tSig: Towards Semantics for a Functional Synchronous Signal Language

(Extended Abstract)

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

[1] University of Bayreuth
[2] <semantics/> GmbH

1 Introduction

Functional programming arguably has some of the most powerful mechanisms for abstraction and reuse of program fragments, namely strong and user-definable data types in terms of categorical constructions such as product and coproduct for statical abstraction, and polymorphic higher-order functions for operational abstraction. A well-typed term in a functional language is an extremely concise and mathematically handy notation for data flow compared to, say, a circuit diagram, no matter whether defined visually or algebraically.

However, these abstractions are conspicuously absent from traditional languages for synchronous signals, where both input and output are functions of discrete time: For apparently historical reasons, primitive numeric data types, first-order weakly-typed operations and verbose data flow diagrams are the state-of-the-art style of expression. From a software-engineering perspective of productive and safe programming, signal languages also tend to lack features such as high-level state for control flow, exception handling, support for algebraic data types and symbolic computation, and associated declarative notations such as pattern matching. Cf. [?,?,?].

Here, we describe tSig, the prototype of a language designed to explicitly address many of the above shortcomings. In contrast to the current trend of *functional reactive programming*, we desist from constructing an embedded domain-specific language (cf. [?]), for various reasons: Firstly, we intend to perform substantial static analysis on the language that is probably beyond the power of the static (type) checker of a host language. Secondly, the intended user community of tSig does not consist of functional programming experts, hence the diagnostics given by language tools must state the detected problems directly, not encoded into the type system of a host language. Lastly, tSig is intended to be executed on a number of technologically very diverse platforms, depending on the context of application, from real-time audio to simulation of ecosystem behaviour.

2 Basic Design

The key idea of tSig is to have coexisting and fully compositional notations for terms and circuit-like data flow, such that the programmer may freely choose either idiom for any computational task. Diagrams are more expressive concerning the shape of data flow, but they do not scale well to more complex programs. The transition between the two perspectives can be understood as a matter of syntactic discipline: Consider a pure term language in the sense of classical, timeless functional programming. If the language has a *let*-construct for local bindings, then an internal canonical form, the so-called *A-normal form* can be defined. It binds every subterm to a local name, such that every expression is of the form *let* $(y_1, \dots, y_n) = f(x_1, \dots, x_m)$ *in* z, where z is either a bound variable or in A-normal form again. This can be seen as an algebraic notation for a data flow diagram, where f transduces input channels $x_1, \dots, x_m$ to output channels $y_1, \dots, y_n$ in a context z. The resulting data flow diagram is an acyclic graph for non-recursive *let*. Arbitrary finite graphs could be encoded with recursive *let*, but their interpretation requires fixpoint semantics and has rather complicated properties. Hence we do not currently consider cyclic data flow; see section 3.2 below.

In a data flow diagram, the individual components can easily be reinterpreted from one-shot computations to stream transducers, thus implementing discrete-time signals. tSig gives such signal semantics to arbitrarily complex nested terms, in a way compatible with the stream interpretation of their A-normal forms. Assuming that signals may have different frequencies, classical functional programming can be retrieved as the pathological case of frequency zero.

3 Semantic Categories

3.1 Signal Processing with Stream Functions

Historically, there are three important semantic characterizations of stream functions: Firstly, *causal* stream functions $f : A^\omega \to B^\omega$ where infinite streams are used holistically as input and output, but the output element at a particular instant in time may only depend on current and past, but never on future, input elements. Secondly, functions in the style of functional reactive programming where every input element is mapped to a pair of an output element and a continuation function, as expressed by the recursive type equation $A \xrightarrow{\star} B = A \to B \times (A \xrightarrow{\star} B)$. Lastly, the state-based or coinductive style where the elementwise mapping of input element to output elements is coupled with a state transition, as expressed in the non-recursive type equation $A \xrightarrow{S} B = S \times A \to B \times S$. (The switched positions of state and I/O are deliberate and a convenient convention, see below.) We shall follow the latter approach, because it is most closely related to low-level implementations, requiring only first-order iteration (loops with variables of non-function types) for its operational semantics.

A stream function can be implemented in tSig if it is specified by a *constructive* elementwise step function of type $A \xrightarrow{S} B$ built from a suitable algebra of basic step operations. The obvious downside of the state-based approach, namely that, for each stream function, the state space and initial state must be given in addition to the elementwise step, is mitigated by the very same algebra: We postulate that the relevant class of stream functions can be specified by two classes of atomic steps, namely *stateless* computations and *elementary delay*, together with synchronous composition of stream functions. Stateless computations are embedded into the setting as having unit state $S = 1 = \{*\}$, such that the initial state is determined trivially. Hence the type signature of a stateless lifting of an elementwise function $f : A \to B$ has the type signature $f^\omega : A \xrightarrow{1} B$ and the defining equation $f^\omega(*, a) = (f(a), *)$. Elementary delay δ (for a single step in time) is specified by the polymorphic type signature $\delta_A : A \xrightarrow{A} A$ and the defining equation $\delta_A = \mathrm{id}_{(A\times A)}$. (Note how the switching of state and I/O positions is exploited.) The synchronous composition of steps $f : A \xrightarrow{S} B$ and $g : B \xrightarrow{Q} C$ has the type signature $g \diamond f : A \xrightarrow{S\times Q} C$ and the defining equation $(g\diamond f)((s, q), a) = (c,(s', q'))$ where $(b, s') = f(s, a)$ and $(c, q') = g(q, b)$. It is easy to show that the stream function specified by the step $g \diamond f$ and the initial state (s, q) is equivalent to the composition of the stream functions specified by steps g and f with initial states s and q, respectively. With synchronous composition and (elementwise) product, arbitrarily complex terms of stream functions may be constructed from the atomic steps. Note that, in accordance with section 2 above, general recursion is not allowed, hence functions may be required to be *total*.

3.2 Coproducts

(Cartesian) products as a structuring device are ubiquitous even in the most primitive data-flow language designs. By contrast, the dual notion of coproducts (disjoint unions) is far less well-established, although it plays a certain role in functional programming. A wide variety of design concepts can be reduced to coproducts: algebraic datatypes (as a coproduct of constructor cases), exception handling (as the Kleisli category of a coproduct monad), finite automata as a large-scale device of control flow, and last but not least the transformation known as *defunctionalisation*, which maps higher-order to first-order functional programs by replacing each function-type argument with an ad-hoc algebraic datatype, and its use with a local interpreter.

We conjecture that the general tendency in language design to privilege products over coproducts results in negative effects from a software-engineering perspective, namely in *premature encoding*, an analog to the infamous *premature optimization*. It denotes the tendency to misrepresent structures that are logically disjunctive in nature, and hence naturally represented by coproducts, by a conjunctive approximation that can be represented by products. This asymmetry is acerbated by product-centric machine-level similes, such as a bus of independent channels, or a set of random-access memory cells.

A simple but effective countermeasure against product dominance is the firm integration of case distinctions based on pattern matching into a language. Since pattern matching naturally specifies *partial* functions, a variety of implicit semantics for systems of pattern-based partial function equations have been proposed, such as *first-fit* and *best-fit* rules. tSig takes a rigorous and explicit approach by distinguishing total and partial function expressions as different syntactic and semantic categories. Partial expressions arise from the use of pattern matching and are combined by a small set of meta-logical operators, namely `&` (intersection), `|` (disjoint union, commutative but with proof obligation) and `/` (overriding union, non-commutative). The transition to the total domain is syntactically explicit and carries a proof obligation. The fact that pattern matching may fail dynamically for certain inputs is reflected in the semantics of tSig by assigning a *guard*, that is, a formal expression that doubles as the static specification of a subset (of a coproduct type) and a runtime test, to each matching operation. Failure propagates across meta-logical operators and is reflected in the propagation of guards to logically connected operations, in the logically obvious way.

In the absence of recursion, partial expressions can be given simple semantics in the style of the relational algebra of database theory: Each operation (of the form *let* $(y_1, \dots, y_n) = f(x_1, \dots, x_m)$ in an A-normal form term) is assigned a potentially infinite table with columns (attributes) $x_1, \dots, x_m, y_1, \dots, y_n$ and a functional dependency of the y-rows on the x-rows. Total (stream) functions are represented by their extension (treating I/O and state variables the same), runtime tests on subsets by partial identity relations. Then `&`, `|` and `/` can be represented by the relational operations $\bowtie$ (join), $\dot\cup$ (disjoint union) and $\oplus$ (override), respectively.

It is easy to show that synchronous composition is a special case of relational join, and that the semantic representations are compositionally well-behaved under a reasonable set of assumptions on the structure of data flow: The join of a finite set of functional relations is well-defined and functional if the data flow graph is cycle-free. (There is an edge in the data flow graph between two operations f and g if an output of f coincides with an input of g). Furthermore, the guard of the aggregate is the intersection of individual guards if outputs are pairwise disjoint.

In terms of circuit diagrams, these assumptions mean precisely that instantaneous feedback is forbidden, and that output channels may not be connected to each other. However, feedback with delay is an important design pattern and should be supported. This is achieved by splitting a delay operation into two independent operations, namely an identity between pre-state and output, and another identity between input and post-state. This way, delayed feedback does not appear as cycles in the data flow graph. From the functional term perspective, general recursion is not allowed, but there are certain apparently recursive equations that translate to delayed feedback, and hence are acceptable. We conjecture that general recursion is not needed for typical signal processing applications, to the effect that the infamous problems caused by fully recursive calculi of computation can be safely ignored here.

3.3 Mutual embedding

The free composition of the partial and the total world can now be formalized by a pair of mutual semantical embeddings. A total (stream) function translates trivially to relational semantics by takign its extension and forgetting the different roles of I/O and state variables. On the other hand, a partial expression with relational semantics can be abstracted as a total function, given that several conditions hold: All inputs are bound to funciton arguments or intermediate outputs; all results have total guards; the above conditions for join hold; all postulated disjointness of unions it witnessed by guards.

4 Conclusion

Apparently all existing technical solutions for signal generation and real-time signal processing include at least one paradigm break, normally between a "configuration" and a "signal" layer of the system architecture, where the semantic model changes as well as the language syntax. The reasons are historical as well as social: Most existing systems are quite old, and their authors merely dabbled in the field of language design and compiler construction. [?,?,?]. But even newer and more professional approaches have similarly fundamental gaps [?,?].

In contrast, tSig proposes a monolithic approach: Functions, given by terms denoting both configuration and basic signal calculation operations, and data flow networks, given as circuit descriptions, are embedded compositionally into a uniform, semantically rigorous calculus. The effective implementation of tSig requires that the expressions of this compound language be iteratively normalized, and advanced techniques of functional program transformation (defunctionalisation, pattern matching) be applied. We hope that the resulting programming system will allow domain experts to concentrate on modeling the specific domain problems, in areas as diverse as live electronic music and simulation of environmental processes, being relieved from the burden of manually translating the intended logic signal flow into some technical encoding.

The authors are confident that the tSig approach will carry far. We are looking forward to practical experience on its concrete limitations, and to insights whether need arises to add expressivity to the framework, and how to do it.

References

1. The Csound Manual (2005), `http://csounds.com/manual/html/indexframes.html`
2. Pure Data Homepage (2011), `http://puredata.info/docs`
3. Nilsson, H., Courtney, A.: Yampa (2008), `http://hackage.haskell.org/package/Yampa`

4. Orlarey, Y., Gräf, A., Kersten, S.: DSP programming with Faust, Q and SuperCollider. In: LAC2006 (2006), `http://lac.zkm.de/2006/papers/lac2006\_orlarey\_et\_al.pdf`
5. Wilson, S., Cottle, D., Collins, N.: The Supercollider Book. The MIT Press (2011), `http://supercolliderbook.net`