Parameterized Dataflow

Extended Abstract

Dominic Duggan
Department of Computer Science,
Stevens Institute of Technology
Hoboken, New Jersey, USA.
dduggan@stevens.edu

Jianhua Yao
Department of Computer Science,
Stevens Institute of Technology
Hoboken, New Jersey, USA.
jyao1@stevens.edu

Dataflow networks have application in various forms of stream processing, for example for parallel processing of multimedia data. The description of dataflow graphs, including their firing behavior, is typically non-compositional and not amenable to separate compilation. This article considers a dataflow language with a type and effect system that captures the firing behavior of actors. This system allows definitions to abstract over actor firing rates, supporting the definition and safe composition of actor definitions where firing rates are not instantiated until a dataflow graph is launched.

1 Introduction

Dataflow or stream processing is becoming increasingly important, with the growing prevalence of signal, video and audio processing, particularly on mobile devices. Dataflow processing is a good match with multicore and GPGPU parallel architectures that are now prevalent on desktop computers, and will shortly be available on consumer mobile devices. The data parallelism of such architectures is at least potentially a good match with the demands of stream processing applications. The synthesis of these architectures with stream processing may provide a domain-specific solution to the challenge of programming the new generations of parallel computing architectures.

Our starting point is a computational model similar to that originally proposed by Kahn [11]. This provides for a network of sequential *actors*, each implemented in a conventional sequential language such as C or Algol. Actors are connected by communication buffers on which they can send and receive data. A key point is that actors cannot nondeterministically select among inputs on several input channels, nor can they test input channels for available inputs (so polling cannot be implemented). This restricts each actor to a completely deterministic semantics. The combination of implicit parallelism and deterministic execution makes dataflow computation a good fit with some of the current thinking of how best to successfully exploit the parallelism available in modern multicore and GPGPU architectures, in those domains where the dataflow paradigm is applicable.

In the embedded systems and digital signal processing community, a very useful class of restricted Kahn networks has been identified, the so-called *synchronous dataflow (SDF)* [14] networks. SDF networks enable static scheduling for multi-rate applications. More recently, domain-specific languages such as Streamit [15] have been defined, based on the principles of SDF, but also providing support for compiling programmer code to run on modern parallel architectures.

Sessional dataflow provides a framework for providing compositional descriptions of dataflow networks [7]. A type and effect system captures the firing behavior of actor bodies, and this information is used to ensure that the composition of actors does not deadlock. For simplicity, that simple effect system did not consider variable firing rates for actors, so for example no communication was possible within a loop (finite loops were still useful for example for windowing computations). Although actors

M. Tribastone and H. Wiklicky (Eds.): International Workshop on
Quantitative Aspects of Programming Languages and Systems (QAPL'16)
EPTCS 227, 2016, pp. 63–81, doi:10.4204/EPTCS.227.5

being combined could have different rates, and adaptation of rates was part of the static checking of actor composition, these rates were hard-coded into the software.

In this article, we consider an approach to incorporating variable actor firing rates into dataflow descriptions. This allows the description of a dataflow graph to be parameterized by the firing rates of various actors in the graph. Actors handle variable firing rates by performing communication in loops. Central to this approach is the introduction of arrays of channels and arrays of actors, and a special form of comprehension for describing effects with this form of rate information. For example, in the language described in this paper, the following loop performs downsampling on an input channel `in` by echoing every second input to the output channel `out` (where `i` and `o` are the type-level names for the input and output channels, respectively):

```
s :  Size(∞)
i :  Channel(0,s)
o :  Channel(0,s)

sz : Size(s)
in : Channel(+,Integer,i)
out : Channel(−,Integer,o)
for (t,x ∈ 1..sz) {
  Integer w = in?;
  when (2 | x) out!w;
}
```

In this example, `s` is a type-level quantity that is used to model firing rates for channels. There are two type-level channel names, `i` and `o`, whose declarations specify that there are no delays in communication on those channels, and which have a bound of `s`. These channel names are used in the declaration of channel variables `in` and `out`, respectively. Communication on these channels is modeled at the type level by input and output events on the corresponding type-level channel names (`i` and `o`).

The size parameter `s` is used in the declaration of a value-level parameter `sz`, that specifies the rate of communication on the channels. For example, `sz` is used as the bound on the loop where the communication is performed. The flowstate for this loop is a sequential composition of two comprehensions

$$\{\mathtt{i}? \mid \mathtt{t} \leftarrow \underline{1}..\mathtt{s}\};\{\mathtt{o}! \mid \mathtt{t} \leftarrow \underline{1}..\mathtt{s}, (\underline{2} \mid \mathtt{t})\}$$

where the type-level parameter `s` models the value-level loop bound `sz`. This can be abbreviated as a flowstate that just counts the number of communication events on each channel:

$$\mathtt{s} * \mathtt{i}?;(\mathtt{s}/\underline{2}) * \mathtt{o}!.$$

As a variation on this example, demonstrating the usefulness of event comprehensions, we can have the code read from an array of channels, combining several paths in a dataflow graph:

```
i :  Channel(0,1)[s]
in : Channel(+,Integer,i)[s]
for (t,x ∈ 1..sz) {
  Integer w = in[x]?;
  when (2 | x) out!w;
}
```

$$
\begin{array}{rcl}
\kappa \in \text{Kind} & ::= & \mathsf{Type} \mid \mathsf{Channel}(b, \tau_{limit}) \mid \mathsf{Channel}(b, \tau_{limit})[\tau] \mid \mathsf{Size}(\tau) \\
\tau \in \text{Simple Type} & ::= & t \mid \underline{n} \mid \underline{\infty} \mid (\tau_1 + \tau_2) \mid (\tau_1 - \tau_2) \mid (\tau_1 * \tau_2) \mid (\tau_1 / \tau_2) \mid \\
 & & \mathsf{min}(\tau_1, \tau_2) \mid \mathsf{Size}(\tau) \mid \mathsf{Index}(\tau) \mid \mathsf{Ref}(\tau) \mid \\
 & & (\overrightarrow{\tau} \xrightarrow{AS_1} \tau) \mid \mathsf{Boolean} \mid \mathsf{Integer} \\
\tau \in \text{Type} & ::= & \tau \mid \mathsf{Channel}(\pi, \tau, \tau_0) \mid \mathsf{Channel}(\pi, \tau, \tau_0)[\tau_1] \\
b \in \text{Channel Delay Flag} & ::= & 1 \mid 0 \\
\pi \in \text{Polarity} & ::= & + \mid - \mid \pm \\
\alpha \in \text{Event} & ::= & t! \mid t[\tau]! \mid t? \mid t[\tau]? \\
\mathscr{I} \in \text{Type Iterator} & ::= & (t \leftarrow \tau_1..\tau_2) \\
\mathscr{G} \in \text{Type Guard} & ::= & (\tau_1\ \rho\ \tau_2) \\
\rho \in \text{Rel Op} & ::= & \ldots \\
AS \in \text{Actor Flowstate} & ::= & \varepsilon \mid \{\alpha \mid \overrightarrow{\mathscr{I}}, \overrightarrow{\mathscr{G}}\} \mid (AS_1; AS_2) \\
\Gamma \in \text{Type Env} & ::= & \varepsilon \mid \Gamma, t : \kappa \\
\Delta \in \text{Value Env} & ::= & \varepsilon \mid \Delta, x : \tau \mid \Delta, c : \tau \\
FS \in \text{Proc Flowstate} & ::= & \varepsilon \mid AS \mid \{AS \mid t \leftarrow \tau_1..\tau_2\} \mid (FS_1 \parallel FS_2) \\
NS \in \text{Network sig} & ::= & \mathsf{Network}(\Gamma, \Delta \rhd FS)
\end{array}
$$

Figure 1: Abstract syntax of $\mathbf{S_{DATA}}$ types

The flowstate for this loop is a sequential composition of two comprehensions

$$\{\mathtt{i}[\mathtt{t}]? \mid \mathtt{t} \leftarrow \underline{1}..\mathtt{s}\}; (\mathtt{s}/\underline{2}) * \mathtt{o}!.$$

In typing the code above, `t` is a type-level witness for the loop index `x`. This type witness has the kind $\mathsf{Size}(\mathtt{s})$, reflecting that it bounded by the size parameter `s`, while the loop index has the type $\mathsf{Index}(\mathtt{t})$. The distinction at the type level between size and index variables, $\mathsf{Size}(\mathtt{s})$ and $\mathsf{Index}(\mathtt{t})$ respectively in the example above, is crucial to the static analysis of buffer sizes and actor firing rates: sizes are fixed for the execution of a dataflow graph, while indexes obviously vary dynamically.

We consider a type system for a simple dataflow language with variable firing rates in Sect. 2. We provide an operational semantics in Sect. 3. Sect. 4 considers related work while Sect. 5 provides our conclusions.

2 Dataflow Language

In this section we consider a core language to prescribe dataflow computations. We name this kernel language $\mathbf{S_{DATA}}$. We describe a type system and an operational semantics for this language. Our "object language" uses a form of session types for dataflow, that we refer to as *sessional dataflow* to express in the type system the contracts between producers and consumers who share message-passing channels.

The syntax of types is provided in Fig. 1. We assume Boolean and integer types for base types, although other types (e.g., floating point) could obviously be easily added.

In order to track communication rates, the type system includes type-level names $\underline{n}$ for size constants n, of the form $\underline{0}$, $\underline{1}$, $\underline{2}$, ... A type constant $\underline{n}$ has the kind $\mathsf{Size}(\underline{n})$. A type parameter will have a kind of the form $\mathsf{Size}(\tau)$, for some type-level upper bound τ. In general, the kind of a type-level numeric quantity records an upper bound on the possible instantiations of a type parameter of that kind, and a subkinding system allows this bound to be inflated, losing precision in the kinds of type quantities. The special constant $\underline{\infty}$ represents the absence of an upper bound (the equivalent of $\top$ in a subtyping system).

We have two types for tracking numeric quantities at the value level. $\mathsf{Size}(\tau)$ represents a size parameter, fixed over the execution of an actor. Typically it is used to parameterize over communication rates, or fan-in or fan-out at an actor. $\mathsf{Index}(\tau)$ is the type-level representative for a loop index, which obviously does vary at execution time. Both types are indexed by a type-level numeric quantity of kind $\mathsf{Size}(\tau)$. Our main reason for distinguishing these two types is to prevent a loop index being used as the bound for another loop, which would be useless for practical applications while complicating the analysis. This distinction between static and dynamic numeric quantities simplifies the extraction of actor firing rates from behavioral types.

The language includes type-level names for channels and channel arrays. These are represented by type variables t with kinds of the form $\mathsf{Channel}(b, \tau_{limit})$ and $\mathsf{Channel}(b, \tau_{limit})[\tau]$, and are used to index the types of values that are tracked by the type system. So we have a channel type $\mathsf{Channel}(\pi, t, \tau)$, where t is the type-level name for the channel (of kind $\mathsf{Channel}(b, \tau_{limit})$), and τ the type of message payloads that can be exchanged. The polarity π allows sending or receiving on a channel. A single actor can only send or receive, but not both, on a channel. In a network, these uniplex channels in actor signatures are instantiated with shared duplex channels that connect different actors. In the channel kind, the type parameter τ_{limit} represents a type-level bound on the number of messages that can be buffered in the channel. The flag b indicates if messages that should be buffered in the channel at beginning of execution of the dataflow graph, to remove a cycle in the firing schedule by introducing a delay. If messages are buffered, the number of messages to be buffered is given by the channel capacity, τ_{limit}.

Channel array types have the form $\mathsf{Channel}(\pi, t, \tau)[\tau_1]$, where t is the type-level name for the channel array (of kind $\mathsf{Channel}(b, \tau_{limit})[\tau_1]$), τ the type of message payloads that can be exchanged, and τ_1 a bound on the size of the array.

The language includes procedure types of the form $(\overrightarrow{\tau}) \xrightarrow{AS_1} \tau$. A procedure takes a sequence of value arguments, of type $\overrightarrow{\tau}$, and produces a result of type τ. In addition, the procedure has a latent effect, reflected by an actor flowstate AS_1 that records the communications performed during the execution of this procedure. For mutable variables, the language includes references, which can be considered as one-element arrays. These could be straightforwardly generalized to n-element arrays, but we use references for simplicity in the presentation.

An event has one of four possible forms, two event forms for sending events and two forms for receiving events. For sending, the three forms are $c!$ (sending on a channel) and $c[\tau]!$ (sending on an element of a channel array, where τ is the type-level representative for the index). There are analogous event types for message receipt: $c?$ and $c[\tau]?$.

A flowstate for an actor AS is a composition of events. In its most general form, an event in a flowstate is described by an *event comprehension* of the form

$$\{\alpha \mid \overrightarrow{t \leftarrow \tau_1..\tau_2}, \overrightarrow{\tau' \rho \tau''}\}$$

The iterators $(t \leftarrow \tau_1..\tau_2)$ record loops in which the event occurred, while the guards $(\tau' \rho \tau')$ record conditions on the occurrence of the events. A guard denotes a conditional communication, and is useful for applications such as decimation, where an actor discards some of its input (e.g., in a downsampler). We admit specific forms of this general description of an event:

1. A singleton event α, that may be a communication on a channel or a channel array element.
2. Iterated and conditional communication over a channel $\{\alpha \mid \overrightarrow{\mathscr{I}}, \overrightarrow{\mathscr{G}}\}$, where α has the form $c!$ or $c?$ for some channel c. The goal of the analysis is to reduce this to a single multiplicity τ for the communication, folding guards into iterators by modifying the bounds, and combining the

iterators. We sometimes denote $\{\alpha \mid t \leftarrow \underline{1}..\tau\}$, where the single iterator variable t does not occur in α, by $\tau * \alpha$. We sometimes use α as shorthand for the flowstate $\underline{1} * \alpha$.

3. Communication over the elements of a channel array, described by the comprehension $\{\alpha \mid t \leftarrow \tau_1..\tau_2, \overrightarrow{\mathscr{I}}\}$, where α has the form $c[t]!$ or $c[t]?$ for some channel array c of size τ. The iterators $\overrightarrow{\mathscr{I}}$ contribute additional multiplicities. Since we can combine the additional iterators, if any, into a single iterator, we sometimes denote $\{\alpha \mid t \leftarrow \tau_1..\tau_2, t_0 \leftarrow \tau_0..\tau_0'\}$ by $(\tau_0' - \tau_0 + \underline{1}) * \{\alpha \mid t \leftarrow \tau_1..\tau_2\}$, where t_0 does not occur free (as a channel array index) in the event α.

We must place sufficient restrictions on a guard to ensure that it can be easily folded into a loop bound. For this article, we restrict guard types to be one of the following forms:

$$\mathscr{G} \quad ::= \quad (\tau \mid t) \quad | \quad (t \leq \tau)$$

The first denotes a predicate asserting that the quantity τ divides the loop index t, while the latter asserts an upper bound on t. Then we allow the following equivalences on flowstates, that fold these conditions into iterators:

$$\{\alpha \mid \overrightarrow{\mathscr{I}}, (t \leftarrow \tau_1..\tau_2), (\tau \mid t), \overrightarrow{\mathscr{G}}\} \equiv \{\alpha \mid \overrightarrow{\mathscr{I}}, (t \leftarrow \underline{1}..((\tau_2 - \tau_1 + \underline{1})/\tau), \overrightarrow{\mathscr{G}}\}$$

$$\{\alpha \mid \overrightarrow{\mathscr{I}}, (t \leftarrow \tau_1..\tau_2), (t \leq \tau), \overrightarrow{\mathscr{G}}\} \equiv \{\alpha \mid \overrightarrow{\mathscr{I}}, (t \leftarrow \tau_1..\mathsf{min}(\tau, \tau_2)), \overrightarrow{\mathscr{G}}\}$$

We only allow conditions on communication in the case where communication is on a channel rather than a channel array, and in this case the actual range of values of the iteration variable is not important, since we are only counting number of occurrences of the communication event in a firing.

A dataflow network has a *network signature NS*, which has three parts:

1. A type environment Γ that binds type-level representatives for channels, channel arrays and sizes.
2. A value environment Δ that captures information about the shared communication channels, using bindings of the form $(c : \tau)$, as well as size parameters for the network description.
3. The *flowstate* of a network FS records its expected firing behavior. This is described by the parallel composition of the flowstates of the actors in the network.

Fig. 2 provides the abstract syntax for programs in $\mathbf{S_{DATA}}$. Values include Booleans (true and false) and integers n. Atomic values also include names c (for channels and breakpoints), and variables x. Our language is a basic expressional language, with functions and call-by-value evaluation. An abstraction, of the form $\lambda \overrightarrow{x}{:}\overrightarrow{\tau}.AS_1 \Rightarrow E$, abstracts over simple value parameters. For now we disallow value-level abstraction over type parameters, such as those for singleton types for channels, as well as numeric quantities, in order to avoid aliasing issues. We allow for abstraction over such parameters in the network graph as a whole, and the instantiation of the network ensures that no aliases are introduced. The latent flowstate AS_1 for the procedure body E (i.e., the communications it offers) is provided as an annotation. As with the function type, this records the communication performed in the function AS_1. An application $E(\overrightarrow{E})$ denotes the application of a procedure to value level parameters $\overrightarrow{E}$. A let construct, which can be read as a combination of abstraction and application, binds a variable in a local context.

A conditional allows dispatching on a Boolean value. The type rules require that both branches in the conditional have identical flowstates. To record conditional communication in an actor, the when construct relates Boolean conditions to communication events in the flowstate.

The finite loop construct binds two local parameters: x, the index variable for the loop, and t, a type-level parameter for the loop index. The latter, in combination with channel array references in

$V \in \text{Values}$	$::=$	$\mathsf{true} \mid \mathsf{false} \mid n \mid c \mid x \mid \mathsf{size}(V)$	
	$\mid$	$\mathsf{index}(V) \mid \lambda \overrightarrow{x} : \overrightarrow{\tau}.AS_1 \Rightarrow E$	
$E \in \text{Expr}$	$::=$	$\lambda \overrightarrow{x} : \overrightarrow{\tau}.AS_1 \Rightarrow E$	Abstraction
	$\mid$	$E(\overrightarrow{E})$	Application
	$\mid$	$\mathsf{let}\ x = E_1\ \mathsf{in}\ E_2$	Bind variable
	$\mid$	$\mathsf{if}\ E\ \mathsf{then}\ E_1\ \mathsf{else}\ E_2$	Conditional
	$\mid$	$\mathsf{when}\ E_1\ \mathsf{do}\ E_2$	Cond Comm
	$\mid$	$\mathsf{for}\ (t, x \in n..E)\ E_2$	Finite Loop
	$\mid$	$\mathsf{ref}(E)$	New reference
	$\mid$	$*E$	Dereference
	$\mid$	$E_1 := E_2$	Assignment
	$\mid$	$\mathsf{size}(E)$	Size constant
	$\mid$	$\mathsf{fromSize}(E)$	Size projection
	$\mid$	$\mathsf{index}(E)$	Loop index
	$\mid$	$\mathsf{fromIndex}(E)$	Index projection
	$\mid$	$c?,\ c[E_0]?$	Receive a message
	$\mid$	$c!E,\ c[E_0]!E$	Send a message
$P \in \text{Proc}$	$::=$	$\mathsf{stop} \mid E \mid \{E \mid t, x \leftarrow V_1..V_2\} \mid (P_1 \parallel P_2)$	
$N \in \text{Network}$	$::=$	$\mathsf{network}(\Gamma, \Delta \vdash FS : P)$	

Figure 2: Abstract syntax of $\mathbf{S_{DATA}}$ expressions and processes

communication events, is used to record communication behavior in a loop. A parameter of size type is created by the constructor $\mathsf{size}(n) \in \mathsf{Size}(\underline{n})$, and is deconstructed using the accessor $\mathsf{fromSize}(E)$, providing access to the underlying integer bound. Similar operations are available for constructing loop index values $\mathsf{index}(n) \in \mathsf{Index}(\underline{n})$ and projecting the loop index out of this value, $\mathsf{fromIndex}(E)$.

There are two operations for receiving messages, receiving on a channel or on a channel array element, and similarly two operations for sending messages. It is instructive that the channel reference is always a name and never a variable. In this account, we are not yet considering a facility for transmitting the ability to send or receive on a channel, as is found in varying degrees in the pi-calculus. The reason is again to avoid issues with channel aliasing, which would subvert flowstate checking on usage of channels.

The type system is formulated using judgements of the following forms:

$\vdash \Gamma\ \mathsf{ok}$	Type environment
$\Gamma \vdash \Delta\ \mathsf{ok}$	Value environment
$\Gamma \vdash \kappa\ \mathsf{ok}$	Kind
$\Gamma \vdash \tau : \kappa$	Type
$\Gamma \vdash \alpha\ \mathsf{ok}$	Event
$\Gamma \vdash AS : \text{Flowstate}$	Flowstate
$\Gamma; \Delta \vdash E : \tau :: AS$	Expression
$\Gamma; \Delta \vdash P : FS$	Process

The type rules for environments, kinds and types are provided in Fig. 3. A "size kind" $\text{Size}(\tau)$ is indexed by an upper bound on numeric types of that kind. This in turn gives rise to a subkinding relationship $\Gamma \vdash \tau_1 \leq \tau_1$ (that can be read as a synonym for $\Gamma \vdash \tau_1 : \text{Size}(\tau_2)$), formalized in Fig. 3. Note that whereas we allow subsumption on size kinds, we do not allow subtyping on size types (of the form $\mathsf{Size}(\tau)$ or $\mathsf{Index}(\tau)$, for some witness of size kind $\text{Size}(\_)$), because size estimates in types, and

$$\frac{}{\vdash \varepsilon\ \mathsf{ok}}\ \textsc{TyEnv Empty} \qquad \frac{\vdash \Gamma\ \mathsf{ok} \quad \Gamma \vdash \kappa\ \mathsf{ok}}{\vdash \Gamma, t : \kappa\ \mathsf{ok}}\ \textsc{TyEnv Extend}$$

$$\frac{\Gamma \vdash \tau :\ \mathsf{Size}(\_)}{\Gamma \vdash\ \mathsf{Size}(\tau)\ \mathsf{ok}}\ \textsc{Kind Size}$$

$$\frac{\vdash \Gamma\ \mathsf{ok} \quad \Gamma \vdash \tau_{limit} :\ \mathsf{Size}(\_) \quad \Gamma \vdash \tau_{init} :\ \mathsf{Size}(\tau_{limit})}{\Gamma \vdash\ \mathsf{Channel}(b, \tau_{limit})\ \mathsf{ok}}\ \textsc{Kind Chan}$$

$$\frac{\Gamma \vdash \tau :\ \mathsf{Size}(\_) \quad \Gamma \vdash \tau_{limit} :\ \mathsf{Size}(\_) \quad \Gamma \vdash \tau_{init} :\ \mathsf{Size}(\tau_{limit})}{\Gamma \vdash\ \mathsf{Channel}(b, \tau_{limit})[\tau]\ \mathsf{ok}}\ \textsc{Kind Chan Array}$$

$$\frac{\Gamma \vdash \tau :\ \mathsf{Size}(\_)}{\Gamma \vdash \tau \leq \tau}\ \textsc{Size Refl} \qquad \frac{\Gamma \vdash \tau_1 \leq \tau_2 \quad \Gamma \vdash \tau_2 \leq \tau_3}{\Gamma \vdash \tau_1 \leq \tau_3}\ \textsc{Size Trans}$$

$$\frac{\Gamma \vdash \tau :\ \mathsf{Size}(\_)}{\Gamma \vdash \tau \leq \underline{\infty}}\ \textsc{Size Infty} \qquad \frac{\Gamma \vdash \tau :\ \mathsf{Size}(\tau_0)}{\Gamma \vdash \tau \leq \tau_0}\ \textsc{Size Bound} \qquad \frac{\vdash \Gamma\ \mathsf{ok} \quad m \leq n}{\Gamma \vdash \underline{m} \leq \underline{n}}\ \textsc{Size Num}$$

$$\frac{\vdash \Gamma\ \mathsf{ok}}{\Gamma \vdash \underline{n} :\ \mathsf{Size}(\underline{n})}\ \textsc{Ty Size} \qquad \frac{\vdash \Gamma\ \mathsf{ok} \quad (t : \kappa) \in \Gamma}{\Gamma \vdash t : \kappa}\ \textsc{Ty Var} \qquad \frac{\vdash \Gamma\ \mathsf{ok}}{\Gamma \vdash \underline{\infty} :\ \mathsf{Size}(\underline{\infty})}\ \textsc{Ty Infty}$$

$$\frac{\Gamma \vdash \tau_1 :\ \mathsf{Channel}(b, \tau_{limit}) \quad \Gamma \vdash \tau_2 :\ \mathsf{Type}}{\Gamma \vdash \mathsf{Channel}(\pi, \tau_1, \tau_2) :\ \mathsf{Type}}\ \textsc{Ty Chan}$$

$$\frac{\Gamma \vdash \tau_1 :\ \mathsf{Channel}(b, \tau_{limit})[\tau_3] \quad \Gamma \vdash \tau_2 :\ \mathsf{Type}}{\Gamma \vdash \mathsf{Channel}(\pi, \tau_1, \tau_2)[\tau_3] :\ \mathsf{Type}}\ \textsc{Ty Chan Array}$$

Figure 3: Type Environments, Kinds and Types

$$\frac{\Gamma \vdash \tau :\ \mathsf{Channel}(b, \tau_{limit})}{\Gamma \vdash \tau!\ \mathsf{ok}}\ \textsc{FS Send} \qquad \frac{\Gamma \vdash \tau :\ \mathsf{Channel}(b, \tau_{limit})}{\Gamma \vdash \tau?\ \mathsf{ok}}\ \textsc{FS Recv}$$

$$\frac{\Gamma \vdash \tau :\ \mathsf{Channel}(b, \tau_{limit})[\tau_0] \quad \Gamma \vdash \tau_1 \leq \tau_0}{\Gamma \vdash \tau[\tau_1]!\ \mathsf{ok}}\ \textsc{FS Array Send}$$

$$\frac{\Gamma \vdash \tau :\ \mathsf{Channel}(b, \tau_{limit})[\tau_0] \quad \Gamma \vdash \tau_1 \leq \tau_0}{\Gamma \vdash \tau[\tau_1]?\ \mathsf{ok}}\ \textsc{FS Array Recv}$$

$$\frac{\Gamma \vdash \tau_1 :\ \mathsf{Size}(\_) \quad \Gamma \vdash \tau_1 :\ \mathsf{Size}(\_)}{\Gamma \vdash t \leftarrow \tau_1..\tau_2\ \mathsf{ok}}\ \textsc{FS Iter} \qquad \frac{\Gamma \vdash \tau_1 :\ \mathsf{Size}(\_) \quad \Gamma \vdash \tau_1 :\ \mathsf{Size}(\_)}{\Gamma \vdash (\tau_1 \mid \tau_2)\ \mathsf{ok}}\ \textsc{FS Gd Div}$$

$$\frac{\Gamma \vdash \tau_1 :\ \mathsf{Size}(\_) \quad \Gamma \vdash \tau_1 :\ \mathsf{Size}(\_)}{\Gamma \vdash (\tau_1 \leq \tau_2)\ \mathsf{ok}}\ \textsc{FS Gd Bnd}$$

$$\frac{\overrightarrow{\mathscr{I}} = \overrightarrow{t \leftarrow \tau_1..\tau_2} \quad \Gamma, \overrightarrow{t} :\ \overrightarrow{\mathsf{Size}(\tau_2)} \vdash \alpha\ \mathsf{ok} \quad \overrightarrow{\Gamma \vdash \mathscr{I}\ \mathsf{ok}} \quad \overrightarrow{\Gamma \vdash \mathscr{G}\ \mathsf{ok}}}{\Gamma \vdash \{\alpha \mid \overrightarrow{\mathscr{I}}, \overrightarrow{\mathscr{G}}\} :\ \mathsf{Flowstate}}\ \textsc{FS Comp}$$

$$\frac{\vdash \Gamma\ \mathsf{ok}}{\Gamma \vdash \varepsilon :\ \mathsf{Flowstate}}\ \textsc{FS Empty} \qquad \frac{\Gamma \vdash AS_1 :\ \mathsf{Flowstate} \quad \Gamma \vdash AS_2 :\ \mathsf{Flowstate}}{\Gamma \vdash AS_1; AS_2 :\ \mathsf{Flowstate}}\ \textsc{FS Seq}$$

Figure 4: Flowstates

particularly in flowstates, are required to be precise.

Fig. 4 provides formation rules for flowstates, while Fig. 5 provides type rules for value environments and values.

Fig. 6 and Fig. 7 provide type rules for value-level expressions in the language. The main type rules for dataflow computation are provided in Fig. 7. The for construct is key, since it allows communication within a loop, as represented by the flowstate AS_1 for each iteration. The loop bound has a type-level witness type τ_1, so the flowstate for the entire loop is $(\{AS_1 \mid t \leftarrow \underline{1}..\tau\})$. Within the loop, the index variable has type $\mathsf{Index}(t)$, where t is an index type parameter of kind $\mathsf{Size}(\tau_1)$, bounded above by the loop bound. AS_2 is the flowstate for the remaining computation after the loop. Within the loop, thie remaining flowstate after each iteration represented by the flowstate expression $(\{\{t_0/t\}AS_1 \mid t_0 \leftarrow$

$$\frac{\vdash \Gamma \text{ ok}}{\Gamma \vdash \varepsilon \text{ ok}} \text{ VALENV EMPTY} \qquad \frac{\Gamma \vdash \Delta \text{ ok} \quad \Gamma \vdash \tau : \text{Type}}{\Gamma \vdash \Delta, x : \tau \text{ ok}} \text{ VALENV EXTEND VAR}$$

$$\frac{\Gamma \vdash \Delta \text{ ok} \quad \Gamma \vdash \tau : \text{Type}}{\Gamma \vdash \Delta, c : \tau \text{ ok}} \text{ VALENV EXTEND NAME} \qquad \frac{\Gamma \vdash \Delta \text{ ok} \quad \Gamma \vdash AS : \text{Flowstate}}{\Gamma ; \Delta \vdash n : \text{Integer}::\varepsilon} \text{ VAL INT}$$

$$\frac{\Gamma \vdash \Delta \text{ ok} \quad \Gamma \vdash AS : \text{Flowstate}}{\Gamma ; \Delta \vdash \text{true} : \text{Boolean}::\varepsilon} \text{ VAL TRUE} \qquad \frac{\Gamma \vdash \Delta \text{ ok} \quad \Gamma \vdash AS : \text{Flowstate}}{\Gamma ; \Delta \vdash \text{false} : \text{Boolean}::\varepsilon} \text{ VAL FALSE}$$

$$\frac{\Gamma \vdash \Delta \text{ ok} \quad (x : \tau) \in \Delta \quad \Gamma \vdash AS : \text{Flowstate}}{\Gamma ; \Delta \vdash x : \tau::\varepsilon} \text{ VAL VAR}$$

Figure 5: Value Environments and Values

$$\frac{\Gamma ; \Delta \vdash E_1 : \tau_1::AS_1 \quad \Gamma ; (\Delta, x : \tau_1) \vdash E_2 : \tau_2::AS_2}{\Gamma ; \Delta \vdash (\text{let } x = E_1 \text{ in } E_2) : \tau_2::(AS_1 ; AS_2)} \text{ VAL LET}$$

$$\frac{\Gamma ; \Delta, \overrightarrow{x} : \overrightarrow{\tau_1} \vdash E : \tau_2::AS_1}{\Gamma ; \Delta \vdash (\lambda \overrightarrow{x} : \overrightarrow{\tau_1}.AS_1 \Rightarrow E) : (\overrightarrow{\tau_1} \xrightarrow{AS_1} \tau_2)::\varepsilon} \text{ VAL ABS}$$

$$\frac{\Gamma ; \Delta \vdash E_0 : ((\tau_1, \ldots, \tau_k) \xrightarrow{AS} \tau)::AS_0 \quad \Gamma ; \Delta \vdash E_i : \tau_i::AS_i \text{ for } i = 1, \ldots, k}{\Gamma ; \Delta \vdash E(E_1, \ldots, E_k) : \tau::(AS_0 ; AS_1 ; \ldots ; AS_k ; AS)} \text{ VAL APP}$$

$$\frac{\Gamma ; \Delta \vdash E_0 : \text{Boolean}::AS_0 \quad \Gamma ; \Delta \vdash E_1 : \tau::AS \quad \Gamma ; \Delta \vdash E_2 : \tau::AS}{\Gamma ; \Delta \vdash (\text{if } E_0 \text{ then } E_1 \text{ else } E_2) : \tau::(AS_0 ; AS)} \text{ VAL COND}$$

$$\frac{\Gamma ; \Delta \vdash E : \tau::AS_0}{\Gamma ; \Delta \vdash \text{ref}(E) : \text{Ref}(\tau)::AS_0} \text{ VAL REF}$$

$$\frac{\Gamma ; \Delta \vdash E : \text{Ref}(\tau)::AS_0}{\Gamma ; \Delta \vdash *E : \tau::AS_0} \text{ VAL DEREF}$$

$$\frac{\Gamma ; \Delta \vdash E_1 : \text{Ref}(\tau)::(AS_1 \quad \Gamma ; \Delta \vdash E_2 : \tau::AS_2}{\Gamma ; \Delta \vdash E_1 := E_2 : \tau::(AS_1 ; AS_2)} \text{ VAL ASSIGN}$$

$$\frac{\Gamma ; \Delta \vdash E : \tau::AS_1' \quad \Gamma \vdash AS_1 : \text{Flowstate} \quad \Gamma \vdash AS_2 : \text{Flowstate} \quad AS_1 \equiv AS_1' \quad AS_2 \equiv AS_2'}{\Gamma ; \Delta \vdash E : \tau::AS_1} \text{ VAL EQ}$$

Figure 6: $\mathbf{S_{DATA}}$: Core Expressions

$(t + \underline{1})..\tau\})$, where t_0 is a new variable introduced to count the remaining iterations.

The expression $\{AS \mid t \leftarrow \tau_1..\tau_2\}$ is a metafunction, generalizing the event comprehension introduced earlier from events to flowstates, defined by:

$$\begin{aligned}
\{\varepsilon \mid t \leftarrow \tau_1..\tau_2\} &= \varepsilon \\
\{AS_1 ; AS_2 \mid t \leftarrow \tau_1..\tau_2\} &= \{AS_1 \mid t \leftarrow \tau_1..\tau_2\} ; \{AS_2 \mid t \leftarrow \tau_1..\tau_2\} \\
\{\{\alpha \mid \overrightarrow{\mathscr{I}}, \overrightarrow{\mathscr{G}}\} \mid t \leftarrow \tau_1..\tau_2\} &= \{\alpha \mid (\overrightarrow{\mathscr{I}}, t \leftarrow \tau_1..\tau_2), \overrightarrow{\mathscr{G}}\}
\end{aligned}$$

There is an important assumption in the second case of this definition, where we distribute a flowstate comprehension over the joining of two flowstates. The assumption is that we are not tracking causality

$$\frac{\begin{array}{c}\Gamma;\Delta \vdash E_1 : \mathsf{Size}(\tau_1)::AS_0 \\ \Gamma, t: \mathsf{Size}(\tau_1);\Delta, x:\mathsf{Index}(t) \vdash E_2 : \tau_2::(AS_1)\end{array}}{\Gamma;\Delta \vdash (\mathsf{for}\ (t, x \in 1..E_1)\ E_2) : \tau::((AS_0;\{AS_1 \mid t \leftarrow \underline{1}..\tau\}))} \text{ VAL FOR}$$

$$\frac{\begin{array}{c}\Gamma;\Delta \vdash E_1 : \tau_1::AS_1 \\ \Gamma;\Delta \vdash E_2 : \tau_2::AS_2 \\ \mathit{wfguard}(\tau_1\rho\tau_2) \quad \Gamma;\Delta \vdash E : \tau::(AS_3)\end{array}}{\Gamma;\Delta \vdash (\mathsf{when}\ E_1\rho E_2\ \mathsf{do}\ E_3) : \tau::((AS_1;AS_2;\{AS_3 \mid \tau_1\rho\tau_2\}))} \text{ VAL WHEN}$$

$$\frac{\Gamma \vdash \Delta\ \mathsf{ok} \quad \Gamma \vdash AS : \mathsf{Flowstate}}{\Gamma;\Delta \vdash \mathsf{size}(n) : \mathsf{Size}(\underline{n})::\varepsilon} \text{ VAL SIZE}$$

$$\frac{\Gamma;\Delta \vdash E : \mathsf{Size}(\tau)::AS_1}{\Gamma;\Delta \vdash \mathsf{fromSize}(E) : \mathsf{Integer}::AS_1} \text{ VAL INT}$$

$$\frac{\Gamma \vdash \Delta\ \mathsf{ok} \quad \Gamma \vdash AS : \mathsf{Flowstate}}{\Gamma;\Delta \vdash \mathsf{index}(n) : \mathsf{Index}(\underline{n})::\varepsilon} \text{ VAL INDEX}$$

$$\frac{\Gamma;\Delta \vdash E : \mathsf{Index}(\tau)::AS_1}{\Gamma;\Delta \vdash \mathsf{fromIndex}(E) : \mathsf{Integer}::AS_1} \text{ VAL FROMINDEX}$$

$$\frac{\Gamma;\Delta \vdash E : \tau::AS_1 \quad (c : \mathsf{Channel}(\pi, \tau_0, \tau)) \in \Delta \quad \pi \in \{-, \pm\}}{\Gamma;\Delta \vdash c!E : \tau::(AS_1;\tau_0!)} \text{ VAL SEND}$$

$$\frac{\Gamma \vdash \Delta\ \mathsf{ok} \quad (c : \mathsf{Channel}(\pi, \tau_0, \tau)) \in \Delta \quad \Gamma \vdash AS : \mathsf{Flowstate} \quad \pi \in \{+, \pm\}}{\Gamma;\Delta \vdash c? : \tau::\tau_0?} \text{ VAL RECEIVE}$$

$$\frac{\begin{array}{c}(c : \mathsf{Channel}(\pi, \tau_1, \tau)[\tau_0']) \in \Delta \quad \pi \in \{-, \pm\} \\ \Gamma;\Delta \vdash E_0 : \mathsf{Index}(\tau_0)::AS_0 \\ \Gamma;\Delta \vdash E_1 : \tau_1::AS_1 \quad \Gamma \vdash \tau_0 \leq \tau_0'\end{array}}{\Gamma;\Delta \vdash c[E_0]!E : \tau::(AS_0;AS_1;\tau_1[\tau_0]!)} \text{ VAL SEND ARRAY}$$

$$\frac{\begin{array}{c}(c : \mathsf{Channel}(\pi, \tau_1, \tau)[\tau_0']) \in \Delta \quad \pi \in \{+, \pm\} \\ \Gamma;\Delta \vdash E_0 : \mathsf{Index}(\tau_0)::AS_0 \quad \Gamma \vdash \tau_0 \leq \tau_0'\end{array}}{\Gamma;\Delta \vdash c[E_0]? : \tau::(AS_0;\tau_0[\tau_1]?)} \text{ VAL RECV ARRAY}$$

Figure 7: $\mathbf{S_{DATA}}$: Dataflow Expressions

within an actor, so we are free to reorder communications within that actor. For example a loop that inputs on one channel c_1 and outputs on another c_2 would have the flowstate $\{c_1?;c_2! \mid t \leftarrow \underline{1}..\underline{n}\}$, which normalizes to $(\{c_1? \mid t \leftarrow \underline{1}..\underline{n}\};\{c_2! \mid t \leftarrow \underline{1}..\underline{n}\})$, which we abbreviate as $(\underline{n}*c_1?;\underline{n}*c_2!)$. The normalized form loses the causality between a receive and send on a single loop iteration. We rely on a global check of the composition of the actors to detect any causal cycles in the firing of a dataflow graph, relying on the fact that we do not have abstraction over the structure of the dataflow graph (beyond the eliding of internal causality within an actor). A global check stratifies the actors based on communication dependencies, and ensures there are no cycles where an actor's inputs depend on its own outputs (unless there is a delay

$$\frac{}{\Gamma;\Delta \vdash \mathsf{stop} : \varepsilon}\ \textsc{Proc Empty}$$

$$\frac{\Gamma;\Delta \vdash E : \tau{::}AS}{\Gamma;\Delta \vdash E : AS}\ \textsc{Proc Exp}$$

$$\frac{\Gamma;\Delta \vdash V : \mathsf{Size}(\tau) \quad \Gamma, t : \mathsf{Size}(\tau);\Delta, x : \mathsf{Index}(t) \vdash E : \tau{::}AS}{\Gamma;\Delta \vdash \{E \mid t, x \leftarrow n..V\} : \{AS \mid t \leftarrow \underline{n}..\tau\}}\ \textsc{Proc Comp}$$

$$\frac{\Gamma;\Delta \vdash P_1 : FS_1 \quad \Gamma;\Delta \vdash P_2 : FS_2}{\Gamma;\Delta \vdash (P_1 \parallel P_2) : FS_1 \parallel FS_2}\ \textsc{Proc Par}$$

Figure 8: $\mathbf{S_{DATA}}$: Processes

in the channel).

The constructs for sizes, that are used to track capacity bounds and communication rates, allow an integer literal to be wrapped as a value $\mathsf{size}(n)$ whose type $\mathsf{Size}(\underline{n})$ reflects the integer quantity. The destructor $\mathsf{fromSize}(E)$ allows this size parameter to be projected to an integer, with $\mathsf{fromSize}(\mathsf{size}(n))$ evaluating to n. Similar constructs are available for dynamic numeric quantities (loop indices).

There are four communication primitives: two for sending and two for receiving. Each sending and receiving operation has a variant for communicating on a single channel or communicating on an element of a channel array. Each of these primitives gives rise to one of the four communication events tracked by the flowstate: $c!$, $c?$, $c[\tau]!$ and $c[\tau]?$. For example, a loop that outputs one value on each element of a channel array would have the form:

$$\mathsf{for}\ (t, x \in 1..E_1)\ c[x]!E_2$$

Assume E_1 has type $\mathsf{Size}(\tau)$, where τ is some type-level size of kind $\mathsf{Size}(\_)$, e.g., $\underline{n}$ of kind $\mathsf{Size}(\underline{n})$. The loop variable has type $\mathsf{Index}(t)$, where the loop type variable t has kind $\mathsf{Size}(\tau)$, reflecting the type-level bound on the number of iterations. Each iteration of the loop has a flowstate $c[t]!$, and the entire loop has the flowstate $\{c[t]! \mid t \leftarrow \underline{1}..\tau\}$.

Conditional communication is performed using the when construct. This tests a condition and adds a guard to the flowstate for the body of the when. The condition is restricted so that it can only be used to refine static bounds, and not introduce a dependency in computation bounds on dynamic quantities (such as a loop index). The well-formedness condition $\mathit{wfguard}(\tau_1 \rho \tau_2)$ on the types of the two values being compared enforces this restriction, where τ_1 and τ_2 are the types of the expressions being compared, and ρ is the relational operator:

$$\begin{array}{rcl} \mathit{wfguard}(\tau_1 \mid \tau_2) & \iff & \tau_1 = \mathsf{Size}(\_) \text{ and } \tau_1 = \mathsf{Index}(\_) \\ \mathit{wfguard}(\tau_1 \leq \tau_2) & \iff & \tau_1 = \mathsf{Index}(\_) \text{ and } \tau_1 = \mathsf{Size}(\_) \end{array}$$

As with iterators, we define a metafunction that distributes guards over flowstates. The expression $\{AS \mid \tau_1 \rho \tau_2\}$ is a metafunction, generalizing the event comprehension introduced earlier from events to flowstates, defined by:

$$\begin{array}{rcl} \{\varepsilon \mid \tau_1 \rho \tau_2\} & = & \varepsilon \\ \{AS_1; AS_2 \mid \tau_1 \rho \tau_2\} & = & \{AS_1 \mid \tau_1 \rho \tau_2\}; \{AS_2 \mid \tau_1 \rho \tau_2\} \\ \{\{\alpha \mid \overrightarrow{\mathscr{I}}, \overrightarrow{\mathscr{G}}\} \mid \tau_1 \rho \tau_2\} & = & \{\alpha \mid \overrightarrow{\mathscr{I}}, (\overrightarrow{\mathscr{G}}, \tau_1 \rho \tau_2)\} \end{array}$$

$$\frac{}{\Gamma \vdash (\varepsilon, FS) \longrightarrow (\varepsilon, FS)} \text{ FS PROG EMPTY}$$

$$\frac{\Gamma \vdash (FS_1, FS_3) \longrightarrow (FS_1', FS_3')}{\Gamma \vdash ((FS_1 \parallel FS_2), FS_3) \longrightarrow ((FS_1' \parallel FS_2), FS_3')} \text{ FS PROG PAR}$$

$$\frac{Producer(\Gamma, \alpha)}{\Gamma \vdash ((\{\alpha \mid \overrightarrow{\mathscr{I}}, \overrightarrow{\mathscr{G}}\}; AS), FS) \longrightarrow (AS, (\{\alpha \mid \overrightarrow{\mathscr{I}}, \overrightarrow{\mathscr{G}}\} \parallel FS))} \text{ FS PROG PROD}$$

$$\frac{Consumer(\Gamma, \alpha)}{\Gamma \vdash ((\{\alpha \mid \overrightarrow{\mathscr{I}}, \overrightarrow{\mathscr{G}}\}; AS), (\{\overline{\alpha} \mid \overrightarrow{\mathscr{I}}, \overrightarrow{\mathscr{G}}\} \parallel FS)) \longrightarrow (AS, FS)} \text{ FS PROG CONS}$$

$$\frac{\Gamma \vdash (FS_1', FS_2') \longrightarrow (FS_3', FS_4') \quad FS_i \equiv FS_i' \text{ for } i = 1, \ldots, 4}{\Gamma \vdash (FS_1, FS_2) \longrightarrow (FS_3, FS_4)} \text{ FS PROG CONG}$$

Figure 9: Progress Conditions for Flowstate

A *dataflow network* $N = \mathsf{network}(\Gamma, \Delta \vdash FS : P)$ is a composition of sequential actors under certain conditions. The composition P contains two form of actor bindings:

1. A simple actor binding of the form E.
2. An *actor comprehension*, an actor array binding of the form $\{E \mid t, x \leftarrow V_1..V_2\}$, that represents an array of actor, all of the same definition E, and with each actor provided with its index via the parameter x when it is initialized. This index ranges over the interval $\{V_1, \ldots, V_2\}$. An actor comprehension is the parallel equivalent of a loop for processing an array of channels.

In order to ensure the well-formedness of a network, we define some global restrictions based on the flowstates for the actors. We formulate these restrictions using the following additional judgement forms:

1. $\Gamma \vdash (FS_1, FS_2) \longrightarrow (FS_1', FS_2')$: Determines if it is possible to evolve from an initial flowstate FS_1 to a flowstate FS_1', where communications represented by FS_2 have already occurred, and leaving an updated record of communications FS_2'. The "records of communications" correspond to send events that are preconditions for receive events (data cannot be read until it written), and in the case of channels with delays, receive events that are preconditions for send events (data on a channel with a delay cannot be written until the data written in the previous firing cycle has been read).
2. $\Gamma \vdash FS$ **deterministic**: Determines that there is a single sending actor and single receiving actor for each channel (and it is not the same actor sending and receiving on a channel). The rules for this judgement form are provided in FIg. 10.

The rules for the first judgement form are provided in Fig. 9. These divide events into two broad categories: "producer" events and "consumer" events, defined by these predicates:

$$\begin{aligned}
Producer(\Gamma, t!) &\iff \Gamma(t) = \mathsf{Channel}(0, \tau_{limit}) \\
Producer(\Gamma, t[\tau]!) &\iff \Gamma(t) = \mathsf{Channel}(0, \tau_{limit}) \\
Producer(\Gamma, t?) &\iff \Gamma(t) = \mathsf{Channel}(1, \tau_{limit}) \\
Producer(\Gamma, t[\tau]?) &\iff \Gamma(t) = \mathsf{Channel}(1, \tau_{limit}) \\
Consumer(\Gamma, t!) &\iff \Gamma(t) = \mathsf{Channel}(1, \tau_{limit}) \\
Consumer(\Gamma, t[\tau]!) &\iff \Gamma(t) = \mathsf{Channel}(1, \tau_{limit}) \\
Consumer(\Gamma, t?) &\iff \Gamma(t) = \mathsf{Channel}(0, \tau_{limit}) \\
Consumer(\Gamma, t[\tau]?) &\iff \Gamma(t) = \mathsf{Channel}(0, \tau_{limit})
\end{aligned}$$

$$\frac{}{\Gamma \vdash \varepsilon \ \mathbf{deterministic}} \ \text{FS DET EMPTY}$$

$$\frac{}{\Gamma \vdash AS \ \mathbf{deterministic}} \ \text{FS DET ACTOR}$$

$$\frac{\begin{array}{cc} \Gamma \vdash FS_1 \ \mathbf{deterministic} & \Gamma \vdash FS_2 \ \mathbf{deterministic} \\ inchans(FS_1) \cap inchans(FS_2) = \{\} & outchans(FS_1) \cap outchans(FS_2) = \{\} \end{array}}{\Gamma \vdash (FS_1 \parallel FS_2) \ \mathbf{deterministic}} \ \text{FS DET PAR}$$

Figure 10: Determinism Conditions for Flowstate

In other words, $Producer(\Gamma, \alpha)$ is true if α corresponds to a communication event that is a precondition for another communication event in this firing cycle (sending on a channel with no delay, or receiving on a channel with a delay). $Consumer(\Gamma, \alpha)$ is true if α corresponds to a communication event that relied on a preceding communication event in this firing cycle.

In the rules in Fig. 9, the FS PROG PROD rule corresponds to a flowstate event that produces the communication to enable a subsequent event, e.g., the sending of a message that will later be consumed, on a channel with no delay:

$$\Gamma \vdash ((\{\alpha \mid \overrightarrow{\mathscr{I}}, \overrightarrow{\mathscr{G}}\};AS), FS) \longrightarrow (AS, (\{\alpha \mid \overrightarrow{\mathscr{I}}, \overrightarrow{\mathscr{G}}\} \parallel FS)).$$

The FS PROG CONS rule corresponds to a flowstate event that consumes the result of this communication later in the computation:

$$\Gamma \vdash ((\{\alpha \mid \overrightarrow{\mathscr{I}}, \overrightarrow{\mathscr{G}}\};AS), (\{\overline{\alpha} \mid \overrightarrow{\mathscr{I}}, \overrightarrow{\mathscr{G}}\} \parallel FS)) \longrightarrow (AS, FS).$$

This latter rule makes use of the notion of the *complement of an event*, defined by:

$$\begin{array}{rcl} \overline{t!} & = & t? \\ \overline{t[\tau]!} & = & t[\tau]? \\ \overline{t?} & = & t! \\ \overline{t[\tau]?} & = & t[\tau]! \end{array}$$

The rules for the $\Gamma \vdash FS$ **deterministic** judgement form are provided in Fig. 10. These rules make use of metafunctions that extract the input and output channels that processes communicate on, as reflected in the flowstate:

$$
\begin{array}{rcl}
\mathit{inchans}(t!,\overrightarrow{\mathscr{I}},\overrightarrow{\mathscr{G}}) & = & \{\} \\
\mathit{inchans}(t[\tau]!,\overrightarrow{\mathscr{I}},\overrightarrow{\mathscr{G}}) & = & \{\} \\
\mathit{inchans}(t?,\overrightarrow{\mathscr{I}},\overrightarrow{\mathscr{G}}) & = & \{t\} \\
\mathit{inchans}(t[\tau]?,\overrightarrow{\mathscr{I}},\overrightarrow{\mathscr{G}}) & = & \begin{cases} \{t[\underline{k}]\} & \text{if } \tau = \underline{k}, \text{ some } n \\ \{t[\underline{m}],\ldots,t[\underline{n}]\} & \text{if } \exists t_0.\tau = t_0 \text{ and } \overrightarrow{\mathscr{I}} = (t_0 \leftarrow \underline{m}..\underline{n}) \text{ and } \overrightarrow{\mathscr{G}} = \varepsilon \\ \{t[\underline{m}..t_1]\} & \text{if } \exists t_0.\tau = t_0 \text{ and } \overrightarrow{\mathscr{I}} = (t_0 \leftarrow \underline{m}..t_1) \text{ and } \overrightarrow{\mathscr{G}} = \varepsilon \end{cases} \\
\mathit{inchans}(\varepsilon) & = & \{\} \\
\mathit{inchans}(\{\alpha \mid \overrightarrow{\mathscr{I}},\overrightarrow{\mathscr{G}}\}) & = & \mathit{inchans}(\alpha,\overrightarrow{\mathscr{I}},\overrightarrow{\mathscr{G}}) \\
\mathit{inchans}(AS_1;AS_2) & = & \mathit{inchans}(AS_1) \cup \mathit{inchans}(AS_2) \\
\mathit{inchans}(FS_1 \parallel FS_2) & = & \mathit{inchans}(FS_1) \cup \mathit{inchans}(FS_2) \\
\mathit{outchans}(t!,\overrightarrow{\mathscr{I}},\overrightarrow{\mathscr{G}}) & = & \{t\} \\
\mathit{outchans}(t[\tau]!,\overrightarrow{\mathscr{I}},\overrightarrow{\mathscr{G}}) & = & \begin{cases} \{t[\underline{k}]\} & \text{if } \tau = \underline{k}, \text{ some } n \\ \{t[\underline{m}],\ldots,t[\underline{n}]\} & \text{if } \exists t_0.\tau = t_0 \text{ and } \overrightarrow{\mathscr{I}} = (t_0 \leftarrow \underline{m}..\underline{n}) \text{ and } \overrightarrow{\mathscr{G}} = \varepsilon \\ \{t[\underline{m}..t_1]\} & \text{if } \exists t_0.\tau = t_0 \text{ and } \overrightarrow{\mathscr{I}} = (t_0 \leftarrow \underline{m}..t_1) \text{ and } \overrightarrow{\mathscr{G}} = \varepsilon \end{cases} \\
\mathit{outchans}(t?,\overrightarrow{\mathscr{I}},\overrightarrow{\mathscr{G}}) & = & \{\} \\
\mathit{outchans}(t[\tau]?,\overrightarrow{\mathscr{I}},\overrightarrow{\mathscr{G}}) & = & \{\} \\
\mathit{outchans}(\varepsilon) & = & \{\} \\
\mathit{outchans}(\{\alpha \mid \overrightarrow{\mathscr{I}},\overrightarrow{\mathscr{G}}\}) & = & \mathit{outchans}(\alpha,\overrightarrow{\mathscr{I}},\overrightarrow{\mathscr{G}}) \\
\mathit{outchans}(AS_1;AS_2) & = & \mathit{outchans}(AS_1) \cup \mathit{outchans}(AS_2) \\
\mathit{outchans}(FS_1 \parallel FS_2) & = & \mathit{outchans}(FS_1) \cup \mathit{outchans}(FS_2)
\end{array}
$$

The *inchans* metafunction computes the set of channels on which a part of a dataflow network performs inputs (as reflected by the flowstate inferred for that network). The obvious complication is that for channel arrays. In general, we assume an event comprehension of the form

$$\{t[t_0]? \mid (t_0 \leftarrow \underline{m}..\tau), \overrightarrow{\mathscr{G}}\}$$

for some filtering conditions $\mathscr{G}$. We assume that these filtering conditions are empty for communication on an array of channels: There is communication on every channel in the array. We would expect that the lower bound $\underline{m}$ is $\underline{1}$, while the upper bound τ may be a type parameter. However we also use the type system to type check intermediate configurations in the next section, and in this case the lower bound may be greater than 1. In that case the upper bound will be instantiated to a number $\underline{n}$, and the equivalence rules allow the comprehension to be unrolled to a collection of singletons $t[\underline{k}]?$ for $k = m,\ldots,n$. We use heap typing to compute a flowstate that reflects communications that have been performed in this firing cycle. Then for an event comprehension of the form $\{t[t_0]? \mid (t_0 \leftarrow \underline{m}..\underline{n})\}$, where $m > 1$, the preceding inputs on channel t_0 will be reflected by singleton events $t[\underline{k}]?$ for $k = 1,\ldots,m-1$.

Similar remarks apply for the metafunction that computes the range of channels on which a subnet performs outputs.

We distinguish two cases when computing the channels on which a subnet performs input or output:

$$\begin{array}{rcl} H \in \text{Heap} & ::= & \varepsilon \;\mid\; l \mapsto V \;\mid\; c \mapsto B \;\mid\; c \mapsto \langle \overrightarrow{B} \rangle \;\mid\; H_1 \uplus H_2 \\ HT \in \text{Heap Type} & ::= & \varepsilon \;\mid\; l : \tau \;\mid\; c : \tau \;\mid\; HT_1 \uplus HT_2 \\ B \in \text{Buffer} & ::= & \varepsilon_k \;\mid\; [V]_k \;\mid\; B_1 @_k B_2 \\ C_{\textbf{DATA}} \in \text{Config} & ::= & (P, H) \end{array}$$

Figure 11: Configurations in $\mathbf{S_{DATA}}$

1. For the case where an agent communicates on an element of an array t, at index $\underline{k}$, we represent this by the array element $t[\underline{k}]$. This corresponds to the case where the upper bound on an event comprehension is instantiated to a number $\underline{n}$, and the equivalence rules unroll the event comprehension to a collection of communications on the elements of the channel array.
2. For the case where an agent communicates on a range of elements of an array t, as given by the iterator $t_0 \leftarrow \underline{m}..\tau$, we have two cases:
   (a) If the upper bound τ is instantiated (e.g. $\underline{n}$), then we compute the channels that are communicated over to be the same as those resulting from the unfolding of the comprehension, $\{t[\underline{m}],\ldots,t[\underline{n}]\}$
   (b) The other case is where the upper bound on an event comprehension is not yet instantiated. Let this supper bound be t_1, then we compute the range of channels as $t[\underline{m}..t_1]$ (where $m = 1$ in this case, since no loop unrolling happens before instantiation of the upper bound).

3 Operational Semantics

We provide a heap-based semantics that binds channels to message buffers on the heap. Message buffers B hold the values transmitted between actors on shared channels. A message buffer is simply a sequence, ensuring FIFO delivery, where $\_ @_k \_$ is the operation for appending buffers. We assume that buffers have bounded size, provided by a parameter k in the constructors and in the buffer type; the constructor operations are undefined for the case where the resulting buffer is larger than the maximum size. We denote the number of items in a buffer by $\mid B \mid$, and the maximum size of a buffer by $size(B)$. We write $[V_1, V_2, \ldots, V_m]_k$ as an abbreviation for $[V_1]_k @_k [V_2]_k @_k \ldots @_k [V_m]_k$, where $m \leq k$. We use $V ::_k B$ to denote $[V]_k @_k B$. We use $\tau[k]$ to denote the type of a buffer that contains values of type τ. These buffer types are not first class, since buffers are handled by the compiler.

There are three types of values stored on the heap: simple values (for reference cells), buffers (for communication channels), and arrays of buffers (for arrays of communication channels. An array value is a tuple of the form $A = \langle V_0, \ldots, V_{k-1} \rangle$. We denote $size(A) = k$. We denote array lookup by $A(i) = V_i$, for $0 \leq i < k$, and array update by $A[i \mapsto V])(j) = \langle V_0, \ldots, V, \ldots, V_{k-1} \rangle$, replacing the ith element of the array.

In order to reason about correctness, we define typing relations for heaps, using the judgement forms:

$$\begin{array}{ll} \Gamma;\Delta \vdash H : FS & \text{Heap} \\ \Gamma;\Delta \vdash B : FS & \text{Buffer} \end{array}$$

The type system not only ensures that values stored in a buffer have the correct type, but also that the buffer has sufficient items to satisfy communications between actors sharing that buffer.

For evaluating expressions, mutable base type variables are bound to locations l, and these must be dereferenced. This dereferencing is performed by the operation of applying the heap to a value, $H(V)$, defined by:

$$\begin{array}{rcl} H(l) & = & V \text{ if } l \mapsto V \in H \\ H(c) & = & B \text{ if } c \mapsto B \in H \\ H(c[\mathsf{index}(i)]) & = & B_i \text{ if } c \mapsto \langle \overrightarrow{B} \rangle \in H \end{array}$$

The semantics is defined using a collection of reduction relations:

$$\begin{array}{rl} \text{Reduction of expressions:} & (E_1,H_1) \longrightarrow (E_2,H_2) \text{ and} \\ & (E_1,H_1) \xrightarrow{\Delta,\alpha} (E_2,H_2) \\ \text{Reduction of processes:} & (P_1,H_1) \xrightarrow{a} (P_2,H_2) \text{ and} \\ & (P_1,H_1) \xrightarrow{\Delta,\alpha} (P_2,H_2) \\ \text{Reduction of flowstates:} & FS_1 \longrightarrow FS_2 \text{ and} \\ & FS_1 \xrightarrow{\alpha} FS_2 \end{array}$$

A reduction of expressions of the form $(E_1,H_1) \longrightarrow (E_2,H_2)$ denotes an internal reduction, while a reduction of the form $(E_1,H_1) \xrightarrow{\Delta,\alpha} (E_2,H_2)$ denotes a reduction that involves a communication event α. We write $(E_1,H_1) \xrightarrow{[\Delta,\alpha]} (E_2,H_2)$ to generically denote a reduction that may be either internal or involve a communication event. Similar remarks hold for reduction of processes.

The reduction relation for flowstates is perhaps surprising, and reflects the use of flowstate: Types themselves evolve during computation, since they are abstract process descriptions for the underlying sequential program. The reduction relation for flowstates is defined in Fig. 13.

Our basic result is that evaluation preserves types, in the sense that a type may simulate the communications performed at the value level:

Theorem 1 (Type Preservation) *If $\Gamma;\Delta \vdash P_1 : FS_1$ and $\Gamma;\Delta \vdash H_1 : FS'_1$, and $(P_1,H_1) \xrightarrow{[\Delta,\alpha]} (P_2,H_2)$, then $\Gamma;\Delta \vdash P_2 : FS_2$ and $\Gamma;\Delta \vdash H_2 : FS'_2$, for some FS_2 and FS'_2, where $FS_1 \xrightarrow{\alpha} FS_2$ and one of the following holds:*

1. *Either Producer(Γ,α) is true and $FS'_2 \equiv (FS'_1 \parallel \alpha)$; or*
2. *Consumer(Γ,α) is true and $FS'_1 \equiv (FS'_2 \parallel \overline{\alpha})$.*

Our progress result reflects that computation is not deadlocked, provided the initial heap is compatible with the remaining actor computation, as reflected in the flowstate. Note that this result is for a single firing of the dataflow graph; for simplicity, we do not consider the unrolling of the graph for another firing.

Theorem 2 (Progress) *If $\Gamma;\Delta \vdash P_1 : FS_1$ and $\Gamma;\Delta \vdash H_1 : FS'_1$, and $\Gamma \vdash (FS_1,FS'_1) \longrightarrow (FS_2,FS'_2)$, then $(P_1,H_1) \xrightarrow{[\Delta,\alpha]} (P_2,H_2)$, for some P_2, H_2 and α.*

Our dataflow language elides any compositional constructs for building dataflow graphs "bottom-up." Examples of operators that might be used for such compositional constructs are provided elsewhere [17, 7]. Central to that work is the abstraction of the communication structure of a component dataflow graph, exposing causal dependencies in communication channels to prevent deadlock during incremental construction of a dataflow graph. An interesting direction for future research would be to consider how to combine the parameterized dataflow considered in this report with those causalities.

$$\frac{m \le n \quad E' = (\mathsf{for}\ (t,x \in (m+1)..n)\ E)}{((\mathsf{for}\ (t,x \in m..\mathsf{size}(n))\ E),H) \longrightarrow ((\{\mathsf{index}(m)/x\}E;E'),H)}\ \text{Red For True}$$

$$\frac{m > n}{((\mathsf{for}\ (t,x \in m..\mathsf{size}(n))\ E),H) \longrightarrow (0,H)}\ \text{Red For False}$$

$$\frac{V_1 \rho V_2}{((\mathsf{when}\ V_1 \rho V_2\ \mathsf{do}\ E),H) \longrightarrow (E,H)}\ \text{Red When True}$$

$$\frac{\neg(V_1 \rho V_2)}{((\mathsf{when}\ V_1 \rho V_2\ \mathsf{do}\ E),H) \longrightarrow (0,H)}\ \text{Red When False}$$

$$\frac{}{(\mathsf{fromSize}(\mathsf{size}(m)),H) \longrightarrow (m,H)}\ \text{Red From Size}$$

$$\frac{}{(\mathsf{fromIndex}(\mathsf{index}(m)),H) \longrightarrow (m,H)}\ \text{Red From Index}$$

$$\frac{k = size(H(c)) \quad |\,H(c)\,| < k \quad H' = H[c \mapsto H(c)@_k[V]_k] \quad (c : \mathsf{Channel}(\pi,t,\tau)) \in \Delta}{(c!V,H) \xrightarrow{\Delta,t!} (0,H')}\ \text{Red Send}$$

$$\frac{H(c) = [V]_k@_kB \quad H' = H[c \mapsto B] \quad (c : \mathsf{Channel}(\pi,t,\tau)) \in \Delta}{(c?,H) \xrightarrow{\Delta,t?} (V,H')}\ \text{Red Receive}$$

$$\frac{k = size(H(c[V_0])) \quad |\,H(c[V_0])\,| < k \quad V_0 = \mathsf{index}(m) \quad H' = H[c[V_0] \mapsto H(c[V_0])@_k[V]_k] \quad (c : \mathsf{Channel}(\pi,t,\tau)[\tau_0]) \in \Delta}{(c[V_0]!V,H) \xrightarrow{\Delta,t[\underline{m}]!} (0,H')}\ \text{Red Send Array}$$

$$\frac{H(c[V_0]) = [V]_k@_kB \quad H' = H[c[V_0] \mapsto B] \quad V_0 = \mathsf{index}(m) \quad (c : \mathsf{Channel}(\pi,t,\tau)[\tau_0]) \in \Delta}{(c[V_0]?,H) \xrightarrow{\Delta,t[\underline{m}]?} (V,H')}\ \text{Red Receive Array}$$

Figure 12: Operational Semantics for $\mathbf{S_{DATA}}$: Dataflow Semantics

4 Related Work

The notion of types that describe resource usage largely come out of the realm of linear [19, 10, 8, 3] and affine [16] type systems for statically checking the safe usage of limited resources. The typical approach is to provide a linear type system where we are guaranteed exactly one reference to a resource. Two particularly significant lines of study in the "linear types" field have been the approach of *typestate* [4] and that of *session types* [6]. The current work uses the framework of sessional dataflow, that combines dataflow with session types. *Usage types* [12] have a similar motivation to sessional dataflow, statically preventing the composition of concurrent components that would produce deadlocks. The approach of session types [6] is commonly motivated by its support for safe Web services. Session types have been realized in both functional [18] and object-oriented language [1] semantics, with both synchronous and asynchronous semantics. Dyadic session types have also been generalized to multiparty [9] interactions, where potentially more than two parties are involved in an interaction. This approach has further been generalized to a dynamically varying number of participants, based on assigning roles [5] to participants and describing generic protocols for each participant role.

$$FS \parallel \varepsilon \equiv FS \qquad FS_1 \parallel FS_2 \equiv FS_2 \parallel FS_1$$

$$(FS_1 \parallel FS_2) \parallel FS_3 \equiv FS_1 \parallel (FS_2 \parallel FS_3)$$

$$FS;\varepsilon \equiv FS \qquad FS_1;FS_2 \equiv FS_2;FS_1$$

$$(FS_1;FS_2);FS_3 \equiv FS_1;(FS_2;FS_3)$$

$$\frac{FS_1 \xrightarrow{\alpha} FS_1'}{(FS_1 \parallel FS_2) \xrightarrow{\alpha} (FS_1' \parallel FS_2)}$$

$$\frac{FS_1 \xrightarrow{\alpha} FS_1'}{(FS_1;FS_2) \xrightarrow{\alpha} (FS_1';FS_2)}$$

$$\frac{m \leq n}{\{\alpha \mid (\overrightarrow{\mathscr{I}}, t \leftarrow \underline{m}..\underline{n}), \overrightarrow{\mathscr{G}}\} \longrightarrow (\{\underline{m}/t\}\{\alpha \mid \overrightarrow{\mathscr{I}}, \overrightarrow{\mathscr{G}}\});\{\alpha \mid (\overrightarrow{\mathscr{I}}, t \leftarrow (\underline{m+1})..\underline{n}), \overrightarrow{\mathscr{G}}\}}$$

$$\frac{m > n}{\{\alpha \mid (\overrightarrow{\mathscr{I}}, t \leftarrow \underline{m}..\underline{n}), \overrightarrow{\mathscr{G}}\} \longrightarrow \{\alpha \mid \overrightarrow{\mathscr{I}}, \overrightarrow{\mathscr{G}}\}}$$

$$\frac{m\rho n}{\{\alpha \mid \overrightarrow{\mathscr{I}}, (\overrightarrow{\mathscr{G}}, \underline{m}\rho\underline{n})\} \longrightarrow \{\alpha \mid \overrightarrow{\mathscr{I}}, \overrightarrow{\mathscr{G}}\}}$$

$$\frac{\neg(m\rho n)}{\{\alpha \mid \overrightarrow{\mathscr{I}}, (\overrightarrow{\mathscr{G}}, \underline{m}\rho\underline{n})\} \longrightarrow \varepsilon}$$

$$\frac{}{\alpha;FS \xrightarrow{\alpha} FS}$$

$$\frac{FS_1 \equiv FS_1' \qquad FS_1' \xrightarrow{\alpha} FS_2' \qquad FS_2 \equiv FS_2'}{FS_1 \xrightarrow{\alpha} FS_2}$$

Figure 13: Type reduction rules

$$(V \parallel P) \equiv P \qquad (P_1 \parallel P_2) \equiv (P_2 \parallel P_1)$$

$$(P_1 \parallel (P_2 \parallel P_3)) \equiv ((P_1 \parallel P_2) \parallel P_3)$$

$$\frac{m \leq n}{\{\alpha \mid t \leftarrow \underline{m}..\underline{n}\} \equiv (\{\underline{m}/t\}\alpha \parallel \ldots \parallel \{\underline{n}/t\}\alpha)}$$

Figure 14: Structural equivalence for processes

Another related line of work is in synchronous languages for real-time and embedded systems. The constraints on the synchronous languages preclude any need for buffering, since all actors operate in lock step on the same clock[1]. The theory of N-synchronous Kahn networks [2] relaxes the synchrony restriction, allowing different actors to have their own clock rates, and allowing buffering between actors to match their clock rates. It is therefore related to the approach of synchronous dataflow [13]. While N-synchronous Kahn networks uses the different clock speeds of the actors to compute the amount of buffer space required, and to schedule the execution of the actors, SDF is instead using the data rates of

[1] Lustre and its descendants allow multiple clocks, used by different components, but all clocks have a common base clock. A true multi-clock synchronous language is the Signal language, but consideration of Signal is outside the scope of the current work.

$$\frac{V = (\lambda \overrightarrow{x} : \overrightarrow{\tau}.AS_1 \Rightarrow E)}{(V(\overrightarrow{V}), H) \longrightarrow (\{\overrightarrow{V}/\overrightarrow{x}\}E, H)} \text{ RED APP}$$

$$\frac{}{((\mathsf{let}\ x = V\ \mathsf{in}\ E), H) \longrightarrow (\{V/x\}E, H)} \text{ RED LET}$$

$$\frac{}{(*l, H) \longrightarrow (H(l), H)} \text{ RED DEREF}$$

$$\frac{H' = H[l \mapsto V]}{(l := V, H) \longrightarrow (V, H')} \text{ RED ASSIGN}$$

$$\frac{}{((\mathsf{if\ true\ then}\ E_1\ \mathsf{else}\ E_2), H) \longrightarrow (E_1, H)} \text{ RED IF TRUE}$$

$$\frac{}{((\mathsf{if\ false\ then}\ E_1\ \mathsf{else}\ E_2), H) \longrightarrow (E_2, H)} \text{ RED IF FALSE}$$

$$\frac{(E_1, H) \xrightarrow{[\Delta,\alpha]} (E_2, H')}{(\mathbb{E}[E_1], H) \xrightarrow{[\Delta,\alpha]} (\mathbb{E}[E_2], H')} \text{ RED EXP CONG}$$

$$\frac{(P_1, H) \xrightarrow{[\Delta,\alpha]} (P_2, H')}{(\mathbb{P}[P_1], H) \xrightarrow{[\Delta,\alpha]} (\mathbb{P}[P_2], H')} \text{ RED PROC CONG}$$

Figure 15: Operational Semantics for $\mathbf{S_{DATA}}$: Core Semantics

the actors on their input and output channels to compute buffer sizes and perform scheduling.

5 Conclusions

We have described a type and effect system for a dataflow language, that allows the firing rates of actors to be parameterized in their description, while allowing modular analysis of actor bodies for their firing behavior. An obvious direction for future research is to consider the combination of this with the compositional sessional dataflow system described in [7]. The main challenge here is the assumption, in the system described in this paper, that there are no causal dependencies between communications in different iterations of a loop.

References

[1] Sara Capecchi, Mario Coppo, Mariangiola Dezani-Ciancaglini, Sophia Drossopoulou & Elena Giachino (2009): *Amalgamating sessions and methods in object-oriented languages with generics*. *Theor. Comput. Sci.* 410, pp. 142–167, doi:`10.1016/j.tcs.2008.09.016`.

[2] Albert Cohen, Marc Duranton, Christine Eisenbeis, Claire Pagetti, Florence Plateau & Marc Pouzet (2006): *N-synchronous Kahn networks: a relaxed model of synchrony for real-time systems*. In: *Principles of Programming Languages (POPL)*, ACM Press, pp. 180–193, doi:`10.1145/1111037.1111054`.

[3] Robert DeLine & Manuel Fähndrich (2001): *Enforcing high-level protocols in low-level software*. *SIGPLAN Not.* 36, pp. 59–69, doi:`10.1145/381694.378811`.

[4] Robert Deline & Manuel Fãhndrich (2004): *Typestates for Objects*. In: *European Conference on Object-Oriented Programming (ECOOP)*, Springer, pp. 465–490, doi:`10.1007/978-3-540-24851-4_21`.

[5] Pierre-Malo Deniélou & Nobuko Yoshida (2011): *Dynamic multirole session types*. In: *ACM Symposium on Principles of Programming Languages*, ACM, New York, NY, USA, pp. 435–446, doi:10.1145/1926385.1926435.

[6] Mariangiola Dezani-Ciancaglini & Ugo De'Liguoro (2010): *Sessions and session types: an overview*. In: *Proceedings of the 6th international conference on Web services and formal methods*, WS-FM'09, Springer-Verlag, pp. 1–28, doi:10.1007/978-3-642-14458-5_1.

[7] Dominic Duggan & Jianhua Yao (2012): *Static Sessional Dataflow*. In: *European Conference on Object-Oriented Programming (ECOOP)*, Beijing, doi:10.1007/978-3-642-31057-7_22.

[8] Manuel Fahndrich & Robert DeLine (2002): *Adoption and focus: practical linear types for imperative programming*. In: *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, ACM, New York, NY, USA, pp. 13–24, doi:10.1145/512529.512532.

[9] Kohei Honda, Nobuko Yoshida & Marco Carbone (2008): *Multiparty asynchronous session types*. *SIGPLAN Not.* 43, pp. 273–284, doi:10.1145/1328897.1328472.

[10] Trevor Jim, J. Greg Morrisett, Dan Grossman, Michael W. Hicks, James Cheney & Yanling Wang (2002): *Cyclone: A Safe Dialect of C*. In: *USENIX Annual Technical Conference*, USENIX Association, Berkeley, CA, USA, pp. 275–288.

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

[12] N. Kobayashi (2003): *Type-based information flow analysis for the pi-calculus*. *Acta Informatica*, doi:10.1007/s00236-005-0179-x.

[13] E.A. Lee & D.G. Messerschmitt (1987): *Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing*. *IEEE Trans. Computers* 36(1), pp. 24–35, doi:10.1109/TC.1987.5009446.

[14] Edward Lee & David Messerschmitt: *Synchronous Data Flow: Describing Signal Processing Algorithm for Parallel Computation*. In: *COMPCON'87, Digest of Papers, Thirty-Second IEEE Computer Society International Conference, San Francisco, California, USA, February 23-27, 1987*, pp. 310–315.

[15] William Thies (2009): *Language and Compiler Support for Stream Programs*. Ph.d. thesis, Massachusetts Institute of Technology, Cambridge, MA.

[16] Jesse A. Tov & Riccardo Pucella (2011): *Practical affine types*. In: *Principles of Programming Languages (POPL)*, ACM, New York, NY, pp. 447–458, doi:10.1145/1926385.1926436.

[17] Stavros Tripakis, Dai N. Bui, Marc Geilen, Bert Rodiers & Edward A. Lee (2013): *Compositionality in synchronous data flow: Modular code generation from hierarchical SDF graphs*. *ACM Trans. Embedded Comput. Syst.* 12(3), p. 83, doi:10.1145/2442116.2442133.

[18] Vasco Vasconcelos, António Ravara & Simon Gay (2004): *Session Types for Functional Multithreading*. In: *CONCUR'04*, Springer-Verlag, pp. 497–511, doi:10.1007/978-3-540-28644-8_32.

[19] Philip Wadler (1990): *Linear Types Can Change the World!* In: *Programming Concepts and Methods*, North.