SIG, it is not a purely functional language, and therefore has more complicated semantics with respect to control flow and nested functions, and fewer exploitable degrees of freedom in implementation. On the other hand, Lucid Synchrone has very expressive features for reactive programming that could be put to good use in a future extension of the SIG frontend language.

On the pure side, functional reactive programming [31], in particular its discrete-time continuation-based implementation [19] and arrowized theoretical framework [12], mark the state of the art. We have decided to pursue a different path for many reasons discussed in [26], for instance better support for domain experts in non-embedded DSLs, better integration with pattern matching, and potential interoperability with legacy execution environments and algorithmic code bases.

Some sporadic domain-specific approaches outside the aforementioned families also deserve mention: Faust [21] is a pure, combinator-based data-stream expression language, compiled to C++, with concise operation set and semantics, although it does suffer from scalability issues, and an unfortunate coupling of delay and concrete network layout. Hume [11] has pioneered several ideas that have influenced the design of SIG, most notably the central role of pattern matching, and the conscious and layered tradeoff of expressivity versus analytic and operational properties for abstract low-level programming.

Functional embedded hardware description languages such as Hydra [20] and Lava [3] make elegant use of functional abstraction, and solve code generation problems for low-level target platforms elegantly, although they inherit too much of the bells and whistles of Haskell to meet our requirements of semantic purity.

Although the connections have not be researched thoroughly yet, we expect to relate our frontend approach to configuration of complex networks by staged metaprogramming to its recent, independent use in high-performance computing [15]. At the backend we expect to benefit from recent research done on tracing just-in-time compilation [10], where SSA-based program representations with mostly non-branching control flow are pervasive.

7.2 Future Work

Many directions for particular future work have already been mentioned in passing. The overarching goal is to turn SIG into a comprehensive tool for sound, safe and effective data-flow programming, spanning a wide variety of notations, application domains and target platforms.

At the frontend, the obvious next steps are extensions for nontrivial type and module systems, as well as notation support for reactive and higher-order programming. A possible, controlled relaxation of the ban on recursive computations will also be investigated.

On the theoretical level, the analysis of worst-case execution times and multi-rate systems, and the relationship of SIG semantics to axiomatic theories deserve our attention, as well as the identification of a hierarchy of language layers with associated properties and target domains.

At the backend, our preliminary encouraging results on leveraging the JVM platform for efficient execution need to be corroborated and scaled to more complex case studies.

The semantics of SIG allows for the parallel decomposition of independent data-flow subnetworks. Correspondingly, the passive and thread-safe execution model of SIG components allows their flexible mapping to several cores. However, for actual parallel speedup, static and/or dynamic load balancing would be required.

Last but not least, we plan to investigate the potential of running SIG on platforms obeying the Real-Time Specification for Java (RTSJ), for embedded hard real-time applications. There we expect very interesting results, since many of the problems regarding memory allocation [4] and locking [22] typical for imperative frontend languages are avoided by the semantic model of SIG.

By design of the SIG language, the operation rate of components is abstracted from in its definition. This allows code to be reused across various rates, but also implies that checks whether an actual execution can meet its real-time latency bounds are not integrated in the compilation process. We envisage a cooperative solution where the SIG compiler relies on off-the-shelf worst-case execution time analysis tools, which are expected to be aided greatly in their job by the minimalistic control flow structure of compiled SIG programs.

8. ACKNOWLEDGMENTS

Thanks to members of the software technology group at Chalmers University for helpful discussions.

9. REFERENCES

[1] B. Armstrong and R. Eigenmann. Challenges in the automatic parallelization of large-scale computational applications. In *Proc. of SPIE* 4528, pp. 50–60, 2001.

[2] J. Backus. Can programming be liberated from the von Neumann style? *Comm. ACM*, 21(8), 1978.

[3] P. Bjesse, K. Claessen, M. Sheeran, and S. Singh. Lava: Hardware design in haskell. In *Proc. ICFP 1998*, pp. 174–184, 1998.

[4] G. Bollella, el al. Programming with non-heap memory in the real-time specification for Java. In *Proc. OOPSLA 2003*, pp. 361–369, ACM, 2003.

[5] P. Caspi, D. Pilaud, N. Halbwachs, and J. Plaice. Lustre: A declarative language for programming synchronous systems. In *Proc. POPL 1987*, pp. 178–188. ACM, 1987.

[6] P. Caspi and M. Pouzet. A co-iterative characterization of synchronous stream functions. *ENTCS*, 11:1–21, 1998.

[7] P. Caspi and M. Pouzet. Lucid Synchrone, a functional extension of Lustre. Tech. rep., Université Pierre et Marie Curie, Laboratoire LIP6, 2000.

[8] W. Citrin, R. Hall, C. Santiago, and B. Zorn. Addressing the scalability problem in visual programming through containment, zooming and fisheyeing. In *Proc. Aerospace Conf.*, volume 4, pp. 189–202. IEEE, 1998.

[9] ESA. *ARIANE 5 Flight 501 Failure Report by the Inquiry Board*, 1996.

[10] A. Gal, et al. Trace-based just-in-time type specialization for dynamic languages. *SIGPLAN Not.*, 44(6):465–478, 2009.

[11] K. Hammond and G. Michaelson. The design of Hume: A high-level language for the real-time embedded systems domain. In *LNCS* 3016, pp. 127–142. Springer-Verlag, 2003.

[12] P. Hudak, A. Courtney, H. Nilsson, and J. Peterson. Arrows, robots, and functional reactive programming. In *LNCS* 2638, pp. 159–187. Springer-Verlag, 2003.

[13] J. Hughes. Why functional programming matters. *Computer Journal*, 32(2), 1989.

[14] J. Hughes. Programming with arrows. In *LNCS* 3622, pp. 73–129. Springer-Verlag, 2005.

[15] O. Kiselyov, C.-C. Shan, and Y. Kameyama. Bridging the theory of staged programming languages and the practice of high-performance computing. Tech. Rep. 2012–4, National Institute of Informatics, Japan, 2012.

[16] M. Lepper and B. Trancón y Widemann. Optimization of visitor performance by reflection-based analysis. In *LNCS* 6707, pp. 15–30. Springer-Verlag, 2011.

[17] H. Liu, E. Cheng, and P. Hudak. Causal commutative arrows. *J. Funct. Program.*, pp. 467–496, 2011.

[18] A. Loth. Synthese von Kontrollfluss für eine synchrone Datenflusssprache. Master's thesis, Ilmenau University of Technology, 2015.

[19] H. Nilsson, A. Courtney, and J. Peterson. Functional reactive programming, continued. In *Proc. Haskell Workshop*, pp. 51–64. ACM, 2002.

[20] J. O'Donnell. Hydra: hardware description in a functional language using recursion equations and high order combining forms. In *The Fusion of Hardware Design and Verification*, pp. 309–328. North-Holland, 1988.

[21] Y. Orlarey, D. Fober, and S. Letz. Syntactical and semantical aspects of Faust. *Soft Comput.*, 8(9):623–632, 2004.

[22] F. Pizlo, D. Frampton, and A. L. Hosking. Fine-grained adaptive biased locking. In *Proc. PPPJ 2011*, pp. 171–181, ACM, 2011.

[23] G. Rouleau and S. Popinchalk. Initializing parameters. Matlab Central Blog, 2008. Retrieved 2013-12-31.

[24] J. M. Spivey. *The Z Notation: a reference manual.* International Series in Computer Science. Prentice Hall, 1988.

[25] W. Taha and T. Sheard. MetaML and multi-stage programming with explicit annotations. *Theor. Comput. Sci.*, 248(1-2):211–242, 2000.

[26] B. Trancón y Widemann and M. Lepper. Foundations of total functional data-flow programming. In *EPTCS* 153, pp. 143–167, 2014.

[27] B. Trancón y Widemann and M. Lepper. Sound and soundness – practical total functional data-flow programming [demo]. In *Proc. FARM 2014*, pp. 35–36. ACM Digital Library, 2014.

[28] B. Trancón y Widemann and M. Lepper. Laminar data flow: On the role of slicing in functional data-flow programming. In *Draft Proc. TFP 2015*. INRIA, 2015.

[29] B. Trancón y Widemann and M. Lepper. The Shepard Tone and Higher-Order Multi-Rate Synchronous Data-Flow Programming in Sig. In *Proc. FARM 2015*, ACM Digital Library, to appear 2015.

[30] T. Uustalu and V. Vene. The essence of dataflow programming. In *LNCS* 3780, pp. 2–18. Springer-Verlag, 2005.

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