Sessional Dataflow

Short Paper

Dominic Duggan Jianhua Yao
Stevens Institute of Technology
dduggan@stevens.edu

Abstract

The purpose of sessional dataflow is to provide a compositional semantics for dataflow computations that can be scheduled at compile-time. The interesting issues arise in enforcing static flow requirements in the composition of actors, ensuring that input and output rates of actors on related channels match, and that cycles in the composition of actors do not introduce deadlock. Ultimately the purpose of sessional dataflow is to support dynamic operations on subnets, ensuring that assumptions underlying static scheduling are not violated by operations such as subnet update and reconfiguration. This account focuses on a simplified case of sessional dataflow, to draw out the key points of the approach.

Categories and Subject Descriptors D.1.3 [*PROGRAMMING TECHNIQUES*]: Concurrent Programming—Parallel Programming; D.3.2 [*PROGRAMMING LANGUAGES*]: Language Classifications—Data-flow languages; F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages—Operational semantics

General Terms Languages.

Keywords Dataflow, semantics, types.

1. Introduction

Dataflow has an honored tradition in declarative parallel programming [9, 10]. It has renewed significance today, given the importance attached to deterministic parallelism as a way of coping with the challenges of scalable parallel programming. Many of the applications of parallel processing are in stream processing, for example of streaming multimedia data, again motivating interest in dataflow processing. Part of the challenge of dataflow processing is in scheduling the execution of dataflow graphs without unbounded buffering of data between actors in the net. In signal processing, synchronous dataflow has enjoyed some success for multi-rate applications, with many variations of the basic idea developed over the years [11].

The purpose of sessional dataflow is to provide a compositional semantics for dataflow computations that can be scheduled at compile-time. To explain why compositionality is important, in synchronous dataflow and its variants, a dataflow graph is described in terms of atomic actors, and flow edges connecting them. A compositional semantics allows both atomic actors, and subnets resulting from the composition of actors, to be viewed uniformly as dataflow actors. Compositionality is obviously important for scaling dataflow programming. The interesting issues arise in enforcing static flow requirements in the composition of actors, ensuring that input and output rates of actors on related channels match, and that cycles in the composition of actors do not introduce deadlock. Ultimately the purpose of sessional dataflow is to support dynamic operations on subnets, including update and reconfiguration, while ensuring that assumptions underlying static scheduling are not violated by these operations . In this account, we focus just on a simplified case of sessional dataflow, to draw out the key points of the approach.

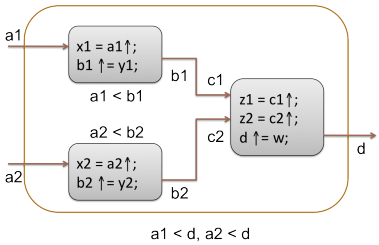
DAMP '12 January 28, 2012, Philadelphia, PA, USA.

2. Actors and Dataflow Nets

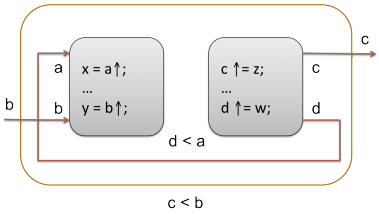
In this section, we provide an example of a limited form of static dataflow that is expressive enough for our purposes for now. So-called synchronous dataflow (SDF), more aptly named *static dataflow*, assumes that on each actor "firing," a statically fixed number of inputs is consumed on each input channel and a statically fixed number of outputs is produced on each output channel. Synchronous dataflow allows static scheduling of multirate applications. To simplify actor composition, and since multirate applications are a specific class of applications that may or may not be useful for certain application domains, we restrict our semantics further to single-rate applications. This means that a single input is consumed on each input channel, and a single output produced on each output channel, on each firing of an actor. This enables all scheduling decisions to be made at compile time, and enables all buffering of data between senders and receivers to be eliminated

An actor specification needs a few other aspects to be defined. Although firing is atomic in SDF, our semantics for firing is implemented in a C-like core language, that consumes and produces messages one at a time. For modeling the states of an actor, we use the notion of *flowstate*, that tracks the state of an actor during a firing cycle. In addition, we need a specification of the input and output channels of an actor, that will subsequently be coupled with channels for other actors to form a dataflow network. An example of a specification for an actor in our type system is provided by the following:

```
actor interface IActor
{
   in channel<float> a;
   in channel<float> b;
   out channel<float> c;
   causality a < b, a < c;
   flowstate ?a, ?b , !c.
}
```

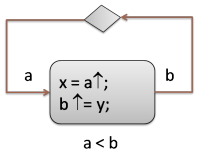


(a) Composite Actor with Internal Parallelism

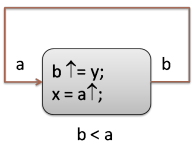


(b) Composite Actor with Input depending on Output

Figure 1. Parallel Inputs and Outputs



(a) Feedback Loop Requiring a Delay



(b) Feedback Loop Not Requiring a Delay

Figure 2. Causality and Feedback

This is the expression of an actor type in our system. The type specifies input and output communication channels, and allowable communications on those channels using a flowstate specification. The flowstate rule in the example above requires that the actor consume an inputs on `a` channel and an input on the `b` channel, and produces an output on the `c` channel. In what order should these inputs and outputs be performed? It is tempting to restrict firings so that all inputs are consumed before any outputs are produced, but once we compose actors into composite actors (dataflow nets), it is no longer possible to ensure this. Even if we restricted actors to only inputs or only outputs, but not a mixture of the two, we could still have scenarios where the consumption of an input in one actor depended on the production of an output in another actor. Therefore we must allow for aribtirary interleavings of inputs and outputs, while avoiding deadlock where for example an output channel and input channel are linked to the same underlying channel.

Therefore we enrich actor interfaces with a notion of *causalities*. This is demonstrated in the example above, where the causalities specify that outputs on channel `c` depend causally on inputs on channel `a` and channel `b` (`a` $<$ `c` and `b` $<$ `c`, respectively). In this simple case, causalities flow from all inputs to the single output, but more generally in a composite actor with parallel threads, there will typically be more refined causal relationships.

Fig. 1(a) and Fig. 1(b) demonstrate two composite actors. Fig. 1(a) provides a composite actor where two actors on the left consume inputs in parallel, and these are then consumed in a particular order by the actor on the right. We define causalities to reflect the fact that message sending is asynchronous, so in some sense the outputs of an actor, once their causally preceding input events occur, may occur in an indeterminate order.Fig. 1(b) provides another composite actor, one where the output on channel `c` must causally precede the input on channel `a`, since the output on internal channel `d` causally precedes the input on internal channel `a`. outputs are parallel despite the fact that they are produced by a single sequential thread.

Our actor semantics is effectively a limited form of *cyclostatic dataflow* [2]. In the latter, an actor has a finite state control logic, and transitions between states of this logic on each firing. Its firing pattern then depends on the current state that it is in. Because we are providing specifications for input consumption and output production at the level of individual communication steps, the semantics of a "firing" in the traditional SDF sense is non-atomic, and we are essentially tracking a finite state control logic in the process of a firing. It is possible to extend this approach to full cyclostatis dataflow, by embedding causality sets into flowstates, so that the specification of an actor behavior transitions through "stages", at each of which it exhibits a particular firing behavior. We omit the details in this introductory article.

The specifications of the input-output behavior make no reference to the actual values that are transmitted. For simplicity we have assumed that the channel types are fixed, so that only values of the declared type may be transmitted on a channel. In practice it may be beneficial to relax this restriction, though we defer these considerations to future work.

An implementation of this actor specification uses a conventional programming language to define the actor behavior, in the style of Kahn's original proposal for dataflow networks:

```
actor Actor implements IActor
{
   float x, y;
   loop { x = a↑; y = b↑; c ↑= (x+y); }
}
```

The definition of the actor implementation inherits the interface specification: input and output channels, causalities and flowstates. The operation for reading from an input channel `c` is denoted by $c\uparrow$, while the operation of writing a value to an output channel is denoted by `c` $\uparrow=$ v. The flowstate in the actor specification establishes behavior obligations for its execution, subject to the constraints imposed by the causalities.

Fig. 2 clarifies the point of the causalities. In general the issue is to detect when connecting two open channels in the same actor may introduce a cycle in the dependencies between the channels. To avoid this cycle which would lead to deadlock, the connecton of the channels is required to introduce a "delay," by filling the buffer for the channel with default initial values. Fig. 2(a) illustrates this, where the single output channel of an actor is connected to its input channel. The actor first reads from the input channel `a`, before outputing to the output channel `b`. Note that we do not try to track data flow dependencies, our interest is in the control flow dependency from the consumption of input on `a` to the production of output on `b`. Suppose these two open channels are connected to the same shared channel. We assume an obvious causality from the output end of a shared channe to the input end, so this binding will introduce the causality `b` $<$ `a`. This will introduce a cycle in the causalities, which we cannot allow. Therefore in this case the connection of two channels `a` and `b` on the same underlying channel must include a delay, as indicated by the diamond in Fig. 2(a).

In the example in Fig. 2(b), on the other hand, the appending of data to the output buffer is done before input is performed. This results in the causality `b` $<$ `a` for the actor body. This does not necessarily mean that data flows from the output event to the

input event, but there is at least a causal dependency, in that the occurrence of the output event is a prerequisite for the occurrence of the input event. If these input and output open channels are connected using the same shared channel, then the output produced on the output channel does indeed propagate to the input channel to be consumed, but this is immaterial as far as scheduling is concerned. The causality `b` $<$ `a` that is added as a result of this binding of channels `b` and `a` is basically a no-op, and no scheduling cycle is introduced, so a delay is not necessary.

We have so far considered the "programming-in-the-small" aspects of ensuring that an actor satisfied its behavior specification. We now consider the "programming-in-the-large" aspect of ensuring that the composition of actors is in some sense well-formed. This issue is greatly simplified by our assumption of single rate actors.

In general, the approach to composition of actors is to provide a binary connection operation for linking the output data channel on one actor with the input data channel of another. We denote this operation by `connect(A.a,B.b)`. Here it is important to distinguish between open channels and shared channels. An *open channel* is one of the form described in the previous section, a channel that is declared in an actor interface, and referred to in an actor body by operations for consuming messages and appending messages to message buffers.

For deterministic semantics, it is important that there be no nondeterministic contention for access to a channel. For example, a nondeterministic merge might be provided by allowing multiple actors to send simultaneously to the same merge channel. Synchronization on access to the channel, performed by the compiler and runtime system, could ensure that the message append operations are atomic. However the order in which messages are appended would be nondeterministic, based on dynamic scheduling of actors and interleaving of their multithreaded executions. While nondeterministic merge is a useful operation in some cases, our intention is to establish a baseline that ensures deterministic execution, before considering later how to extend this with nondeterminism.

Our approach is to ensure exclusive access to a communication channel between two actors, the one actor sending on that channel and the other actor receiving on that channel. The connection operation `connect(A.a,B.b)` creates a new private communication channel, binds the `a` output channel on the `A` actor to the output part of this new private channel, and binds the `b` input channel on the `B` actor to the input part of this new private channel. We refer to such a private channel as a *shared channel*. Since (for now) we provide no way for an actor to send any of its communication channels to another actor, exclusive access by a pair of actors to a shared channel is ensured[1].

What is the result of connecting actors? The semantics should be compositional, so that the connection of two actors should be indistinguishable to outside observers from a single actor. For synchronous dataflow, the only part of the outside interface of note for a combined actor is the remaining open channels after a connection, and the firing rates for those channels. In a multi-rate system, connecting two actors with potentially different flow rates on the different channels on which they are connected might require adjustment of their respective rates.

In the relatively simple semantics described in this article, we avoid the problem of variable aliasing by not allowing aliased references to actors. This is compatible with approaches such as for example session types that similarly constrain the bindings of variables to resources whose usage is tracked by linear or affine types. If we relax this restriction, then we must face the issue of how to deal with scenarios such as the following:

```
IActor2 f (IActor A, IActor B)
{ return connect(A.a,B.b); }
```

How can we prevent a scenario such as `f(A0,A0)`, for some actor `A0` that implements the `IActor` specification? This is a known issue in type systems for safe resource management. However, for the purposes of this article, we avoid these issues in the interests of brevity by simply not providing a syntax for the copying of references.

[1] Actor interfaces include polarity information about access to channels, and the connection operation requires that the accesses by the actors be complementary. It is indeed possible that the actors being connected are the same. The connection operation requires knowledge of when it is the case that the same actors are being connected, as we will see.

$T \in$ Type	::=	$\mathsf{float} \mid AS$
	$\mid$	$\mathsf{channel}\ \pi \mid \mathsf{update}\ AS$
$AS \in$ Actor sig	::=	$\mathsf{actsig}(K, O, FS)$
$K \in$ Causalities	::=	$\{\} \mid \{a < b\} \mid K_1 \cup K_2$
$O \in$ Open channels	::=	$\{\} \mid O_1 \cup O_2$
	$\mid$	$\{(\mathbf{c} : c : \mathsf{channel}\ \pi)\}$
$\pi \in$ Polarity	::=	$\mathtt{+} \mid \mathtt{-} \mid \pm$
$\alpha \in$ Event	::=	$\mathtt{!}a \mid \mathtt{?}a$
$ES \in$ Event set	::=	$\{\} \mid \{\alpha\} \mid ES_1 \uplus ES_2$
$FS \in$ Flowstate	::=	$ES \mid (FS_1; FS_2)$
	$\mid$	$FS^* \mid FS^\omega$

Figure 3. Abstract syntax of $\mathbf{S_{YNC}}$ Types

3. A Kernel Language

The syntax of types is provided in Fig. 3. For simplicity we assume a single base type of float, for floating point values. Similarly we assume that only floating point values are exchanged between actors in each message exchange, so the channel type does not need to describe the type of data exchanged on the channel. Although the polyadic pi-calculus generalizes messages to include tuples of values, this is not necessary in our current framework because channels are private to a single sender and receiver. We therefore do not need a mechanism for atomically sending several values, whereas this is necessary in the pi-calculus because channels may be shared between multiple senders and receivers. We do record polarity information for a channel, which records whether it can be used by that actor for input (polarity $\mathtt{+}$), or for output (polarity $\mathtt{-}$), or both (polarity $\pm$).

The type of interest is that of actors. An *actor signature* has several parts, as we have seen:

1. A *causality set* K is a set of causality constraints between events, of the form $a < b$, that reflects dependencies between events.
2. A set of *open channels* O. The channel has two names: its *internal name* c by which it is identified internally in the actor, and its *external name* $\mathbf{c}$ by which it is referenced when composing with other actors. We distinguish these names in order to allow renaming apart of internal channel names when actors are composed, without affecting the external interface.
3. The *flowstate* of an actor records its expected firing behavior, as recorded by occurrences of events in event sets. For simplicity we assume single-rate systems, so in a given firing there is just one communication along each channel in a firing. In an extension of this system with synchronous dataflow, a flowstate includes a multiplicity on each channel, recording how many times it is used on a firing. In the extension with cyclostatic dataflow, causality sets are embedded in event sets.

$v \in$ Vals	::=	$n \mid a \mid x$	
$s \in$ Stmt	::=	$(\mathsf{var}\ x = e;\ s)$	Bind variables
	|	$\mathsf{if}\ (v)\ s_1;\ \mathsf{else}\ s_2$	Conditional
	|	$\mathsf{while}\ (v)\ s$	Loop
	|	$\mathsf{loop}\ s$	Infinite loop
	|	$\mathsf{run}\ v$	Run a network
	|	$(s_1; s_2)$	Sequential
$e \in$ Exp	::=	$f(v_1, \ldots, v_k)$	Builtin
	|	$v_1 = v_2$	Assignment
	|	$c\uparrow$	Receive a message
	|	$c\uparrow\!=\!v$	Send a message
	|	$\mathsf{actor}(K, O, FS, s)$	Atomic actor
	|	$\mathsf{connect}(v_1.\mathbf{c}_1, v_2.\mathbf{c}_2)$	Connect two actors
	|	$\mathsf{connectWithDelay}(v_1.\mathbf{c}_1, v_2.\mathbf{c}_2)$	Connect with delay

Figure 4. Abstract syntax of $\mathbf{S_{SYNC}}$ Statements

Fig. 4 provides the abstract syntax for programs in $\mathbf{S_{SYNC}}$. There are three constructrs for defining actors: the definition of an atomic actor, and two operations for connecting actors on complementary open channels, with and without a delay. The run operation runs a dataflow network for which all channels are bound.

4. Related Work and Conclusions

Sessional dataflow comes out of the realm of linear and affine type systems for statically checking the safe usage of limited resources. The approach of session types [8] is commonly motivated by its support for safe Web services. In the simplest case, session types are used to mediate the exchanges between two parties in a dyadic interaction. Each session offers a "shared channel" (different from our use of the terminology), essentially an service endpoint URL that a client connects to. On connection, a new server thread is forked and a private session channel is established between the client and this thread. This channel has a behavioral type that is essentially an abstract single-threaded process, that constrains the communications between the parties. Since only the client and the server share their private channel, the execution is in fact deterministic.

Although sessional dataflow might appear at first related to session types, the connection is actually rather weak, because of the nature of the interactions in dataflow. The closest our system comes to a session types system is in the behavioral constraint on the behavior of an actor, in terms of matching the specified input and output data rates on each firing specified on an actor interface. However this behaviorial specification only constrains a single actor, and places no constraint on the behavior of its neighboring actors (upstream or downstream). Furthermore a session type specifies, for each participant in an interaction, a very precise single-threaded behavior, in terms of data exchanged on the private session channels at each point in the execution. In contrast, the behavioral specification for an actor in sessional dataflow is declarative, specifying expected communications subject to causality constraints. Deniélou and Yoshida [7] describe a version of session types that allows a dynamic number of participants in a session protocol. As with other approaches to session types, the approach is to provide operational specifications of participant behaviors, using a top-down approach where one reasons from the specified global protocol to the behavior of individual participants. In contrast, the sessional dataflow approach is bottom-up and declarative, specifying declarative causality constraints on individual actors independent of whatever interactions they are integrated into.

Another related line of work is in synchronous languages for real-time and embedded systems. Such languages assume a "clock" on all computations, with variables representing potentially infinite streams of values, indexed by clock ticks. Here the most relevant example for sessional dataflow is that of Lustre [4], a language that is a dataflow language in the tradition of Lucid, [1], and is a synchronous language in the sense of the synchronous languages such as Esterel [3], but which we cannot call a synchronous dataflow language for fear of confusing the reader. The constraints on the synchronous languages preclude any need for buffering, since all actors operate in lock step on the same clock. The theory of these "synchronous," "dataflow" networks has been described in terms of synchronous Kahn networks [5], which have the property that no buffering is required at all between actors, since all execution is synchronous and governed by a common clock. This is clearly a very strong restriction, albeit one that facilitates compilation of programs to hardware circuits. The theory of N-synchronous Kahn networks [6] relaxes this restriction, allowing different actors to have their own clock rates, and allowing buffering between actors to match their clock rates. It is therefore very much related to the approach of synchronous dataflow.

Our language for composing actors is simple, obviously so in order to simplify the semantics. A more general operation for the composition of several actors simultaneously can be added as syntactic sugar. There are many extensions of the form of synchronous dataflow that we have investigated for sessional dataflow, and it appears plausible that all of these extensions can be applied to sessional dataflow, since most of them translate into synchronous and cyclostatic dataflow. However our interest is in other extensions for sessional dataflow that go beyond synchronous dataflow.

References

[1] E. A. Ashcroft and W. W. Wadge. *Lucid, the dataflow programming language*. Academic Press, 1985.

[2] G. Bilsen, M. Engels, R. Lauwereins, and J.A. Peperstraete. Cyclo-static data flow. In *International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, volume 5, pages 3255 –3258 vol.5, May 1995.

[3] F. Boussinot and R. de Simone. The Esterel language. *Proc. IEEE*, 79:1270–1282, September 1991.

[4] F. Boussinot and R. de Simone. The synchronous data flow programming language Lustre. *Proc. IEEE*, 79:1305–1320, September 1991.

[5] Paul Caspi and Marc Pouzet. Synchronous kahn networks. In *in the first ACM SIGPLAN international conference on Functional programming (ICFP '96)*, 1996.

[6] Albert Cohen, Marc Duranton, Christine Eisenbeis, Claire Pagetti, Florence Plateau, and Marc Pouzet. N-synchronous kahn networks: a relaxed model of synchrony for real-time systems. In *in ACM International Conference on Principles of Programming Languages (POPL '06)*, pages 180–193. ACM Press, 2006.

[7] Pierre-Malo Deniélou and Nobuko Yoshida. Dynamic multirole session types. In *ACM Symposium on Principles of Programming Languages*, pages 435–446, New York, NY, USA, January 2011. ACM.

[8] Mariangiola Dezani-Ciancaglini and Ugo De'Liguoro. Sessions and session types: an overview. In *Proceedings of the 6th international conference on Web services and formal methods*, WS-FM'09, pages 1–28. Springer-Verlag, 2010.

[9] Stephen A. Edwards. *Languages for Digital Embedded Systems*. Kluwer Academic Publishers, 2000.

[10] Gilles Kahn. The semantics of a simple language for parallel programming. In *Information Processing 74: Proceedings of the IFIP Congress*, pages 471–475, Stockholm, Sweden, August 1974. North-Holland.

[11] Edward Lee and David Messerschmitt. Synchronous data flow. *Proc. IEEE*, 75(9):1235–1245, September 1987.