Parallel Complexity Analysis with Temporal Session Types

ANKUSH DAS, Carnegie Mellon University, USA
JAN HOFFMANN, Carnegie Mellon University, USA
FRANK PFENNING, Carnegie Mellon University, USA

We study the problem of parametric parallel complexity analysis of concurrent, message-passing programs. To make the analysis local and compositional, it is based on a conservative extension of binary session types, which structure the type and direction of communication between processes and stand in a Curry-Howard correspondence with intuitionistic linear logic. The main innovation is to enrich session types with the *temporal modalities next* ($\bigcirc A$), *always* ($\Box A$), and *eventually* ($\Diamond A$), to additionally prescribe the timing of the exchanged messages in a way that is precise yet flexible. The resulting *temporal session types* uniformly express properties such as the message rate of a stream, the latency of a pipeline, the response time of a concurrent queue, or the span of a fork/join parallel program. The analysis is parametric in the cost model and the presentation focuses on communication cost as a concrete example. The soundness of the analysis is established by proofs of progress and type preservation using a timed multiset rewriting semantics. Representative examples illustrate the scope and usability of the approach.

CCS Concepts: • **Theory of computation → Concurrency**; **Modal and temporal logics**; *Linear logic*;

Additional Key Words and Phrases: Session Types, Linear logic, Concurrency, Resource analysis

ACM Reference Format:
Ankush Das, Jan Hoffmann, and Frank Pfenning. 2018. Parallel Complexity Analysis with Temporal Session Types. *Proc. ACM Program. Lang.* 2, ICFP, Article 91 (September 2018), 30 pages. https://doi.org/10.1145/3236786

1 INTRODUCTION

For sequential programs, several type systems and program analyses have been proposed to structure, formalize [Danner et al. 2015; Lago and Gaboardi 2011; Çiçek et al. 2017], and automate [Avanzini et al. 2015; Gulwani et al. 2009; Hoffmann et al. 2017] complexity analysis. Analyzing the complexity of concurrent, message-passing processes poses additional challenges that these systems do not address. To begin with, we need information about the possible interactions between processes to enable compositional and local reasoning about concurrent cost.

Session types [Honda et al. 1998] provide a structured way to prescribe communication behavior between message-passing processes and are a natural foundation for compositional, concurrent complexity analysis. In particular, we use a system of binary session types that stands in a Curry-Howard correspondence with intuitionistic linear logic [Caires and Pfenning 2010; Caires et al. 2016]. Our communication model is *asynchronous* in the sense of the asynchronous π-calculus: sending always succeeds immediately, while receiving blocks until a message arrives.

In addition to the structure of communication, the timing of messages is of central interest for analyzing concurrent cost. With information on message timing we may analyze not only properties such as the rate or latency with which a stream of messages can proceed through a

Authors' addresses: Ankush Das, Carnegie Mellon University, USA, ankushd@cs.cmu.edu; Jan Hoffmann, Carnegie Mellon University, USA, jhoffmann@cmu.edu; Frank Pfenning, Carnegie Mellon University, USA, fp@cs.cmu.edu.

2475-1421/2018/9-ART91
https://doi.org/10.1145/3236786

pipeline, but also the span of a parallel computation, which can be defined as the time of the final response message assuming maximal parallelism.

There are several possible ways to enrich session types with timing information. A challenge is to find a balance between precision and flexibility. We would like to express precise times according to a global clock as in synchronous data flow languages whenever that is possible. However, sometimes this will be too restrictive. For example, we may want to characterize the response time of a concurrent queue where enqueue and dequeue operations arrive at unpredictable intervals.

In this paper, we develop a type system that captures the parallel complexity of session-typed message-passing programs by adding *temporal modalities next* ($\bigcirc A$), *always* ($\Box A$), and *eventually* ($\Diamond A$), interpreted over a linear model of time. When considered as types, the temporal modalities allow us to express properties of concurrent programs such as the *message rate* of a stream, the *latency* of a pipeline, the *response time* of concurrent data structure, or the *span* of a fork/join parallel program, all in the same uniform manner. Our results complement prior work on expressing the *work* of session-typed processes in the same base language [Das et al. 2017]. Together, they form a foundation for analyzing the parallel implementation complexity of session-typed processes.

The type system is constructed conservatively over the base language of session types, which makes it quite general and easily able to accommodate various concrete cost models. Our language contains standard session types and process expressions, and their typing rules remain unchanged. They correspond to processes that do not induce cost and send all messages at the constant time 0.

To model computation cost we introduce a new syntactic form delay, which advances time by one step. To specify a particular cost semantics we take an ordinary, non-temporal program and add delays capturing the intended cost. For example, if we decide only the blocking operations should cost one unit of time, we add a delay before the continuation of every receiving construct. If we want sends to have unit cost as well, we also add a delay immediately after each send operation. Processes that contain delays cannot be typed using standard session types.

To type processes with non-zero cost, we first introduce the type $\bigcirc A$, which is inhabited only by the process expression (delay ; P). This forces time to advance on all channels that P can communicate along. The resulting types prescribe the *exact* time a message is sent or received and sender and receiver are precisely synchronized.

As an example, consider a stream of bits terminated by \$, expressed as the recursive session type

$$\mathsf{bits} = \oplus\{\mathsf{b0} : \mathsf{bits}, \mathsf{b1} : \mathsf{bits}, \$: \mathbf{1}\}$$

where $\oplus$ stands for *internal choice* and $\mathbf{1}$ for *termination*, ending the session. A simple cost model for asynchronous communication prescribes a cost of one unit of time for every receive operation. A stream of bits then needs to delay every continuation to give the recipient time to receive the message, expressing a *rate* of one. This can be captured precisely with the temporal modality $\bigcirc A$:

$$\mathsf{bits} = \oplus\{\mathsf{b0} : \bigcirc\mathsf{bits}, \mathsf{b1} : \bigcirc\mathsf{bits}, \$: \bigcirc\mathbf{1}\}$$

A transducer *neg* that negates each bit it receives along channel x and passes it on along channel y would be typed as

$$x : \mathsf{bits} \vdash \mathit{neg} :: (y : \bigcirc\mathsf{bits})$$

expressing a *latency* of one. A process *negneg* that puts two negations in sequence has a latency of two, compared with *copy* which passes on each bit, and *id* which terminates and identifies the channel y with the channel x, short-circuiting the communication.

$$x : \mathsf{bits} \vdash \mathit{negneg} :: (y : \bigcirc\bigcirc\mathsf{bits}) \qquad x : \mathsf{bits} \vdash \mathit{copy} :: (y : \bigcirc\mathsf{bits}) \qquad x : \mathsf{bits} \vdash \mathit{id} :: (y : \mathsf{bits})$$

All these processes have the same extensional behavior, but different latencies. They also have the same rate since after the pipelining delay, the bits are sent at the same rate they are received, as expressed in the common type bits used in the context and the result.

While precise and minimalistic, the resulting system is often too precise for typical concurrent programs such as pipelines or servers. We therefore introduce the dual type formers $\Diamond A$ and $\Box A$ to talk about varying time points in the future. Remarkably, even if part of a program is typed using these constructs, we can still make precise and useful statements about other aspects.

For example, consider a transducer *compress* that shortens a stream by combining consecutive 1 bits so that, for example, 00110111 becomes 00101. For such a transducer, we cannot bound the latency statically, even if the bits are received at a constant rate like in the type bits. So we have to express that after seeing a 1 bit we will *eventually* see either another bit or the end of the stream. For this purpose, we introduce a new type sbits with the same message alternatives as bits, but different timing. In particular, after sending b1 we have to send either the next bit or end-of-stream *eventually* ($\Diamond$sbits), rather than immediately.

$$\mathsf{sbits} = \oplus\{\mathsf{b0} : \bigcirc\mathsf{sbits}, \mathsf{b1} : \bigcirc\Diamond\mathsf{sbits}, \$: \bigcirc\mathbf{1}\}$$
$$x : \mathsf{bits} \vdash \mathit{compress} :: (y : \bigcirc\mathsf{sbits})$$

We write $\bigcirc\Diamond$sbits instead of $\Diamond$sbits for the continuation type after b1 to express that there will always be a delay of at least one; to account for the unit cost of receive in this particular cost model.

The dual modality, $\Box A$, is useful to express, for example, that a server providing A is *always* ready, starting from "now". As an example, consider the following temporal type of an interface to a process of type $\Box\mathsf{queue}_A$ with elements of type $\Box A$. It expresses that there must be at least four time units between successive enqueue operations and that the response to a dequeue request is immediate, only one time unit later ($\&$ stands for external choice, the dual to internal choice).

$$\begin{aligned}\mathsf{queue}_A = \&\{\ &\mathsf{enq} : \bigcirc(\Box A \multimap \bigcirc^3\Box\mathsf{queue}_A),\\ &\mathsf{deq} : \bigcirc\oplus\{\ \mathsf{none} : \bigcirc\mathbf{1}, \mathsf{some} : \bigcirc(\Box A \otimes \bigcirc\Box\mathsf{queue}_A)\ \}\ \}\end{aligned}$$

As an example of a *parametric* cost analysis, we can give the following type to a process that appends inputs l_1 and l_2 to yield l, where the message rate on all three lists is $r + 2$ units of time (that is, the interval between consecutive list elements needs to be at least 2).

$$l_1 : \mathsf{list}_A[n], l_2 : \bigcirc^{(r+4)n+2}\,\mathsf{list}_A[k] \vdash \mathit{append} :: (l : \bigcirc\bigcirc\mathsf{list}_A[n+k])$$

It expresses that *append* has a latency of two units of time and that it inputs the first message from l_2 after $(r + 4)n + 2$ units of time, where n is the number of elements sent along l_1.

To analyze the span of a fork/join parallel program, we capture the time at which the (final) answer is sent. For example, the type tree$[h]$ describes the span of a process that computes the parity of a binary tree of height h with boolean values at the leaves. The session type expresses that the result of the computation is a single boolean that arrives at time $5h + 3$ after the parity request.

$$\mathsf{tree}[h] = \&\{\ \mathsf{parity} : \bigcirc^{5h+3}\,\mathsf{bool}\ \}$$

In summary, the main contributions of the paper are (1) a generic framework for parallel cost analysis of asynchronously communicating session-typed processes rooted in a novel combination of temporal and linear logic, (2) a soundness proof of the type system with respect to a timed operational semantics, showing progress and type preservation (3) instantiations of the framework with different cost models, e.g. where either just receives, or receives and sends, cost one time unit each, and (4) examples illustrating the scope of our method. Our technique for proving progress and preservation does not require dependency graphs and may be of independent interest. We further provide decidable systems for *time reconstruction* and *subtyping* that greatly simplify the programmer's task. They also enhance modularity by allowing the same program to be assigned temporally different types, depending on the context of use.

Related is work on space and time complexity analysis of interaction nets by Gimenez and Moser [2016], which is a parallel execution model for functional programs. While also inspired by linear logic and, in particular, proof nets, it treats only special cases of the additive connectives and

Type	**Provider Action**	**Session Continuation**
$\oplus\{\ell : A_\ell\}_{\ell \in L}$	send label $k \in L$	A_k
$\&\{\ell : A_\ell\}_{\ell \in L}$	receive and branch on label $k \in L$	A_k
$\mathbf{1}$	send token close	*none*
$A \otimes B$	send channel $c : A$	B
$A \multimap B$	receive channel $c : A$	B

Fig. 1. Basic Session Types. Every provider action has a matching client action.

recursive types and does not have analogues of the $\Box$ and $\Diamond$ modalities. It also does not provide a general source-level programming notation with a syntax-directed type system. On the other hand they incorporate sharing and space bounds, which are beyond the scope of this paper.

Another related thread is the research on timed multiparty session types [Bocchi et al. 2014] for modular verification of real-time choreographic interactions. Their system is based on explicit global timing interval constraints, capturing a new class of communicating timed automata, in contrast to our system based on binary session types in a general concurrent language. Therefore, their system has no need for general $\Box$ and $\Diamond$ modalities, the ability to pass channels along channels, or the ability to identify channels via forwarding. Their work is complemented by an expressive dynamic verification framework in real-time distributed systems [Neykova et al. 2014], which we do not consider. Semantics counting communication costs for work and span in session-typed programs were given by Silva et al. [2016], but no techniques for analyzing them were provided.

The remainder of the paper is organized as follows. We review our basic system of session types in Section 2, then introduce the next-time modality $\bigcirc A$ in Section 3 followed by $\Diamond A$ and $\Box A$ in Section 4. We establish fundamental metatheoretic type safety properties in Section 5 and time reconstruction in Section 6. Additional examples in Section 7 are followed by a theorem in Section 8 connecting the semantics presented in Figure 4 to the standard semantics of session-typed programs. Section 9 discusses further related work followed by a brief conclusion.

2 THE BASE SYSTEM OF SESSION TYPES

The underlying base system of session types is derived from a Curry-Howard interpretation of intutionistic linear logic [Caires and Pfenning 2010; Caires et al. 2016]. We present it here to fix our particular formulation, which can be considered the purely linear fragment of SILL [Pfenning and Griffith 2015; Toninho et al. 2013]. Remarkably, the rules remain exactly the same when we consider temporal extensions in the next section. The key idea is that an intuitionistic linear sequent

$$A_1, A_2, \ldots, A_n \vdash C$$

is interpreted as the interface to a *process expression* P. We label each of the antecedents with a channel name x_i and the succedent with channel name z. The x_i's are *channels used by P* and z is the *channel provided by P*.

$$x_1 : A_1, x_2 : A_2, \ldots, x_n : A_n \vdash P :: (z : C)$$

The resulting judgment formally states that process P provides a service of session type C along channel z, while using the services of session types $A_1, \ldots, A_n$ provided along channels $x_1, \ldots, x_n$ respectively. All these channels must be distinct, and we sometimes implicitly rename them to preserve this presupposition. We abbreviate the antecedent of the sequent by Ω.

Figure 1 summarizes the basic session types and their actions. The process expression for these actions are shown in Figure 2; the process typing rules in Figure 3. The first few examples (well into Section 4) only use internal choice, termination, and recursive types, together with process

		Expression	**Action**	**Continuation**	**Rules**
P, Q	$::=$	$x \leftarrow f \leftarrow \overline{e} \; ; Q$	spawn process named f	$[a/x]Q$	def
	$\mid$	$x{:}A \leftarrow P \; ; Q$	spawn $[a/x]P$	$[a/x]Q$	cut
	$\mid$	$c \leftarrow d$	identify c and d	*none*	id
	$\mid$	$c.k \; ; P$	send label k along c	P	$\oplus R, \& L$
	$\mid$	case $c \; (\ell \Rightarrow P_\ell)_{\ell \in L}$	receive label k along c	P_k	$\oplus L, \& R$
	$\mid$	close c	close c	*none*	$\mathbf{1}R$
	$\mid$	wait $c \; ; P$	wait for c to close	P	$\mathbf{1}L$
	$\mid$	send $c \; d \; ; P$	send d along c	P	$\otimes R, \multimap L$
	$\mid$	$x \leftarrow$ recv $c \; ; P$	receive d along c	$[d/x]P$	$\otimes L, \multimap R$

Fig. 2. Basic Process Expressions

definitions and forwarding, so we explain these in some detail together with their formal operational semantics. A summary of all the operational semantics rules can be found in Figure 4.

2.1 Internal Choice

A type A is said to describe a *session*, which is a particular sequence of interactions. As a first type construct we consider *internal choice* $\oplus\{\ell : A_\ell\}_{\ell \in L}$, an n-ary labeled generalization of the linear logic connective $A \oplus B$. A process that provides $x : \oplus\{\ell : A_\ell\}_{\ell \in L}$ can send any label $k \in L$ along x and then continue by providing $x : A_k$. We write the corresponding process as $(x.k \; ; P)$, where P is the continuation. This typing is formalized by the *right rule* $\oplus R$ in our sequent calculus. The corresponding client branches on the label received along x as specified by the *left rule* $\oplus L$.

$$\frac{(k \in L) \quad \Omega \vdash P :: (x : A_k)}{\Omega \vdash (x.k \; ; P) :: (x : \oplus\{\ell : A_\ell\}_{\ell \in L})} \oplus R \qquad \frac{(\forall \ell \in L) \quad \Omega, x{:}A_\ell \vdash Q_\ell :: (z : C)}{\Omega, x{:}\oplus\{\ell : A_\ell\}_{\ell \in L} \vdash \mathsf{case}\; x \; (\ell \Rightarrow Q_\ell)_{\ell \in L} :: (z : C)} \oplus L$$

We formalize the operational semantics as a system of *multiset rewriting rules* [Cervesato and Scedrov 2009]. We introduce semantic objects proc(c, t, P) and msg(c, t, M) which mean that process P or message M provide along channel c and are at an integral time t. A *process configuration* is a multiset of such objects, where any two offered channels are distinct. Communication is asynchronous, so that a process $(c.k \; ; P)$ sends a message k along c and continues as P without waiting for it to be received. As a technical device to ensure that consecutive messages on a channel arrive in order, the sender also creates a fresh continuation channel c' so that the message k is actually represented as $(c.k \; ; c \leftarrow c')$ (read: send k along c and continue as c').

$$(\oplus S) \quad \mathsf{proc}(c, t, c.k \; ; P) \;\mapsto\; \mathsf{proc}(c', t, [c'/c]P), \mathsf{msg}(c, t, c.k \; ; c \leftarrow c') \quad (c' \text{ fresh})$$

When the message k is received along c, we select branch k and also substitute the continuation channel c' for c.

$$(\oplus C) \quad \mathsf{msg}(c, t, c.k \; ; c \leftarrow c'), \mathsf{proc}(d, t, \mathsf{case}\; c \; (\ell \Rightarrow Q_\ell)_{\ell \in L}) \;\mapsto\; \mathsf{proc}(d, t, [c'/c]Q_k)$$

The *message* $(c.k \; ; c \leftarrow c')$ is just a particular form of process, where $c \leftarrow c'$ is *identity* or *forwarding*, explained in Section 2.3. Therefore no separate typing rules for messages are needed; they can be typed as processes [Balzer and Pfenning 2017].

In the receiving rule we require the time t of the message and receiver process to match. Until we introduce temporal types, this is trivially satisfied since all actions are considered instantaneous and processes will always remain at time $t = 0$.

$$\frac{\Omega' \vdash P :: (x : A) \quad \Omega, x : A \vdash Q :: (z : C)}{\Omega, \Omega' \vdash (x{:}A \leftarrow P \;;\; Q) :: (z : C)}\;\mathsf{cut} \qquad \frac{}{y : A \vdash (x \leftarrow y) :: (x : A)}\;\mathsf{id}$$

$$\frac{(k \in L) \quad \Omega \vdash P :: (x : A_k)}{\Omega \vdash (x.k \;;\; P) :: (x : \oplus\{\ell : A_\ell\}_{\ell \in L})}\;\oplus R \qquad \frac{(\forall \ell \in L) \quad \Omega, x{:}A_\ell \vdash Q_\ell :: (z : C)}{\Omega, x{:}\oplus\{\ell : A_\ell\}_{\ell \in L} \vdash \mathsf{case}\ x\ (\ell \Rightarrow Q_\ell)_{\ell \in L} :: (z : C)}\;\oplus L$$

$$\frac{(\forall \ell \in L) \quad \Omega \vdash P_\ell :: (x : A_\ell)}{\Omega \vdash \mathsf{case}\ x\ (\ell \Rightarrow P_\ell)_{\ell \in L} :: (x : \&\{\ell : A_\ell\}_{\ell \in L})}\;\& R \qquad \frac{\Omega, x{:}A_k \vdash Q :: (z : C)}{\Omega, x{:}\&\{\ell : A_\ell\}_{\ell \in L} \vdash (x.k \;;\; Q) :: (z : C)}\;\& L$$

$$\frac{}{\cdot \vdash (\mathsf{close}\ x) :: (x : \mathbf{1})}\;\mathbf{1}R \qquad \frac{\Omega \vdash Q :: (z : C)}{\Omega, x{:}\mathbf{1} \vdash (\mathsf{wait}\ x \;;\; Q) :: (z : C)}\;\mathbf{1}L$$

$$\frac{\Omega \vdash P :: (x : B)}{\Omega, y{:}A \vdash (\mathsf{send}\ x\ y \;;\; P) :: (x : A \otimes B)}\;\otimes R \qquad \frac{\Omega, y{:}A, x{:}B \vdash Q :: (z : C)}{\Omega, x{:}A \otimes B \vdash (y \leftarrow \mathsf{recv}\ x \;;\; Q) :: (z : C)}\;\otimes L$$

$$\frac{\Omega, y{:}A \vdash P :: (x : B)}{\Omega \vdash (y \leftarrow \mathsf{recv}\ x \;;\; P) :: (x : A \multimap B)}\;\multimap R \qquad \frac{\Omega, x{:}B \vdash Q :: (z : C)}{\Omega, x{:}A \multimap B, y{:}A \vdash (\mathsf{send}\ x\ y \;;\; Q) :: (z : C)}\;\multimap L$$

$$\frac{(\Omega' \vdash f = P_f :: (x : A)) \in \Sigma \quad \Omega, x{:}A \vdash Q :: (z : C)}{\Omega, \Omega' \vdash (x \leftarrow f \leftarrow \Omega' \;;\; Q) :: (z : C)}\;\mathsf{def}$$

Fig. 3. Basic Typing Rules

The dual of internal choice is *external choice* $\&\{\ell : A_\ell\}_{\ell \in L}$, which just reverses the role of provider and client and reuses the same process notation. It is the n-ary labeled generalization of the linear logic connective $A \,\&\, B$.

2.2 Termination

The type $\mathbf{1}$, the multiplicative unit of linear logic, represents termination of a process, which (due to linearity) is not allowed to use any channels.

$$\frac{}{\cdot \vdash \mathsf{close}\ x :: (x : \mathbf{1})}\;\mathbf{1}R \qquad \frac{\Omega \vdash Q :: (z : C)}{\Omega, x{:}\mathbf{1} \vdash (\mathsf{wait}\ x \;;\; Q) :: (z : C)}\;\mathbf{1}L$$

Operationally, a client has to wait for the corresponding closing message, which has no continuation since the provider terminates.

$$\begin{array}{lll} (\mathbf{1}S) & \mathsf{proc}(c, t, \mathsf{close}\ c) & \mapsto \quad \mathsf{msg}(c, t, \mathsf{close}\ c) \\ (\mathbf{1}C) & \mathsf{msg}(c, t, \mathsf{close}\ c), \mathsf{proc}(d, t, \mathsf{wait}\ c \;;\; Q) & \mapsto \quad \mathsf{proc}(d, t, Q) \end{array}$$

2.3 Forwarding

A process $x \leftarrow y$ *identifies* the channels x and y so that any further communication along either x or y will be along the unified channel. Its typing rule corresponds to the logical rule of *identity*.

$$\frac{}{y : A \vdash (x \leftarrow y) :: (x : A)}\;\mathsf{id}$$

We have already seen this form in the continuations of message objects. Operationally, the intuition is realized by *forwarding*: a process $c \leftarrow d$ *forwards* any message M that arrives along d to c and

$$\begin{array}{lll}
(\mathsf{cut}C) & \mathsf{proc}(c, t, x{:}A \leftarrow P \;;\; Q) \;\mapsto\; \mathsf{proc}(a, t, [a/x]P), \mathsf{proc}(c, t, [a/x]Q) & (a \text{ fresh}) \\
(\mathsf{def}C) & \mathsf{proc}(c, t, x \leftarrow f \leftarrow \overline{e} \;;\; Q) \;\mapsto\; \mathsf{proc}(a, t, [a/x, \overline{e}/\Omega_f]P_f), \mathsf{proc}(c, t, [a/x]Q) & (a \text{ fresh}) \\
(\mathsf{id}^+C) & \mathsf{msg}(d, t, M), \mathsf{proc}(c, s, c \leftarrow d) \;\mapsto\; \mathsf{msg}(c, t, [c/d]M) & (t \geq s) \\
(\mathsf{id}^-C) & \mathsf{proc}(c, s, c \leftarrow d), \mathsf{msg}(e, t, M(c)) \;\mapsto\; \mathsf{msg}(e, t, [d/c]M(c)) & (s \leq t) \\
(\oplus S) & \mathsf{proc}(c, t, c.k \;;\; P) \;\mapsto\; \mathsf{proc}(c', t, [c'/c]P), \mathsf{msg}(c, t, c.k \;;\; c \leftarrow c') & (c' \text{ fresh}) \\
(\oplus C) & \mathsf{msg}(c, t, c.k \;;\; c \leftarrow c'), \mathsf{proc}(d, t, \mathsf{case}\ c\ (\ell \Rightarrow Q_\ell)_{\ell \in L}) \;\mapsto\; \mathsf{proc}(d, t, [c'/c]Q_k) & \\
(\& S) & \mathsf{proc}(d, t, c.k \;;\; Q) \;\mapsto\; \mathsf{msg}(c', t, c.k \;;\; c' \leftarrow c), \mathsf{proc}(d, t, [c'/c]Q) & (c' \text{ fresh}) \\
(\& C) & \mathsf{proc}(c, t, \mathsf{case}\ c\ (\ell \Rightarrow Q_\ell)_{\ell \in L}), \mathsf{msg}(c', t, c.k \;;\; c' \leftarrow c) \;\mapsto\; \mathsf{proc}(c', t, [c'/c]Q_k) & \\
(\mathbf{1}S) & \mathsf{proc}(c, t, \mathsf{close}\ c) \;\mapsto\; \mathsf{msg}(c, t, \mathsf{close}\ c) & \\
(\mathbf{1}C) & \mathsf{msg}(c, t, \mathsf{close}\ c), \mathsf{proc}(d, t, \mathsf{wait}\ c \;;\; Q) \;\mapsto\; \mathsf{proc}(d, t, Q) & \\
(\otimes S) & \mathsf{proc}(c, t, \mathsf{send}\ c\ d \;;\; P) \;\mapsto\; \mathsf{proc}(c', t, [c'/c]P), \mathsf{msg}(c, t, \mathsf{send}\ c\ d \;;\; c \leftarrow c') & (c' \text{ fresh}) \\
(\otimes C) & \mathsf{msg}(c, t, \mathsf{send}\ c\ d \;;\; c \leftarrow c'), \mathsf{proc}(e, t, x \leftarrow \mathsf{recv}\ c \;;\; Q) \;\mapsto\; \mathsf{proc}(e, t, [c', d/c, x]Q) & \\
(\multimap S) & \mathsf{proc}(e, t, \mathsf{send}\ c\ d \;;\; Q) \;\mapsto\; \mathsf{msg}(c', t, \mathsf{send}\ c\ d \;;\; c' \leftarrow c), \mathsf{proc}(e, t, [c'/c]Q) & (c' \text{ fresh}) \\
(\multimap C) & \mathsf{proc}(c, t, x \leftarrow \mathsf{recv}\ x \;;\; P), \mathsf{msg}(c', t, \mathsf{send}\ c\ d \;;\; c' \leftarrow c) \;\mapsto\; \mathsf{proc}(c', t, [c', d/c, x]P) &
\end{array}$$

Fig. 4. Basic Operational Semantics

vice versa. Because channels are used linearly the forwarding process can then terminate, making sure to apply the proper renaming. The corresponding rules of operational semantics are as follows.

$$\begin{array}{llll}
(\mathsf{id}^+C) & \mathsf{msg}(d, t, M), \mathsf{proc}(c, s, c \leftarrow d) & \mapsto & \mathsf{msg}(c, t, [c/d]M) \quad (t \geq s) \\
(\mathsf{id}^-C) & \mathsf{proc}(c, s, c \leftarrow d), \mathsf{msg}(e, t, M(c)) & \mapsto & \mathsf{msg}(e, t, [d/c]M(c)) \quad (s \leq t)
\end{array}$$

In the last transition, we write $M(c)$ to indicate that c must occur in M, which implies that this message is the sole client of c. In anticipation of the extension by temporal operators, we do not require the time of the message and the forwarding process to be identical, but just that the forwarding process is ready *before* the message arrives.

2.4 Process Definitions

Process definitions have the form $\Omega \vdash f = P :: (x : A)$ where f is the name of the process and P its definition. All definitions are collected in a fixed global signature Σ. We require that $\Omega \vdash P :: (x : A)$ for every definition, which allows the definitions to be mutually recursive. For readability of the examples, we break a definition into two declarations, one providing the type and the other the process definition binding the variables x and those in Ω (generally omitting their types):

$$\begin{array}{c}
\Omega \vdash f :: (x : A) \\
x \leftarrow f \leftarrow \Omega = P
\end{array}$$

A new instance of a defined process f can be spawned with the expression

$$x \leftarrow f \leftarrow \overline{y} \;;\; Q$$

where $\overline{y}$ is a sequence of variables matching the antecedents Ω. The newly spawned process will use all variables in $\overline{y}$ and provide x to the continuation Q. The operational semantics is defined by

$$(\mathsf{def}C) \quad \mathsf{proc}(c, t, x \leftarrow f \leftarrow \overline{e} \;;\; Q) \;\mapsto\; \mathsf{proc}(a, t, [a/x, \overline{e}/\Omega]P), \mathsf{proc}(c, t, [a/x]Q) \quad (a \text{ fresh})$$

Here we write $\overline{e}/\Omega$ to denote substitution of the channels in $\overline{e}$ for the corresponding variables in Ω.

Sometimes a process invocation is a *tail call*, written without a continuation as $x \leftarrow f \leftarrow \overline{y}$. This is a short-hand for $x' \leftarrow f \leftarrow \overline{y} \;;\; x \leftarrow x'$ for a fresh variable x', that is, we create a fresh channel and immediately identify it with x (although it is generally implemented more efficiently).

2.5 Recursive Types

Session types can be naturally extended to include recursive types. For this purpose we allow (possibly mutually recursive) type definitions $X = A$ in the signature, where we require A to be *contractive* [Gay and Hole 2005]. This means here that A should not itself be a type name. Our type definitions are *equi-recursive* so we can silently replace X by A during type checking, and no explicit rules for recursive types are needed.

As a first example, consider a stream of bits (introduced in Section 1) defined recursively as

$$\mathrm{bits} = \oplus\{\mathrm{b0} : \mathrm{bits}, \mathrm{b1} : \mathrm{bits}, \$: \mathbf{1}\}$$

When considering bits as representing natural numbers, we think of the least significant bit being sent first. For example, a process *six* sending the number $6 = (110)_2$ would be

$$\cdot \vdash \mathit{six} :: (x : \mathrm{bits})$$
$$x \leftarrow \mathit{six} = x.\mathrm{b0} \; ; x.\mathrm{b1} \; ; x.\mathrm{b1} \; ; x.\$ \; ; \mathrm{close}\ x$$

Executing $\mathrm{proc}(c_0, 0, c_0 \leftarrow \mathit{six})$ yields (with some fresh channels $c_1, \ldots, c_4$)

$$\begin{array}{rcl}\mathrm{proc}(c_0, 0, c_0 \leftarrow \mathrm{six}) & \mapsto^* & \mathrm{msg}(c_4, 0, \mathrm{close}\ c_4), \\ & & \mathrm{msg}(c_3, 0, c_3.\$ \; ; c_3 \leftarrow c_4), \\ & & \mathrm{msg}(c_2, 0, c_2.\mathrm{b1} \; ; c_2 \leftarrow c_3), \\ & & \mathrm{msg}(c_1, 0, c_1.\mathrm{b1} \; ; c_1 \leftarrow c_2), \\ & & \mathrm{msg}(c_0, 0, c_0.\mathrm{b0} \; ; c_0 \leftarrow c_1)\end{array}$$

As a first example of a recursive process definition, consider one that just copies the incoming bits.

```
y : bits ⊢ copy :: (x : bits)
x ← copy ← y =
   case y  ( b0 ⇒ x.b0 ; x ← copy ← y        % received b0 on y, send b0 on x, recurse
           | b1 ⇒ x.b1 ; x ← copy ← y        % received b1 on y, send b1 on x, recurse
           | $ ⇒ x.$ ; wait y ; close x )    % received $ on y, send $ on x, wait on y, close x
```

The process neg mentioned in the introduction would just swap the occurrences of $x.\mathrm{b0}$ and $x.\mathrm{b1}$. We see here an occurrence of a (recursive) *tail call* to *copy*.

A last example in this section: to increment a bit stream we turn b0 to b1 but then forward the remaining bits unchanged ($x \leftarrow y$), or we turn b1 to b0 but then increment the remaining stream ($x \leftarrow \mathit{plus1} \leftarrow y$) to capture the effect of the carry bit.

```
y : bits ⊢ plus1 :: (x : bits)
x ← plus1 ← y =
   case y  ( b0 ⇒ x.b1 ; x ← y
           | b1 ⇒ x.b0 ; x ← plus1 ← y
           | $ ⇒ x.$ ; wait y ; close x )
```

3 THE TEMPORAL MODALITY NEXT ($\bigcirc A$)

In this section we introduce *actual cost* by explicitly advancing time. Remarkably, all the rules we have presented so far remain literally unchanged. As mentioned, they correspond to the cost-free fragment of the language in which time never advances. In addition, we have a new type construct $\bigcirc A$ (read: *next A*) with a corresponding process construct (delay ; P), which advances time by one unit. In the corresponding typing rule

$$\frac{\Omega \vdash P :: (x : A)}{\bigcirc\Omega \vdash (\mathrm{delay} \; ; P) :: (x : \bigcirc A)} \; \bigcirc LR$$

we abbreviate $y_1{:}\bigcirc B_1, \ldots, y_m{:}\bigcirc B_m$ by $\bigcirc(y_1{:}B_1, \ldots, y_m{:}B_m)$. Intuitively, when $(\mathsf{delay}\ ;\ P)$ idles, time advances on *all* channels connected to P. Computationally, we delay the process for one time unit without any external interactions.

$$(\bigcirc C) \quad \mathsf{proc}(c, t, \mathsf{delay}\ ;\ P) \ \mapsto\ \mathsf{proc}(c, t+1, P)$$

There is a subtle point about forwarding: A process $\mathsf{proc}(c, t, c \leftarrow d)$ may be ready to forward a message *before* a client reaches time t while in all other rules the times must match exactly. We can avoid this mismatch by transforming uses of forwarding $x \leftarrow y$ at type $\bigcirc^n S$ where $S \neq \bigcirc(-)$ to $(\mathsf{delay}^n\ ;\ x \leftarrow y)$. In this discussion we have used the following notation which will be useful later:

$$\begin{array}{lcl@{\qquad\qquad}lcl}
\bigcirc^0 A & = & A & \mathsf{delay}^0\ ;\ P & = & P \\
\bigcirc^{n+1} A & = & \bigcirc\bigcirc^n A & \mathsf{delay}^{n+1}\ ;\ P & = & \mathsf{delay}\ ;\ \mathsf{delay}^n\ ;\ P
\end{array}$$

3.1 Modeling a Cost Semantics

Our system allows us to represent a variety of different abstract cost models in a straightforward way. We will mostly use two different abstract cost models. In the first, called $\mathcal{R}$, we assign unit cost to every receive (or wait) action while all other operations remain cost-free. We may be interested in this since receiving a message is the only blocking operation in the asynchronous semantics. A second one, called $\mathcal{RS}$ and considered in Section 7, assigns unit cost to both send and receive actions.

To capture $\mathcal{R}$ we take a source program and insert a delay operation before the continuation of every receive. We write this delay as tick in order to remind the reader that it arises systematically from the cost model and is never written by the programmer. In all other respects, tick is just a synonym for delay.

For example, the earlier copy process would become

$$\mathsf{bits} = \oplus\{\mathsf{b0} : \mathsf{bits}, \mathsf{b1} : \mathsf{bits}, \$: \mathbf{1}\}$$

$$\begin{array}{l}
y : \mathsf{bits} \vdash \mathit{copy} :: (x : \mathsf{bits}) \qquad\qquad \%\ \textit{No longer correct!} \\
x \leftarrow \mathit{copy} \leftarrow y = \\
\quad \mathsf{case}\ y\ \ (\,\mathsf{b0} \Rightarrow \mathsf{tick}\ ;\ x.\mathsf{b0}\ ;\ x \leftarrow \mathit{copy} \leftarrow y \\
\qquad\qquad\ \mid \mathsf{b1} \Rightarrow \mathsf{tick}\ ;\ x.\mathsf{b1}\ ;\ x \leftarrow \mathit{copy} \leftarrow y \\
\qquad\qquad\ \mid \$ \Rightarrow \mathsf{tick}\ ;\ x.\$\ ;\ \mathsf{wait}\ y\ ;\ \mathsf{tick}\ ;\ \mathsf{close}\ x\,)
\end{array}$$

As indicated in the comment, the type of *copy* is now no longer correct because the bits that arrive along y are delayed by one unit before they are sent along x. We can observe this concretely by starting to type-check the first branch

$$\begin{array}{l}
y : \mathsf{bits} \vdash \mathit{copy} :: (x : \mathsf{bits}) \\
x \leftarrow \mathit{copy} \leftarrow y = \\
\quad \mathsf{case}\ y\ \ (\,\mathsf{b0} \Rightarrow \qquad\qquad\qquad\qquad \%\ y : \mathsf{bits} \vdash x : \mathsf{bits} \\
\qquad\qquad\qquad\quad \mathsf{tick}\ ;\ \ldots)
\end{array}$$

We see that the delay tick does not type-check, because neither x nor y have a type of the form $\bigcirc(-)$. We need to redefine the type bits so that the continuation type after every label is delayed by one, anticipating the time it takes to receive the label b0, b1, or \$. Similarly, we capture in the type of *copy* that its *latency* is one unit of time.

$$\mathsf{bits} = \oplus\{\mathsf{b0} : \bigcirc\mathsf{bits}, \mathsf{b1} : \bigcirc\mathsf{bits}, \$: \bigcirc\mathbf{1}\}$$

$$y : \mathsf{bits} \vdash \mathit{copy} :: (x : \bigcirc\mathsf{bits})$$

With these declarations, we can now type-check the definition of *copy*. We show the intermediate type of the used and provided channels after each interaction.

```
x ← copy ← y =
  case y  ( b0 ⇒                          % y : ◯bits ⊢ x : ◯bits
               tick ;                     % y : bits ⊢ x : bits
               x.b0 ;                     % y : bits ⊢ x : ◯bits
               x ← copy ← y               % well-typed by type of copy
          | b1 ⇒                          % y : ◯bits ⊢ x : ◯bits
               tick ;                     % y : bits ⊢ x : bits
               x.b1 ;                     % y : bits ⊢ x : ◯bits
               x ← copy ← y
          | $ ⇒                           % y : ◯1 ⊢ x : ◯bits
               tick ;                     % y : 1 ⊢ x : bits
               x.$ ;                      % y : 1 ⊢ x : ◯1
               wait y ;                   % · ⊢ x : ◯1
               tick ;                     % · ⊢ x : 1
               close x )
```

Armed with this experience, we now consider the increment process *plus1*. Again, we expect the latency of the increment to be one unit of time. Since we are interested in detailed type-checking, we show the transformed program, with a delay tick after each receive.

$$\mathsf{bits} = \oplus\{\mathsf{b0} : \bigcirc\mathsf{bits}, \mathsf{b1} : \bigcirc\mathsf{bits}, \$: \bigcirc\mathbf{1}\}$$

```
y : bits ⊢ plus1 :: (x : ◯bits)
x ← plus1 ← y =
  case y  ( b0 ⇒ tick ; x.b1 ; x ← y              % type error here!
          | b1 ⇒ tick ; x.b0 ; x ← plus1 ← y
          | $ ⇒ tick ; x.$ ; wait y ; tick ; close x )
```

The branches for b1 and \$ type-check as before, but the branch for b0 does not. We make the types at the crucial point explicit:

```
x ← plus1 ← y =
  case y  ( b0 ⇒ tick ; x.b1 ;            % y : bits ⊢ x : ◯bits
               x ← y                      % ill-typed, since bits ≠ ◯bits
          | . . . )
```

The problem here is that identifying x and y removes the delay mandated by the type of *plus1*. A solution is to call *copy* to reintroduce the latency of one time unit.

```
y : bits ⊢ plus1 :: (x : ◯bits)
x ← plus1 ← y =
  case y  ( b0 ⇒ tick ; x.b1 ; x ← copy ← y
          | b1 ⇒ tick ; x.b0 ; x ← plus1 ← y
          | $ ⇒ tick ; x.$ ; wait y ; tick ; close x )
```

In order to write *plus2* as a pipeline of two increments we need to delay the second increment explicitly in the program and stipulate, in the type, that there is a latency of two.

```
y : bits ⊢ plus2 :: (x : ◯◯bits)
x ← plus2 ← y =
  z ← plus1 ← y ;               % z : ◯bits ⊢ x : ◯◯bits
  delay ;                       % z : bits ⊢ x : ◯bits
  x ← plus1 ← z
```

Programming with so many explicit delays is tedious, but fortunately we can transform a source program without all these delay operations (but explicitly temporal session types) automatically in two steps: (1) we insert the delays mandated by the cost model (here: a tick after each receive), and (2) we perform *time reconstruction* to insert the additional delays so the result is temporally well-typed or issue an error message if this is impossible (see Section 6).

3.2 The Interpretation of a Configuration

We reconsider the program to produce the number $6 = (110)_2$ under the cost model from the previous section where each receive action costs one unit of time. There are no receive operations in this program, but time reconstruction must insert a delay after each send in order to match the delays mandated by the type bits.

$$\mathsf{bits} = \oplus\{\mathsf{b0} : \bigcirc\mathsf{bits}, \mathsf{b1} : \bigcirc\mathsf{bits}, \$: \bigcirc\mathbf{1}\}$$

$$\cdot \vdash \mathit{six} :: (x : \mathsf{bits})$$
$$x \leftarrow \mathit{six} = x.\mathsf{b0}\ ;\ \mathsf{delay}\ ;\ x.\mathsf{b1}\ ;\ \mathsf{delay}\ ;\ x.\mathsf{b1}\ ;\ \mathsf{delay}\ ;\ x.\$\ ;\ \mathsf{delay}\ ;\ \mathsf{close}\ x$$

Executing $\mathsf{proc}(c_0, 0, c_0 \leftarrow \mathit{six})$ then leads to the following configuration

$$\begin{array}{l}
\mathsf{msg}(c_4, 4, \mathsf{close}\ c_4),\\
\mathsf{msg}(c_3, 3, c_3.\$\ ;\ c_3 \leftarrow c_4),\\
\mathsf{msg}(c_2, 2, c_2.\mathsf{b1}\ ;\ c_2 \leftarrow c_3),\\
\mathsf{msg}(c_1, 1, c_1.\mathsf{b1}\ ;\ c_1 \leftarrow c_2),\\
\mathsf{msg}(c_0, 0, c_0.\mathsf{b0}\ ;\ c_0 \leftarrow c_1)
\end{array}$$

These messages are at increasing times, which means any client of c_0 will have to immediately (at time 0) receive b0, then (at time 1) b1, then (at time 2) b1, etc. In other words, the time stamps on messages predict *exactly* when the message will be received. Of course, if there is a client in parallel we may never reach this state because, for example, the first b0 message along channel c_0 may be received before the continuation of the sender produces the message b1. So different configurations may be reached depending on the *scheduler* for the concurrent processes. It is also possible to give a time-synchronous semantics in which all processes proceed *in parallel* from time 0 to time 1, then from time 1 to time 2, etc.

4 THE TEMPORAL MODALITIES ALWAYS ($\Box A$) AND EVENTUALLY ($\Diamond A$)

The strength and also the weakness of the system so far is that its timing is very precise. Now consider a process *compress* that combines runs of consecutive 1's to a single 1. For example, compressing 11011100 should yield 10100. First, in the cost-free setting we might write

$$\mathsf{bits} = \oplus\{\mathsf{b0} : \mathsf{bits}, \mathsf{b1} : \mathsf{bits}, \$: \mathbf{1}\}$$

$$\begin{array}{l}
y : \mathsf{bits} \vdash \mathit{compress} :: (x : \mathsf{bits})\\
y : \mathsf{bits} \vdash \mathit{skip1s} :: (x : \mathsf{bits})
\end{array}$$

$$\begin{array}{l}
x \leftarrow \mathit{compress} \leftarrow y =\\
\quad \mathsf{case}\ y\ \ (\ \mathsf{b0} \Rightarrow x.\mathsf{b0}\ ;\ x \leftarrow \mathit{compress} \leftarrow y\\
\qquad\qquad\ \mid \mathsf{b1} \Rightarrow x.\mathsf{b1}\ ;\ x \leftarrow \mathit{skip1s} \leftarrow y\\
\qquad\qquad\ \mid \$ \Rightarrow x.\$\ ;\ \mathsf{wait}\ y\ ;\ \mathsf{close}\ x\ )
\end{array}$$

$$\begin{array}{l}
x \leftarrow \mathit{skip1s} \leftarrow y =\\
\quad \mathsf{case}\ y\ \ (\ \mathsf{b0} \Rightarrow x.\mathsf{b0}\ ;\ x \leftarrow \mathit{compress} \leftarrow y\\
\qquad\qquad\ \mid \mathsf{b1} \Rightarrow x \leftarrow \mathit{skip1s} \leftarrow y\\
\qquad\qquad\ \mid \$ \Rightarrow x.\$\ ;\ \mathsf{wait}\ y\ ;\ \mathsf{close}\ x\ )
\end{array}$$

The problem is that if we adopt the cost model $\mathcal{R}$ where every receive takes one unit of time, then this program cannot be typed. Actually worse: there is no way to insert next-time modalities into the type and additional delays into the program so that the result is well-typed. This is because if the input stream is unknown we cannot predict how long a run of 1's will be, but the length of such a run will determine the delay between sending a bit 1 and the following bit 0.

The best we can say is that after a bit 1 we will *eventually* send either a bit 0 or the end-of-stream token \$. This is the purpose of the type $\Diamond A$. We capture this timing in the type sbits (for *slow bits*).

$$\begin{aligned} &\mathrm{bits} = \oplus\{\mathrm{b0} : \bigcirc\mathrm{bits}, \mathrm{b1} : \bigcirc\mathrm{bits}, \$: \bigcirc\mathbf{1}\} \\ &\mathrm{sbits} = \oplus\{\mathrm{b0} : \bigcirc\mathrm{sbits}, \mathrm{b1} : \bigcirc\Diamond\mathrm{sbits}, \$: \bigcirc\mathbf{1}\} \end{aligned}$$

$$\begin{aligned} &y : \mathrm{bits} \vdash \mathit{compress} :: (x : \bigcirc\mathrm{sbits}) \\ &y : \mathrm{bits} \vdash \mathit{skip1s} :: (x : \bigcirc\Diamond\mathrm{sbits}) \end{aligned}$$

In the next section we introduce the process constructs and typing rules so we can revise our *compress* and *skip1s* programs so they have the right temporal semantics.

4.1 Eventually A

A process providing $\Diamond A$ promises only that it will eventually provide A. There is a somewhat subtle point here: since not every action may require time and because we do not check termination separately, $x : \Diamond A$ expresses only that *if the process providing* x *terminates* it will eventually provide A. Thus, it expresses non-determinism regarding the (abstract) *time* at which A is provided, rather than a strict liveness property. Therefore, $\Diamond A$ is somewhat weaker than one might be used to from LTL [Pnueli 1977]. When restricted to a purely logical fragment, without unrestricted recursion, the usual meaning is fully restored so we feel our terminology is justified. Imposing termination, for example along the lines of Fortier and Santocanale [2013] or Toninho et al. [2014] is an interesting item for future work but not necessary for our present purposes.

When a process offering $c : \Diamond A$ is ready, it will send a now! message along c and then continue at type A. Conversely, the client of $c : \Diamond A$ will have to be ready and waiting for the now! message to arrive along c and then continue at type A. We use ($\mathrm{when?}\ c\ ;\ Q$) for the corresponding client. These explicit constructs are a conceptual device and may not need to be part of an implementation. They also make type-checking processes entirely syntax-directed and trivially decidable.

The typing rules for now! and when? are somewhat subtle.

$$\frac{\Omega \vdash P :: (x : A)}{\Omega \vdash (\mathrm{now!}\ x\ ;\ P) :: (x : \Diamond A)}\ \Diamond R \qquad \frac{\bigcirc^*\Box\Omega' = \Omega \quad \Omega, x{:}A \vdash Q :: (z : C) \quad C = \bigcirc^*\Diamond C'}{\Omega, x{:}\Diamond A \vdash (\mathrm{when?}\ x\ ;\ Q) :: (z : C)}\ \Diamond L$$

The $\Diamond R$ rule just states that, without constraints, we can at any time decide to communicate along $x : \Diamond A$ and then continue the session at type A. The $\Diamond L$ rule expresses that the process must be ready to receive a now! message along $x : \Diamond A$, but there are two further constraints. Because the process ($\mathrm{when?}\ x\ ;\ Q$) may need to wait an indefinite period of time, the rule must make sure that communication along z and any channel in Ω can also be postponed an indefinite period of time. We write $C = \bigcirc^*\Diamond C'$ to require that C may be delayed a fixed finite number of time steps and then must be allowed to communicate at an arbitrary time in the future. Similarly, for every channel $y : B$ in Ω, B must have the form $\bigcirc^*\Box B$, where $\Box$ (as the dual of $\Diamond$) is introduced in Section 4.3.

In the operational semantics, the central restriction is that when? is ready *before* the now! message arrives so that the continuation can proceed immediately as promised by the type.

$$\begin{array}{llll} (\Diamond S) & \mathrm{proc}(c, t, \mathrm{now!}\ c\ ;\ P) & \mapsto & \mathrm{proc}(c', t, [c'/c]P), \mathrm{msg}(c, t, \mathrm{now!}\ c\ ;\ c \leftarrow c') \quad (c'\ \mathrm{fresh}) \\ (\Diamond C) & \mathrm{msg}(c, t, \mathrm{now!}\ c\ ;\ c \leftarrow c'), \mathrm{proc}(d, s, \mathrm{when?}\ c\ ;\ Q) & \mapsto & \mathrm{proc}(d, t, [c'/c]Q) \quad (t \geq s) \end{array}$$

We are now almost ready to rewrite the *compress* process in our cost model $\mathcal{R}$. First, we insert tick before all the actions that must be delayed according to our cost model. Then we insert appropriate additional delay, when?, and now! actions. While *compress* turns out to be straightforward, *skip1s* creates a difficulty after it receives a b1:

$$\begin{aligned}&\mathsf{bits} = \oplus\{\mathsf{b0} : \bigcirc\mathsf{bits}, \mathsf{b1} : \bigcirc\mathsf{bits}, \$: \bigcirc\mathbf{1}\}\\&\mathsf{sbits} = \oplus\{\mathsf{b0} : \bigcirc\mathsf{sbits}, \mathsf{b1} : \bigcirc\Diamond\mathsf{sbits}, \$: \bigcirc\mathbf{1}\}\end{aligned}$$

$$\begin{aligned}&y : \mathsf{bits} \vdash \mathit{compress} :: (x : \bigcirc\mathsf{sbits})\\&y : \mathsf{bits} \vdash \mathit{skip1s} :: (x : \bigcirc\Diamond\mathsf{sbits})\end{aligned}$$

```
x ← compress ← y =
   case y  ( b0 ⇒ tick ; x.b0 ; x ← compress ← y
           | b1 ⇒ tick ; x.b1 ; x ← skip1s ← y
           | $ ⇒ tick ; x.$ ; wait y ; tick ; close x )

x ← skip1s ← y =
   case y  ( b0 ⇒ tick ; now! x ; x.b0 ; x ← compress ← y
           | b1 ⇒ tick ;                      % y : bits ⊢ x : ◇sbits
                 x′ ← skip1s ← y ;            % x′ : ○◇sbits ⊢ x : ◇sbits
                 x ← idle ← x′                % with x′ : ○◇sbits ⊢ idle :: (x : ◇sbits)
           | $ ⇒ tick ; now! x ; x.$ ; wait y ; tick ; close x )
```

At the point where we would like to call *skip1s* recursively, we have

$$\begin{aligned}&y : \mathsf{bits} \vdash x : \Diamond\mathsf{sbits}\\&\text{but}\quad y : \mathsf{bits} \vdash \mathit{skip1s} :: (x : \bigcirc\Diamond\mathsf{sbits})\end{aligned}$$

which prevents a tail call since $\bigcirc\Diamond\mathsf{sbits} \neq \Diamond\mathsf{sbits}$. Instead we call *skip1s* to obtain a new channel x' and then use another process called *idle* to go from $x' : \bigcirc\Diamond\mathsf{sbits}$ to $x : \Diamond\mathsf{sbits}$. Intuitively, it should be possible to implement such an idling process: $x : \Diamond\mathsf{sbits}$ expresses *at some time in the future, including possibly right now* while $x' : \bigcirc\Diamond\mathsf{sbits}$ says *at some time in the future, but not right now.*

To type the idling process, we need to generalize the $\bigcirc LR$ rule to account for the interactions of $\bigcirc A$ with $\Box A$ and $\Diamond A$. After all, they speak about the same underlying model of time.

4.2 Interactions of $\bigcirc A$ and $\Diamond A$

Recall the left/right rule for $\bigcirc$:

$$\frac{\Omega \vdash P :: (x : A)}{\bigcirc\Omega \vdash (\mathsf{delay} \ ; P) :: (x : \bigcirc A)}\ \bigcirc LR$$

If the succedent were $x : \Diamond A$ instead of $x : \bigcirc A$, we should still be able to delay since we can freely choose when to interact along x. We could capture this in the following rule (superseded later by a more general form of $\bigcirc LR$):

$$\frac{\Omega \vdash P :: (x : \Diamond A)}{\bigcirc\Omega \vdash (\mathsf{delay} \ ; P) :: (x : \Diamond A)}\ \bigcirc\Diamond$$

We keep $\Diamond A$ as the type of x since we retain the full flexibility of using x at any time in the future after the initial delay. We will generalize the rule once more in the next section to account for interactions with $\Box A$.

With this, we can define and type the idling process parametrically over A:

$$\begin{aligned}&x' : \bigcirc\Diamond A \vdash \mathit{idle} :: (x : \Diamond A)\\&x \leftarrow \mathit{idle} \leftarrow x' = \mathsf{delay} \ ; x \leftarrow x'\end{aligned}$$

This turns out to be an example of subtyping (see Section 6.1), which means that the programmer actually will not have to explicitly define or even reference an idling process. The programmer simply writes the original *skip1s* process (without referencing the *idle* process) and our subtyping algorithm will use the appropriate rule to typecheck it successfully.

4.3 Always A

We now turn our attention to the last temporal modality, $\Box A$, which is dual to $\Diamond A$. If a process P provides $x : \Box A$ it means it is ready to receive a now! message along x at any point in the future. In analogy with the typing rules for $\Diamond A$, but flipped to the other side of the sequent, we obtain

$$\frac{\bigcirc^{*}\Box\Omega' = \Omega \quad \Omega \vdash P :: (x : A)}{\Omega \vdash (\mathsf{when?}\ x\ ;\ P) :: (x : \Box A)}\ \Box R \qquad \frac{\Omega, x{:}A \vdash Q :: (z : C)}{\Omega, x{:}\Box A \vdash (\mathsf{now!}\ x\ ;\ Q) :: (z : C)}\ \Box L$$

The operational rules just reverse the role of provider and client from the rules for $\Diamond A$.

$$\begin{array}{llcll}
(\Box S) & \mathsf{proc}(d, t, \mathsf{now!}\ c\ ;\ Q) & \mapsto & \mathsf{msg}(c', t, \mathsf{now!}\ c\ ;\ c' \leftarrow c), \mathsf{proc}(d, t, [c'/c]Q) & (c' \text{ fresh}) \\
(\Box C) & \mathsf{proc}(c, s, \mathsf{when?}\ c\ ;\ P), \mathsf{msg}(c', t, \mathsf{now!}\ c\ ;\ c' \leftarrow c) & \mapsto & \mathsf{proc}(c', t, [c'/c]P) & (s \leq t)
\end{array}$$

As an example for the use of $\Box A$, and also to introduce a new kind of example, we specify and implement a counter process that can receive inc and val messages. When receiving an inc it will increment its internally maintained counter, when receiving val it will produce a finite bit stream representing the current value of the counter. In the cost-free setting we have the type

$$\begin{aligned}
\mathsf{bits} &= \oplus\{\mathsf{b0} : \mathsf{bits}, \mathsf{b1} : \mathsf{bits}, \$: \mathbf{1}\} \\
\mathsf{ctr} &= \&\{\mathsf{inc} : \mathsf{ctr}, \mathsf{val} : \mathsf{bits}\}
\end{aligned}$$

A counter is implemented by a chain of processes, each holding one bit (either *bit0* or *bit1*) or signaling the end of the chain (*empty*). For this purpose we implement three processes:

$$\begin{aligned}
&d : \mathsf{ctr} \vdash \mathit{bit0} :: (c : \mathsf{ctr}) \\
&d : \mathsf{ctr} \vdash \mathit{bit1} :: (c : \mathsf{ctr}) \\
&\cdot \vdash \mathit{empty} :: (c : \mathsf{ctr})
\end{aligned}$$

```
c ← bit0 ← d =
   case c  ( inc ⇒ c ← bit1 ← d              % increment by continuing as bit1
           | val ⇒ c.b0 ; d.val ; c ← d )    % send b0 on c, send val on d, identify c and d

c ← bit1 ← d =
   case c  ( inc ⇒ d.inc ; c ← bit0 ← d      % send inc (carry) on d, continue as bit1
           | val ⇒ c.b1 ; d.val ; c ← d )    % send b1 on c, send val on d, identify c and d

c ← empty =
   case c  ( inc ⇒ e ← empty ;               % spawn a new empty process with channel e
                   c ← bit1 ← e              % continue as bit1
           | val ⇒ c.$ ; close c )           % send $ on c and close c
```

Using our standard cost model $\mathcal{R}$ we notice a problem: the *carry bit* (the d.inc message sent in the *bit1* process) is sent only on every other increment received because *bit0* continues as *bit1 without* a carry, and *bit1* continues as *bit0 with* a carry. So it will actually take 2^k increments received at the lowest bit of the counter (which represents the interface to the client) before an increment reaches the kth process in the chain. This is not a constant number, so we cannot characterize the behavior exactly using only the next time modality. Instead, we say, from a certain point on, a counter is always ready to receive either an inc or val message.

$$\begin{aligned}
\mathsf{bits} &= \oplus\{\mathsf{b0} : \bigcirc\mathsf{bits}, \mathsf{b1} : \bigcirc\mathsf{bits}, \$: \bigcirc\mathbf{1}\} \\
\mathsf{ctr} &= \Box\&\{\mathsf{inc} : \bigcirc\mathsf{ctr}, \mathsf{val} : \bigcirc\mathsf{bits}\}
\end{aligned}$$

In the program, we have ticks mandated by our cost model and some additional delay, when?, and now! actions to satisfy the stated types. The two marked lines may look incorrect, but are valid based on the generalization of the $\bigcirc LR$ rule in Section 4.4.

```
d : ○ctr ⊢ bit0 :: (c : ctr)
d : ctr ⊢ bit1 :: (c : ctr)
· ⊢ empty :: (c : ctr)

c ← bit0 ← d =
  when? c ;                          % d : ○ctr ⊢ c : &{...}
  case c  ( inc ⇒ tick ;             % d : ctr ⊢ c : ctr
                  c ← bit1 ← d
          | val ⇒ tick ;             % d : ctr ⊢ c : bits
                  c.b0 ;             % d : ctr ⊢ c : ○bits
                  now! d ; d.val ;   % d : ○bits ⊢ c : ○bits
                  c ← d )

c ← bit1 ← d =
  when? c ;                          % d : ctr ⊢ c : &{...}
  case c  ( inc ⇒ tick ;             % d : ctr ⊢ c : ctr        (see Section 4.4)
                  now! d ; d.inc ;   % d : ○ctr ⊢ c : ctr
                  c ← bit0 ← d
          | val ⇒ tick ;             % d : ctr ⊢ c : bit        (see Section 4.4)
                  c.b1 ;             % d : ctr ⊢ c : ○bits
                  now! d ; d.val ;   % d : ○bits ⊢ c : ○bits
                  c ← d )

c ← empty =
  when? c ;                          % · ⊢ c : &{...}
  case c  ( inc ⇒ tick ;             % · ⊢ c : ctr
                  e ← empty ;        % e : ctr ⊢ c : ctr
                  c ← bit1 ← e
          | val ⇒ tick ; c.$ ;       % · ⊢ c : ○1
                  delay ; close c )
```

4.4 Interactions Between Temporal Modalities

Just as $\bigcirc A$ and $\Diamond A$ interacted in the rules since their semantics is based on the same underlying notion of time, so do $\bigcirc A$ and $\Box A$. If we execute a delay, we can allow any channel of type $\Box A$ that we use and leave its type unchanged because we are not obligated to communicate along it at any particular time. It is a little awkward to formulate this because among the channels used there may be some of type $\bigcirc B$ and some of type $\Box B$.

$$\frac{\Box\Omega, \Omega' \vdash P :: (x : A)}{\Box\Omega, \bigcirc\Omega' \vdash (\mathsf{delay} \; ; P) :: (x : \bigcirc A)} \; \bigcirc$$

In the example of *bit1* at the end of the previous section, we have already seen two lines where this generalization was crucial, observing that ctr $= \Box\&\{\ldots\}$.

But even this rule does not cover all possibilities, because the channel x could be of type $\Diamond A$. We introduce a new notation, writing $[A]_L^{-1}$ and $[A]_R^{-1}$ on types and extend it to contexts. Depending on one's point of view, this can be seen as stepping forward or backward by one unit of time.

$$\begin{array}{lcl@{\qquad}lcl@{\qquad}lcl}
[\bigcirc A]_L^{-1} & = & A & [\bigcirc A]_R^{-1} & = & A & [x : A]_L^{-1} & = & x : [A]_L^{-1} \\
[\Box A]_L^{-1} & = & \Box A & [\Box A]_R^{-1} & = & \mathit{undefined} & [x : A]_R^{-1} & = & x : [A]_R^{-1} \\
[\Diamond A]_L^{-1} & = & \mathit{undefined} & [\Diamond A]_R^{-1} & = & \Diamond A & [\cdot]_L^{-1} & = & \cdot \\
[S]_L^{-1} & = & \mathit{undefined} & [S]_R^{-1} & = & \mathit{undefined} & [\Omega, \Omega']_L^{-1} & = & [\Omega]_L^{-1}, [\Omega']_L^{-1}
\end{array}$$

$$\frac{[\Omega]_L^{-1} \vdash P :: [x : A]_R^{-1}}{\Omega \vdash (\mathsf{delay}\ ; P) :: (x : A)}\ \bigcirc LR \qquad \frac{}{\bigcirc^* \Box A\ \mathsf{delayed}^{\Box}} \qquad \frac{}{\bigcirc^* \Diamond A\ \mathsf{delayed}^{\Diamond}}$$

$$\frac{\Omega \vdash P :: (x : A)}{\Omega \vdash (\mathsf{now!}\ x\ ; P) :: (x : \Diamond A)}\ \Diamond R \qquad \frac{\Omega\ \mathsf{delayed}^{\Box} \quad \Omega, x{:}A \vdash Q :: (z : C) \quad C\ \mathsf{delayed}^{\Diamond}}{\Omega, x{:}\Diamond A \vdash (\mathsf{when?}\ x\ ; Q) :: (z : C)}\ \Diamond L$$

$$\frac{\Omega\ \mathsf{delayed}^{\Box} \quad \Omega \vdash P :: (x : A)}{\Omega \vdash (\mathsf{when?}\ x\ ; P) :: (x : \Box A)}\ \Box R \qquad \frac{\Omega, x{:}A \vdash Q :: (z : C)}{\Omega, x{:}\Box A \vdash (\mathsf{now!}\ x\ ; Q) :: (z : C)}\ \Box L$$

Fig. 5. Explicit Temporal Typing Rules

Here, S stands for any basic session type constructor as in Figure 1. We use this notation in the general rule $\bigcirc LR$ which can be found in Figure 5 together with the final set of rules for $\Box A$ and $\Diamond A$. In conjunction with the rules in Figure 3 this completes the system of temporal session types where all temporal actions are explicit. The rule $\bigcirc LR$ only applies if both $[\Omega]_L^{-1}$ and $[x : A]_R^{-1}$ are defined.

We call a type A *patient* if it does not force communication along a channel $x : A$ at any particular point in time. Because the direction of communication is reversed between the two sides of a sequent, a type A is patient if it has the form $\bigcirc^* \Box A'$ if it is among the antecedents, and $\bigcirc^* \Diamond A'$ if it is in the succedent. We write $A\ \mathsf{delayed}^{\Box}$ and $A\ \mathsf{delayed}^{\Diamond}$ and extend it to contexts $\Omega\ \mathsf{delayed}^{\Box}$ if for every declaration $(x : A) \in \Omega$, we have $A\ \mathsf{delayed}^{\Box}$.

5 PRESERVATION AND PROGRESS

The main theorems that exhibit the deep connection between our type system and the timed operational semantics are the usual *type preservation* and *progress*, sometimes called *session fidelity* and *deadlock freedom*, respectively. Compared to other recent treatments of linear session types [Balzer and Pfenning 2017; Pfenning and Griffith 2015], new challenges are presented by abstract time and the temporal modalities.

5.1 Configuration Typing

A key question is how we type configurations $\mathcal{C}$. Configurations consist of multiple processes and messages, so they both *use* and *provide* a collection of channels. And even though we treat a configuration as a multiset, typing imposes a partial order on the processes and messages where a provider of a channel appears to the left of its client.

$$\text{Configuration} \quad \mathcal{C} \quad ::= \quad \cdot \mid \mathcal{C}\ \mathcal{C}' \mid \mathsf{proc}(c, t, P) \mid \mathsf{msg}(c, t, M)$$

We say $\mathsf{proc}(c, t, P)$ and $\mathsf{msg}(c, t, M)$ *provide* c. We stipulate that no two distinct processes or messages in a configuration provide the same channel c. Also recall that messages M are simply processes of a particular form and are typed as such. We can read off the possible messages (of which there is one for each type constructor) from the operational semantics. They are summarized here for completeness.

$$M \quad ::= \quad (c.k\ ; c \leftarrow c') \mid (c.k\ ; c' \leftarrow c) \mid \mathsf{close}\ c \mid (\mathsf{send}\ c\ d\ ; c' \leftarrow c) \mid (\mathsf{send}\ c\ d\ ; c \leftarrow c')$$

The typing judgment has the form $\Omega' \vDash \mathcal{C} :: \Omega$ meaning that if composed with a configuration that provides Ω', the result will provide Ω.

$$\frac{}{\Omega \vDash (\cdot) :: \Omega}\ \mathsf{empty} \qquad \frac{\Omega_0 \vDash \mathcal{C}_1 :: \Omega_1 \quad \Omega_1 \vDash \mathcal{C}_2 :: \Omega_2}{\Omega_0 \vDash (\mathcal{C}_1\ \mathcal{C}_2) :: \Omega_2}\ \mathsf{compose}$$

To type processes and messages, we begin by considering *preservation*: we would like to achieve that if $\Omega' \vDash C :: \Omega$ and $C \mapsto C'$ then still $\Omega' \vDash C' :: \Omega$. Without the temporal modalities, this is guaranteed by the design of the sequent calculus: the right and left rules match just so that cut reduction (which is the basis for reduction in the operational semantics) leads to a well-typed deduction. The key here is what happens with time. Consider the special case

$$\frac{\Omega \vdash P :: A}{\bigcirc\Omega \vdash (\mathsf{delay}\ ; P) :: (x : \bigcirc A)}\ \bigcirc LR \qquad \mathsf{proc}(c, t, \mathsf{delay}\ ; P) \ \mapsto\ \mathsf{proc}(c, t+1, P)$$

Note that, inevitably, the type of the channel c changes in the transition, from $c : \bigcirc A$ to $c : A$ and similarly for all channels used by P. So if in $\mathsf{proc}(c, t, Q)$ we were to use the type of Q as the type of the semantic process object, preservation would fail. But while the type changes from $\bigcirc A$ to A, *time* also advances from t to $t + 1$. This suggests the following rule should keep the configuration type invariant:

$$\frac{\Omega \vdash P :: (c : A)}{\bigcirc^t\Omega \vDash \mathsf{proc}(c, t, P) :: (c : \bigcirc^t A)}\ \mathsf{proc}^{\bigcirc}$$

When we transition from $\mathsf{delay}\ ; P$ to P we strip one $\bigcirc$ modality from Ω and A, but because we also advance time from t to $t + 1$, the $\bigcirc$ modality is restored, keeping the interface type invariant.

When we also consider types $\Box A$ and $\Diamond A$ the situation is a little less straightforward because of their interaction with $\bigcirc$, as we have already encountered in Section 4.4. We reuse the idea of the solution, allowing the subtraction of time from a type, possibly stopping when we meet a $\Box$ or $\Diamond$.

$$\begin{array}{lcl@{\qquad}lcl}
[A]_L^{-0} & = & A & [A]_R^{-0} & = & A \\
[A]_L^{-(t+1)} & = & [[A]_L^{-t}]_L^{-1} & [A]_R^{-(t+1)} & = & [[A]_R^{-t}]_R^{-1}
\end{array}$$

This is extended to channel declarations in the obvious way. Additionally, the imprecision of $\Box A$ and $\Diamond A$ may create temporal gaps in the configuration that need to be bridged by a weak form of subtyping $A <: B$ (not to be confused with the much stronger form $A \leq B$ in Section 6.1),

$$\frac{m \leq n}{\bigcirc^m \Box A <: \bigcirc^n \Box A}\ \Box_{\mathsf{weak}} \qquad \frac{m \geq n}{\bigcirc^m \Diamond A <: \bigcirc^n \Diamond A}\ \Diamond_{\mathsf{weak}} \qquad \frac{}{A <: A}\ \mathsf{refl}$$

This relation is specified to be reflexive and clearly transitive. We extend it to contexts Ω in the obvious manner. In our final rules we also account for some channels that are not used by P or M but just passed through.

$$\frac{\Omega' <: \Omega \quad [\Omega]_L^{-t} \vdash P :: [c : A]_R^{-t} \quad A <: A'}{\Omega_0, \Omega' \vDash \mathsf{proc}(c, t, P) :: (\Omega_0, c : A')}\ \mathsf{proc} \qquad \frac{\Omega' <: \Omega \quad [\Omega]_L^{-t} \vdash M :: [c : A]_R^{-t} \quad A <: A'}{\Omega_0, \Omega' \vDash \mathsf{msg}(c, t, M) :: (\Omega_0, c : A')}\ \mathsf{msg}$$

5.2 Type Preservation

With the four rules for typing configurations (empty, compose, proc and msg), type preservation is relatively straightforward. We need some standard lemmas about being able to split a configuration and be able to move a provider (whether process or message) to the right in a typing derivation until it rests right next to its client. Regarding time shifts, we need the following properties.

Lemma 5.1 (Time Shift).

(i) *If* $[A]_L^{-t} = [B]_R^{-t}$ *and both are defined then* $A = B$.

(ii) $[[A]_L^{-t}]_L^{-s} = [A]_L^{-(t+s)}$ *and if either side is defined, the other is as well.*

(iii) $[[A]_R^{-t}]_R^{-s} = [A]_R^{-(t+s)}$ *and if either side is defined, the other is as well.*

Theorem 5.2 (Type Preservation). *If* $\Omega' \vDash C :: \Omega$ *and* $C \mapsto \mathcal{D}$ *then* $\Omega' \vDash \mathcal{D} :: \Omega$.

Proof. By case analysis on the transition rule, applying inversion to the given typing derivation, and then assembling a new derivation of $\mathcal{D}$. □

Type preservation on basic session types is a simple special case of this theorem.

5.3 Global Progress

We say a process or message is *poised* if it is trying to communicate along the channel that it provides. A poised process is comparable to a value in a sequential language. A configuration is poised if every process or message in the configuration is poised. Conceptually, this implies that the configuration is trying to communicate externally, i.e. along one of the channel it provides. The progress theorem then shows that either a configuration can take a step or it is poised. To prove this we show first that the typing derivation can be rearranged to go strictly from right to left and then proceed by induction over this particular derivation. This much is standard, even for significantly more complicated session-typed languages [Balzer and Pfenning 2017].

The question is how can we prove that processes are either at the same time (for most interactions) or that the message recipient is ready before the message arrives (for when?, now!, and some forwards)? The key insight here is in the following lemma.

Lemma 5.3 (Time Inversion).

(i) *If* $[A]_R^{-s} = [A]_L^{-t}$ *and either side starts with a basic session type constructor then* $s = t$.
(ii) *If* $[A]_L^{-t} = \Box B$ *and* $[A]_R^{-s} \neq \bigcirc(-)$ *then* $s \leq t$ *and* $[A]_R^{-s} = \Box B$.
(iii) *If* $[A]_R^{-t} = \Diamond B$ *and* $[A]_L^{-s} \neq \bigcirc(-)$ *then* $s \leq t$ *and* $[A]_L^{-s} = \Diamond B$.

Theorem 5.4 (Global Progress). *If* $\cdot \vDash C :: \Omega$ *then either*

(i) $C \mapsto C'$ *for some* C', *or*
(ii) C *is poised.*

Proof. By induction on the right-to-left typing of C so that either C is empty (and therefore poised) or $C = (\mathcal{D}\ \mathsf{proc}(c, t, P))$ or $C = (\mathcal{D}\ \mathsf{msg}(c, t, M))$. By induction hypothesis, $\mathcal{D}$ can either take a step (and then so can C), or $\mathcal{D}$ is poised. In the latter case, we analyze the cases for P and M, applying multiple steps of inversion and Lemma 5.3 to show that in each case either C can take a step or is poised. □

6 TIME RECONSTRUCTION

The process expressions introduced so far have straightforward syntax-directed typing rules. This requires the programmer to write a significant number of explicit delay, when?, and now! constructs in their code. This in turn hampers reuse: we would like to be able to provide multiple types for the same process definition so it can be used in different contexts, with different types, even under a single, fixed cost model.

In this section we introduce an implicit system which may be thought of as a *temporal refinement* of the basic session type system in Section 2. The delay, when?, and now! constructs never appear in the source, and, as before, tick is added before type-checking and never by the programmer. The rules for the new judgment $\Omega \overset{i}{\vdash} P :: (x : A)$ are shown in Figure 6; the other rules remain the same (except for def, see below). We still need an explicit rule for the tick synonym of delay which captures the cost model.

These rules are trivially sound and complete with respect to the explicit system in Section 4 because from an implicit type derivation we can read off the explicit process expression and vice versa. They are also manifestly decidable because the types in the premises are smaller than those in the conclusion, with one possible exception: In the $\bigcirc LR$ rule the premise may be equal to the

$$\frac{[\Omega]_L^{-1} \vdash^{i} P :: [x : A]_R^{-1}}{\Omega \vdash^{i} P :: (x : A)}\ \bigcirc LR \qquad \frac{[\Omega]_L^{-1} \vdash^{i} P :: [x : A]_R^{-1}}{\Omega \vdash^{i} (\mathsf{tick}\ ;\ P) :: (x : A)}\ \bigcirc LR'$$

$$\frac{\Omega \vdash^{i} P :: (x : A)}{\Omega \vdash^{i} P :: (x : \Diamond A)}\ \Diamond R \qquad \frac{\Omega\ \text{delayed}^{\Box} \quad \Omega, x{:}A \vdash^{i} Q :: (z : C) \quad C\ \text{delayed}^{\Diamond}}{\Omega, x{:}\Diamond A \vdash^{i} Q :: (z : C)}\ \Diamond L$$

$$\frac{\Omega\ \text{delayed}^{\Box} \quad \Omega \vdash^{i} P :: (x : A)}{\Omega \vdash^{i} P :: (x : \Box A)}\ \Box R \qquad \frac{\Omega, x{:}A \vdash^{i} Q :: (z : C)}{\Omega, x{:}\Box A \vdash^{i} Q :: (z : C)}\ \Box L$$

Fig. 6. Implicit Temporal Rules

$$\frac{}{A \leq A}\ \text{refl} \qquad \frac{A \leq B}{\bigcirc A \leq \bigcirc B}\ \bigcirc\bigcirc \qquad \frac{\Box A \leq B}{\Box A \leq \bigcirc B}\ \Box\bigcirc \qquad \frac{A \leq \Diamond B}{\bigcirc A \leq \Diamond B}\ \bigcirc\Diamond$$

$$\frac{\bigcirc^n \Box A \leq B}{\bigcirc^n \Box A \leq \Box B}\ \Box R \qquad \frac{A \leq B}{\Box A \leq B}\ \Box L \qquad \frac{A \leq B}{A \leq \Diamond B}\ \Diamond R \qquad \frac{A \leq \bigcirc^n \Diamond B}{\Diamond A \leq \bigcirc^n \Diamond B}\ \Diamond L$$

Fig. 7. Subtyping Rules

conclusion if neither Ω nor A contain a type of the form $\bigcirc(-)$. In this case, $B = \Box B'$ for every $y : B$ in Ω and $A = \Diamond A'$ and there P can delay by any finite number of time steps. Time reconstruction avoids such an arbitrary delay.

Our examples revealed a significant shortcoming in these rules: when calling upon a process definition, the types in the antecedent and succedent often do not match the types of the process to be spawned. For example, the process *skip1s* in Section 4.1 we have

$$\begin{aligned}&\text{bits} = \oplus\{\text{b0} : \bigcirc\text{bits}, \text{b1} : \bigcirc\text{bits}, \$: \bigcirc\mathbf{1}\}\\ &\text{sbits} = \oplus\{\text{b0} : \bigcirc\text{sbits}, \text{b1} : \bigcirc\Diamond\text{sbits}, \$: \bigcirc\mathbf{1}\}\end{aligned}$$

$$\begin{aligned}&y : \text{bits} \vdash \mathit{compress} :: (x : \bigcirc\text{sbits})\\ &y : \text{bits} \vdash \mathit{skip1s} :: (x : \bigcirc\Diamond\text{sbits})\end{aligned}$$

```
x ← skip1s ← y =
    case y  ( b1 ⇒ tick ;                % y : bits ⊢ x : ◇sbits
                    x ← skip1s ← y       % does not type-check!
            | . . . )
```

The indicated line does not type-check (neither in the explicit nor the implicit system presented so far) since the type $\bigcirc\Diamond$sbits offered by *skip1s* does not match $\Diamond$sbits. We had to write a process *idle* to account for this mismatch:

$$\begin{aligned}&x' : \bigcirc\Diamond A \vdash \mathit{idle} :: (x : \Diamond A)\\ &x \leftarrow \mathit{idle} \leftarrow x' = \mathsf{delay}\ ;\ x \leftarrow x'\end{aligned}$$

In the implicit system the version with an explicit identity *can* in fact be reconstructed:

```
x ← skip1s ← y =
    case y  ( b1 ⇒ tick ;                % y : bits ⊢ⁱ x : ◇sbits
                    x′ ← skip1s ← y      % x′ : ○◇sbits ⊢ⁱ x : ◇sbits
                                         % x′ : ◇sbits ⊢ⁱ x : ◇sbits     using rule ○LR
                    x ← x′
            | . . . )
```

6.1 Subtyping

Extrapolating from the example of *skip1s* above, we can generalize process invocations by allowing *subtyping* on all used channels. The implicit rule for process invocation then reads

$$\frac{\Omega' \leq \Omega_f \quad (\Omega_f \vdash^{i} f = P_f :: (x : A)) \in \Sigma \quad \Omega, x{:}A \vdash^{i} Q :: (z : C)}{\Omega, \Omega' \vdash^{i} (x \leftarrow f \leftarrow \Omega' \,;\, Q) :: (z : C)}\ \text{def}$$

But how do we define subtyping $A \leq B$? We would like the coercion to be an identity on basic session types and just deal with temporal mismatches through appropriate delay, when?, and now! actions. In other words, A should be a subtype of B if and only if $y : A \vdash^{i} x \leftarrow y :: (x : B)$. Given this desired theorem, we can just read off the subtyping rules from the implicit typing rules in Figure 6 by using the forwarding process $x \leftarrow y$ as the subject in each rule! This form of subtyping is independent from subtyping between basic session types [Gay and Hole 2005], which we believe can be added to our system in a sound way, even if it would not be complete for asynchronous communication [Lange and Yoshida 2017].

This approach yields the rules in Figure 7, where we have split the $\bigcirc LR$ rule into three different cases. We have expanded the definitions of *patient* types to make it syntactically more self-contained.

Theorem 6.1 (Subtyping Identity). *$A \leq B$ iff $y : A \vdash^{i} x \leftarrow y :: (x : B)$*

Proof. In each direction by induction over the structure of the given deduction. □

The subtyping rules are manifestly decidable. In the bottom-up search for a subtyping derivation, the rules $\bigcirc\bigcirc$, $\Box R$, and $\Diamond L$ can be applied eagerly without losing completeness. There is a nontrivial decision point between the $\Box\bigcirc$ and $\Box L$ rules. The examples $\Box S \leq \bigcirc\Box S$ and $\Box\bigcirc S \leq \bigcirc S$ for a basic session type S show that sometimes $\Box\bigcirc$ must be chosen and sometimes $\Box L$ when both rules apply. A dual non-deterministic choice exists between $\bigcirc\Diamond$ and $\Diamond R$. The cost of backtracking is minimal in all examples we have considered.

We already know that subtype coercions are identities. To verify that we have a sensible subtype relation it remains to prove that transitivity is admissible. For this purpose we need two lemmas regarding patient types, as they appear in the $\Box R$ and $\Diamond L$ rules.

Lemma 6.2 (Patience).

(i) *If $A \leq \bigcirc^n \Box B$ then $A = \bigcirc^k \Box A'$ for some k and A'.*
(ii) *If $\bigcirc^n \Diamond A \leq B$ then $B = \bigcirc^k \Diamond B'$ for some k and B'.*

Proof. By separate inductions over the structure of the given deductions. □

Lemma 6.3 (Impatience).

(i) *If $\bigcirc\bigcirc^n \Box A \leq B$ then $\bigcirc^n \Box A \leq B$.*
(ii) *If $A \leq \bigcirc\bigcirc^n \Diamond B$ then $A \leq \bigcirc^n \Diamond B$.*

Proof. By separate inductions over the structure of the given deductions. □

Theorem 6.4 (Transitivity of Subtyping).
If $A \leq B$ and $B \leq C$ then $A \leq C$.

Proof. By simultaneous induction on the structure of the deductions $\mathcal{D}$ of $A \leq B$ and $\mathcal{E}$ of $B \leq C$ with appeals to the preceding lemmas in four cases. □

7 FURTHER EXAMPLES

In this section we present example analyses of some of the properties that we can express in the type system, such as the message rates of streams, the response time of concurrent data structures, and the span of a fork/join parallel program.

In some examples we use parametric definitions, both at the level of types and processes. For example, stack_A describes stacks parameterized over a type A, $\mathsf{list}_A[n]$ describes lists of n elements, and $\mathsf{tree}[h]$ describes binary trees of height h. Process definitions are similarly parameterized. We think of these as families of ordinary definitions and calculate with them accordingly, at the metalevel, which is justified since they are only implicitly quantified across whole definitions. This common practice (for example, in work on interaction nets Gimenez and Moser [2016]) avoids significant syntactic overhead, highlighting conceptual insight. It is of course possible to internalize such parameters (see, for example, work on refinement of session types [Griffith and Gunter 2013] or explicitly polymorphic session types [Caires et al. 2013; Griffith 2016]).

7.1 Response Times: Stacks and Queues

To analyze response times, we study concurrent stacks and queues. A stack data structure provides a client with a choice between a push and a pop. After a push, the client has to send an element, and the provider will again behave like a stack. After a pop, the provider will reply either with the label none and terminate (if there are no elements in the stack), or send an element and behave again like a stack. In the cost-free model, this is expressed in the following session type.

$$\begin{aligned}\mathsf{stack}_A = \&\{\ & \mathsf{push} : A \multimap \mathsf{stack}_A,\\ & \mathsf{pop} : \oplus\{\ \mathsf{none} : \mathbf{1}, \mathsf{some} : A \otimes \mathsf{stack}_A\ \}\ \}\end{aligned}$$

We implement a stack as a chain of processes. The bottom to the stack is defined by the process *empty*, while a process *elem* holds a top element of the stack as well as a channel with access to the top of the remainder of the stack.

$$\begin{aligned}& x : A, t : \mathsf{stack}_A \vdash \mathit{elem} :: (s : \mathsf{stack}_A)\\ & \cdot \vdash \mathit{empty} :: (s : \mathsf{stack}_A)\end{aligned}$$

The cost model we would like to consider here is $\mathcal{RS}$ where both receives and sends cost one unit of time. Because a receive costs one unit, every continuation type must be delayed by one tick of the clock, which we have denoted by prefixing continuations by the $\bigcirc$ modality. This delay is not an artifact of the implementation, but an inevitable part of the cost model—one reason we have distinguished the synonyms tick (delay of one, due to the cost model) and delay (delay of one, to correctly time the interactions). In this section of examples we will make the same distinction for the next-time modality: we write $\text{‘}A$ for a step in time mandated by the cost model, and $\bigcirc A$ for a delay necessitated by a particular set of process definitions.

As a first approximation, we would have

$$\begin{aligned}\mathsf{stack}_A = \&\{\ & \mathsf{push} : \text{‘}(A \multimap \text{‘}\mathsf{stack}_A),\\ & \mathsf{pop} : \text{‘}\oplus\{\ \mathsf{none} : \text{‘}\mathbf{1}, \mathsf{some} : \text{‘}(A \otimes \text{‘}\mathsf{stack}_A)\ \}\ \}\end{aligned}$$

There are several problems with this type. The stack is a data structure and has little or no control over *when* elements will be pushed onto or popped from the stack. Therefore we should use a type $\Box\mathsf{stack}_A$ to indicate that the client can choose the times of interaction with the stack. While the elements are held by the stack time advances in an indeterminate manner. Therefore, the elements stored in the stack must also have type $\Box A$, not A (so that they are always available).

$$\begin{aligned}\mathsf{stack}_A = \&\{\ & \mathsf{push} : \text{‘}(\Box A \multimap \text{‘}\Box\mathsf{stack}_A),\\ & \mathsf{pop} : \text{‘}\oplus\{\ \mathsf{none} : \text{‘}\mathbf{1}, \mathsf{some} : \text{‘}(\Box A \otimes \text{‘}\Box\mathsf{stack}_A)\ \}\ \}\end{aligned}$$

$$x : \Box A, t : \Box\mathsf{stack}_A \vdash \mathit{elem} :: (s : \Box\mathsf{stack}_A)$$

$$\cdot \vdash \mathit{empty} :: (s : \Box\mathsf{stack}_A)$$

This type expresses that the data structure is very efficient in its response time: there is no additional delay after it receives a push and then an element of type $\Box A$ before it can take the next request, and it will respond immediately to a pop request. It may not be immediately obvious that such an efficient implementation actually exists in the $\mathcal{RS}$ cost model, but it does. We use the implicit form from Section 6 omitting the tick constructs after each receive and send, and also the when? before each case that goes along with type $\Box A$.

```
s ← elem ← x t =
  case s ( push ⇒ y ← recv s ;
                  s′ ← elem ← x t ;          % previous top of stack, holding x
                  s ← elem ← y s′            % new top of stack, holding y
         | pop ⇒  s.some ;
                  send s x ;                 % send channel x along s
                  s ← t )                    % s is now provided by t, via forwarding
s ← empty =
  case s ( push ⇒ y ← recv s ;
                  e ← empty ;                % new bottom of stack
                  s ← elem ← y e
         | pop ⇒  s.none ;
                  close s )
```

The specification and implementation of a queue is very similar. The key difference in the implementation is that when we receive a new element we pass it along the chain of processes until it reaches the end. So instead of

```
s′ ← elem ← x t ;          % previous top of stack, holding x
s ← elem ← y s′            % new top of stack, holding y
```

we write

```
t.enq ;
send t y ;                 % send y to the back of the queue
s ← elem ← x t
```

These two send operations take two units of time, which must be reflected in the type: after a channel of type $\Box A$ has been received, there is a delay of an additional two units of time before the provider can accept the next request.

$$\begin{aligned}\mathsf{queue}_A = \&\{\, & \mathsf{enq} : {}^{\backprime}(\Box A \multimap {}^{\backprime}\bigcirc\bigcirc\Box\mathsf{queue}_A),\\ & \mathsf{deq} : {}^{\backprime}\oplus\{\, \mathsf{none} : {}^{\backprime}\mathbf{1}, \mathsf{some} : {}^{\backprime}(\Box A \otimes {}^{\backprime}\Box\mathsf{queue}_A)\,\}\,\}\end{aligned}$$

$$x : \Box A, t : \bigcirc\bigcirc\Box\mathsf{queue}_A \vdash \mathit{elem} :: (s : \Box\mathsf{queue}_A)$$
$$\cdot \vdash \mathit{empty} :: (s : \Box\mathsf{queue}_A)$$

Time reconstruction will insert the additional delays in the *empty* process through subtyping, using $\Box\mathsf{queue}_A \leq \bigcirc\bigcirc\Box\mathsf{queue}_A$. We have syntactically expanded the tail call so the second use of subtyping is more apparent.

```
s ← empty =
  case s ( enq ⇒ y ← recv s ;            % y : □A ⊢ s : ○○□queue_A
                 e ← empty ;             % y : □A, e : □queue_A ⊢ s : ○○□queue_A
                 s′ ← elem ← y e ;       % □queue_A ≤ ○○□queue_A (on e)
                 s ← s′                  % □queue_A ≤ ○○□queue_A (on s′)
         | deq ⇒ s.none ;
```

```
                    close s )
```

The difference between the *response times* of stacks and queues in the cost model is minimal: both are constant, with the queue being two units slower. This is in contrast to the total work [Das et al. 2017] which is constant for the stack but linear in the number of elements for the queue.

This difference in response times can be realized by typing clients of both stacks and queues. We compare clients S_n and Q_n that insert n elements into a stack and queue, respectively, send the result along channel d, and then terminate. We show only their type below, omitting the implementations.

$$x_1 : \Box A, \ldots, x_n : \Box A, s : \Box\mathsf{stack}_A \vdash S_n :: (d : \bigcirc^{2n} (\Box\mathsf{stack}_A \otimes {‘}\mathbf{1}))$$
$$x_1 : \Box A, \ldots, x_n : \Box A, s : \Box\mathsf{queue}_A \vdash Q_n :: (d : \bigcirc^{4n} (\Box\mathsf{queue}_A \otimes {‘}\mathbf{1}))$$

The types demonstrate that the total execution time of S_n is only $2n + 1$, while it is $4n + 1$ for Q_n. The difference comes from the difference in response times. Note that we can infer precise execution times, even in the presence of the $\Box$ modality in the stack and queue types.

7.2 Parametric Rates: Lists and Streams

Lists describe an interface that sends either nil and ends the session, or sends cons followed by a channel of some type A and then behaves again like a list. In the cost-free setting:

$$\mathsf{list}_A = \oplus\{\, \mathsf{cons} : A \otimes \mathsf{list}_A, \mathsf{nil} : \mathbf{1} \,\}$$

Here is the straightforward definition of *append*.

$$l_1 : \mathsf{list}_A, l_2 : \mathsf{list}_A \vdash \mathit{append} : (l : \mathsf{list}_A)$$

```
$l \leftarrow append \leftarrow l_1\ l_2$ =
  case $l_1$  ( cons ⇒ $x \leftarrow$ recv $l_1$ ;           % receive element $x$ from $l_1$
                       $l$.cons ; send $l\ x$ ;           % send $x$ along $l$
                       $l \leftarrow append \leftarrow l_1\ l_2$      % recurse
              | nil ⇒  wait $l_1$ ;                      % wait for $l_1$ to close
                       $l \leftarrow l_2$ )                   % identify $l$ and $l_2$
```

In this example we are interested in analyzing the timing of several processes precisely, but parametrically over an arrival rate. Because it takes two units of time to copy the inputs to the outputs, the arrival rate needs to be at least 2, which we represent by writing it as $r + 2$. Since we append the two lists, the second list will be idle while we copy the elements from the first list to the output. We could give this list type $\Box(-)$, but we can also precisely determine the delay if we index lists by the number of elements. We write $\mathsf{list}_A[n]$ for a list sending exactly n elements. We have the following types in the $\mathcal{RS}$ cost model:

$$\mathsf{list}_A[0] = \oplus\{\, \mathsf{nil} : {‘}\mathbf{1} \,\}$$
$$\mathsf{list}_A[n+1] = \oplus\{\, \mathsf{cons} : {‘}(\Box A \otimes {‘}\bigcirc^{r+2}\, \mathsf{list}_A[n]) \,\}$$

As before, the tick marks account for the delay mandated by the cost model. The $\bigcirc^{r+2}$ accounts for the arrival rate of $r + 2$. We use type $\Box A$ for the elements since they will be in the lists for an indeterminate amount of time. The precise type of *append* then becomes

$$l_1 : \mathsf{list}_A[n], l_2 : \bigcirc^{(r+4)n+2}\, \mathsf{list}_A[k] \vdash \mathit{append} :: (l : \bigcirc\bigcirc \mathsf{list}_A[n+k])$$

It expresses that the output list has the same rate as the input lists, but with a delay of 2 cycles relative to l_1. The channel l_2 has to sit idle for $r + 4$ cycles for each element of l_1, accounting for the two inputs along l_1 and two outputs along l_2. It takes 2 further cycles to input the nil and the end token for the list.

With our type system and just a little bit of arithmetic we can verify this type, checking the definition twice: once for a list of length 0 and once for $n + 1$. We show here the latter, where $l_1 : \mathsf{list}_A[n+1]$.

```
$l \leftarrow \mathit{append} \leftarrow l_1\ l_2 =$
case $l_1$ ( cons $\Rightarrow$      % $l_1{:}\Box A \otimes \text{‘}\bigcirc^{r+2}\,\text{list}_A[n], l_2 : [\bigcirc^{(r+4)(n+1)+2}\,\text{list}_A[k]]_L^{-1} \vdash l : \bigcirc\text{list}_A[(n+1)+k]$
         $x \leftarrow$ recv $l_1$ ;  % $x{:}\Box A, l_1{:}\bigcirc^{r+2}\,\text{list}_A[n], l_2 : [\bigcirc^{(r+4)(n+1)+2}\,\text{list}_A[k]]_L^{-2} \vdash l : \text{list}_A[(n+1)+k]$
         $l$.cons ;        % $x{:}\Box A, l_1{:}\bigcirc^{r+1}\,\text{list}_A[n], l_2 : [\bigcirc^{(r+4)(n+1)+2}\,\text{list}_A[k]]_L^{-3} \vdash l : \Box A \otimes \text{‘}\bigcirc^{r+2}\,\text{list}_A[n+k]$
         send $l\ x$ ;       % $l_1{:}\bigcirc^{r}\,\text{list}_A[n], l_2 : [\bigcirc^{(r+4)(n+1)+2}\,\text{list}_A[k]]_L^{-4} \vdash l : \bigcirc^{r+2}\,\text{list}_A[n+k]$
         % delay$^r$         % $l_1{:}\text{list}_A[n], l_2 : [\bigcirc^{(r+4)(n+1)+2}\,\text{list}_A[k]]^{-4-r} \vdash l : \bigcirc^{2}\,\text{list}_A[n+k]$
                           % $l_1{:}\text{list}_A[n], l_2 : \bigcirc^{(r+4)n+2}\,\text{list}_A[k] \vdash l : \bigcirc\bigcirc\text{list}_A[n+k]$
         $l \leftarrow \mathit{append} \leftarrow l_1\ l_2$
         | nil $\Rightarrow \ldots$ )
```

We showed only the one delay by r units inserted by time reconstruction since it is the critical step. The case for nil does not apply for $l_1 : \text{list}_A[n+1]$. Here is the typing derivation when $l_1 : \text{list}_A[0]$ where the cons branch does not apply.

```
$l \leftarrow \mathit{append} \leftarrow l_1\ l_2 =$
  case $l_1$ ( cons $\Rightarrow \ldots$
           | nil $\Rightarrow$              % $l_1 : \mathbf{1}, l_2 : \bigcirc\text{list}_A[k] \vdash l : \bigcirc\text{list}_A[k]$
                  wait $l_1$ ;       % $l_2 : \text{list}_A[k] \vdash l : \text{list}_A[k]$
                  $l \leftarrow l_2$ )
```

As a related example we consider a process that alternates the elements between two infinite input streams. At first we might expect if the two input streams come in with a rate of 2 then the output stream will have a rate of 1. However, in the $\mathcal{RS}$ cost model one additional tick is required for sending on the messages which means that the input streams need to have rate 3 and be offset by 2 cycles. We parameterize the type of stream by its rate k

$$\text{stream}_A^k = \Box A \otimes \text{‘}\bigcirc^k\,\text{stream}_A^k$$

$$l_1 : \text{stream}_A^3, l_2 : \bigcirc^2\,\text{stream}_A^3 \vdash \mathit{alternate} :: (l : \bigcirc^1\,\text{stream}_A^1)$$

```
$l \leftarrow \mathit{alternate} \leftarrow l_1\ l_2 =$
  $x \leftarrow$ recv $l_1$ ;          % $x : \Box A,\ l_1 : \bigcirc^3\,\text{stream}_A^3, l_2 : \bigcirc^1\,\text{stream}_A^3 \vdash l : \text{stream}_A^1$
  send $l\ x$ ;               %          $l_1 : \bigcirc^2\,\text{stream}_A^3, l_2 : \text{stream}_A^3 \vdash l : \bigcirc^1\,\text{stream}_A^1$
  $l \leftarrow \mathit{alternate} \leftarrow l_2\ l_1$ )
```

A more general parametric type for the same code would be

$$l_1 : \text{stream}_A^{2k+3}, l_2 : \bigcirc^{k+2}\,\text{stream}_A^{2k+3} \vdash \mathit{alternate} :: (l : \bigcirc^1\,\text{stream}_A^{k+1})$$

from which we can recover the more specialized one with $k = 0$.

7.3 Span Analysis: Trees

We use trees to illustrate an example that is typical for fork/join parallelism and computation of *span*. In order to avoid integers, we just compute the parity of a binary tree of height h with boolean values at the leaves. We do not show the obvious definition of *xor*, which in the $\mathcal{RS}$ cost model requires a delay of four from the first input.

$$\text{bool} = \oplus\{\,\text{b0} : \text{‘}\mathbf{1}, \text{b1} : \text{‘}\mathbf{1}\,\}$$

$$a : \text{bool}, b : \bigcirc^2\,\text{bool} \vdash \mathit{xor} :: (c : \bigcirc^4\,\text{bool})$$

In the definition of *leaf* and *node* we have explicated the delays inferred by time reconstruction, but not the tick delays. The type of tree$[h]$ gives the *span* of this particular parallel computation as $5h + 2$. This is the time it takes to compute the parity under maximal parallelism, assuming that *xor* takes 4 cycles as shown in the type above.

$$\text{tree}[h] = \&\{\,\text{parity} : \text{‘}\bigcirc^{5h+2}\,\text{bool}\,\}$$

```
$\cdot \vdash \mathit{leaf} :: (t : \mathsf{tree}[h])$
$t \leftarrow \mathit{leaf} =$
   case $t$ ( parity $\Rightarrow$                   % $\cdot \vdash t : \bigcirc^{5h+2}\,\mathsf{bool}$
                     % delay$^{5h+2}$             % $\cdot \vdash t : \mathsf{bool}$
                     $t$.b0 ;                     % $\cdot \vdash t : \mathbf{1}$
                     close $t$ )
```

```
$l : \bigcirc^1\mathsf{tree}[h], r : \bigcirc^3\,\mathsf{tree}[h] \vdash \mathit{node} :: (t : \mathit{tree}[h+1])$
$t \leftarrow \mathit{node} \leftarrow l\ r =$
   case $t$ ( parity $\Rightarrow$                   % $l : \mathsf{tree}[h], r : \bigcirc^2\,\mathsf{tree}[h] \vdash t : \bigcirc^{5(h+1)+2}\,\mathsf{bool}$
                     $l$.parity ;                 % $l : \bigcirc^{5h+2}\,\mathsf{bool}, r : \bigcirc^1\mathsf{tree}[h] \vdash t : \bigcirc^{5(h+1)+1}\,\mathsf{bool}$
                     % delay                      % $l : \bigcirc^{5h+1}\,\mathsf{bool}, r : \mathsf{tree}[h] \vdash t : \bigcirc^{5h+5}\,\mathsf{bool}$
                     $r$.parity ;                 % $l : \bigcirc^{5h}\,\mathsf{bool}, r : \bigcirc^{5h+2}\,\mathsf{bool} \vdash t : \bigcirc^{5h+4}\,\mathsf{bool}$
                     % delay$^{5h}$               % $l : \mathsf{bool}, r : \bigcirc^2\,\mathsf{bool} \vdash t : \bigcirc^4\,\mathsf{bool}$
                     $t \leftarrow \mathit{xor} \leftarrow l\ r$ )
```

The type $l : \bigcirc^1\,\mathsf{tree}[h]$ comes from the fact that, after receiving a parity request, we first send out the parity request to the left subtree l. The type $r : \bigcirc^3\,\mathsf{tree}[h]$ is determined from the delay of 2 between the two inputs to *xor*. The magic number 5 in the type of tree was derived in reverse from setting up the goal of type-checking the *node* process under the constraints already mentioned. We can also think of it as 4+1, where 4 is the time to compute the exclusive or at each level and 1 as the time to propagate the parity request down each level.

As is often done in abstract complexity analysis, we can also impose an alternative cost model. For example, we may count only the number of calls to *xor* while all other operations are cost free. Then we would have

$$a : \mathsf{bool}, b : \mathsf{bool} \vdash \mathit{xor} :: (c : \bigcirc\mathsf{bool}) \qquad \cdot \vdash \mathsf{leaf} :: (t : \mathsf{tree}[h])$$

$$\mathsf{tree}[h] = \&\{\,\mathsf{parity} : \bigcirc^h\,\mathsf{bool}\,\} \qquad l : \mathsf{tree}[h], r : \mathsf{tree}[h] \vdash \mathsf{node} :: (t : \mathsf{tree}[h+1])$$

with the same code but different times and delays from before. The reader is invited to reconstruct the details.

7.4 A Higher-Order Example

As an example of higher-order programming we show how to encode a process analogue of a fold function. Because our language is purely linear the process to fold over a list has to be recursively defined. In the cost-free setting we would write

```
$\mathsf{folder}_{AB} = \&\{\,\mathsf{next} : A \multimap (B \multimap (B \otimes \mathsf{folder}_{AB})), \mathsf{done} : \mathbf{1}\,\}$
$l : \mathsf{list}_A, f : \mathsf{folder}_{AB}, b : B \vdash \mathit{fold} :: (r : B)$
$r \leftarrow \mathit{fold} \leftarrow l\ f\ b =$
   case $l$  ( cons $\Rightarrow$ $x \leftarrow$ recv $l$ ;
                     $f$.next ; send $f$ $x$ ; send $f$ $b$ ;        % send $x$ and $b$ to folder $f$
                     $y \leftarrow$ recv $f$ ; $r \leftarrow \mathit{fold} \leftarrow l\ f\ y$       % receive $y$ from $f$ and recurse
             | nil $\Rightarrow$   wait $l$ ; $f$.done ; wait $f$ ; $r \leftarrow b$ )
```

If we want to assign precise temporal types to the *fold* process then the incoming list should have a delay of at least 4 between successive elements. Working backwards from the code we obtain the following types.

$$\mathsf{list}_A[0] = \oplus\{\,\mathsf{nil} : `\mathbf{1}\,\}$$
$$\mathsf{list}_A[n+1] = \oplus\{\,\mathsf{cons} : `(\Box A \otimes `\bigcirc^{k+4}\,\mathsf{list}_A[n])\,\}$$

$$\begin{aligned}&\mathsf{folder}_{AB} = \&\{\,\mathsf{next} : `(\Box A \multimap `(B \multimap `\bigcirc^k(\bigcirc^5 B \otimes `\bigcirc^2\,\mathsf{folder}_{AB}))), \mathsf{done} : `\mathbf{1}\,\}\\ &l : \mathsf{list}_A[n], f : \bigcirc^2\,\mathsf{folder}_{AB}, b : \bigcirc^4 B \vdash \mathit{fold} :: (r : \bigcirc^{(k+5)n+4}\,B)\end{aligned}$$

The type of *fold* indicates that if the combine function of folder_{AB} takes k time units to compute, the result r is produced after $(k+5)n+4$ time units in the $\mathcal{RS}$ cost model.

8 RELATION TO THE STANDARD SEMANTICS

While our temporal semantics stands on its own, one may ask precisely in which way it captures properties of the standard semantics. We analyze this here for the fragment with only the next-time modality $\bigcirc A$; the general case including $\Box A$ and $\Diamond A$ is just slightly more complicated. We proceed in several steps.

Step 1: Standard Semantics. The standard operational semantics for basic session types (without temporal modalities) is precisely that in Figure 4 where the time t always remains at 0.

Step 2: Measuring Span. We are interested in capturing the *span* of a standard computation, which is the number of steps required to completion under the assumption that any action takes place as soon as possible, subject only to its dependencies. We can measure this by *instrumenting* the standard semantics, as was done by Silva et al. [2016], except that we increment time on every step instead of just when messages are received. We write $\mathsf{proc}^*(c,t,P)$ and $\mathsf{msg}^*(c,t,M)$ to distinguish them because the time t has a different interpretation, namely the earliest time the process could have arrived at this point in the computation under an asynchronous model of communication. We show the two rules for internal choice to illustrate this semantics.

$$\begin{array}{l}\mathsf{proc}^*(c,t,c.k\ ;P) \;\mapsto\; \mathsf{proc}^*(c',t+1,[c'/c]P), \mathsf{msg}^*(c,t+1,c.k\ ;c \leftarrow c') \quad (c'\ \text{fresh})\\ \mathsf{msg}^*(c,t,c.k\ ;c \leftarrow c'), \mathsf{proc}^*(d,t',\mathsf{case}\ c\ (\ell \Rightarrow Q_\ell)_{\ell\in L}) \;\mapsto\; \mathsf{proc}^*(d,\max(t,t'+1),[c'/c]Q_k)\end{array}$$

Step 3: Relating Computations. We define $|A|$ and $|P|$ as the result of erasing all next-time modalities from A and delay actions from $|P|$, respectively. We would like to relate computations of $\Omega \vdash P : A$ to those of $|\Omega| \vdash |P| :: (c : |A|)$. However, we have to ensure that there are enough delay operators in P to account for the fact that the cost model for standard computations counts every step. To this end, we define two mutually recursive relations $P \geq Q$ and $P > Q$ where $Q = |P|$ but there are further constraints on P. $P \geq Q$ expresses that P and Q start with the same action (which cannot be a delay) and the continuations P' and Q' are related with $P' > Q'$. This in turn requires one or more initial delays in P' with its continuation $P'' \geq Q'$. These relations embody the idea that a delay of the cost model *precedes* each action, which is necessary since close and forwarding actions have no continuation. We only show the rules for sending and receiving labels.

$$\frac{P > Q}{c.k\ ;P \geq c.k\ ;Q} \qquad \frac{(\forall \ell \in L)\quad P_\ell > Q_\ell}{\mathsf{case}\ c\ (\ell \Rightarrow P_\ell)_{\ell\in L} \geq \mathsf{case}\ c\ (\ell \Rightarrow Q_\ell)_{\ell\in L}} \qquad \frac{P > Q}{\mathsf{delay}\ ;P > Q} \qquad \frac{P \geq Q}{\mathsf{delay}\ ;P > Q}$$

We then extend this relation to all semantic objects, expressing that the temporal semantics may be an over-approximation of the standard semantics because it may contain arbitrarily many additional delay actions. This is expressed formally in the conditions on the time stamps of corresponding processes and messages.

$$\begin{array}{ll}\mathsf{proc}(c,s,P) \geq \mathsf{proc}^*(c,t,Q) & \text{if} \quad (P > Q \text{ and } s \geq t) \text{ or } (P \geq Q \text{ and } s > t)\\ \mathsf{msg}(c,s,M) \geq \mathsf{msg}^*(c,t,M) & \text{if } s \geq t\end{array}$$

We then extend this compositionally to full configurations, $\mathcal{C} \geq \mathcal{D}$.

Bisimulation. The $\geq$ relation is a weak bisimulation in the following sense, where $\mapsto^{\leq 1}$ means at most one transition and $\mapsto^{\geq 1}$ means at least one transition. Note that the approximation property comes from the relation between time stamps in the two configurations, not the number of transitions.

Theorem 8.1. *Assume* $\cdot \vDash \mathcal{C} :: \Omega$ *and* $\cdot \vDash \mathcal{D} :: |\Omega|$.

(i) *If* $\mathcal{C} \geq \mathcal{D}$ *and* $\mathcal{C} \mapsto \mathcal{C}'$ *then* $\mathcal{D} \mapsto^{\leq 1} \mathcal{D}'$ *for some* $\mathcal{D}'$ *with* $\mathcal{C}' \geq \mathcal{D}'$.
(ii) *If* $\mathcal{C} \geq \mathcal{D}$ *and* $\mathcal{D} \mapsto \mathcal{D}'$ *then* $\mathcal{C} \mapsto^{\geq 1} \mathcal{C}'$ *for some* $\mathcal{C}'$ *with* $\mathcal{C}' \geq \mathcal{D}'$.

Proof. In each direction, by analyzing each case of the given reduction, with a case analysis or induction over the definition of $\geq$. □

The treatment of $\Box A$ and $\Diamond A$ is slightly more complicated. We obtain the simplest generalization by defining $|\Box A| = \&\{\mathsf{now} : |A|\}$ and $|\Diamond A| = \oplus\{\mathsf{now} : |A|\}$ with the corresponding erasure in the process expressions. If we want to avoid such "administrative messages" we can instead fully erase all temporal constructs but enforce a normal form on process expressions where every when? x action is always immediately followed by another receive action along x.

9 FURTHER RELATED WORK

In addition to the related work already mentioned, we highlight a few related threads of research.

Session types and process calculi. In addition to the work on timed multiparty session types [Bocchi et al. 2014; Neykova et al. 2014], time has been introduced into the π-calculus (see, for example, Saeedloei and Gupta [2014]) or session-based communication primitives (see, for example, López et al. [2009]) but generally these works do not develop a type system. Kobayashi [2002] extends a (synchronous) π-calculus with means to count parallel reduction steps. He then provides a type system to verify time-boundedness. This is more general in some dimension than our work because of a more permissive underlying type and usage system, but it lacks internal and external choice, genericity in the cost model, and provides bounds rather than a fine gradation between exact and indefinite times. Session types can also be derived by a Curry-Howard interpretation of *classical linear logic* [Wadler 2012] but we are not aware of temporal extensions. We conjecture that there is a classical version of our system where $\Box$ and $\Diamond$ are dual and $\bigcirc$ is self-dual.

Reactive programming. Synchronous data flow languages such as Lustre [Halbwachs et al. 1991], Esterel [Berry and Gonthier 1992], or Lucid Synchrone [Pouzet 2006] are time-synchronous with uni-directional flow and thus may be compared to the fragment of our language with internal choice ($\oplus$) and the next-time modality ($\bigcirc A$), augmented with existential quantification over basic data values like booleans and integers (which we have omitted here only for the sake of brevity). The global clock would map to our underlying notion of time, but data-dependent local clocks would have to be encoded at a relatively low level using streams of option type, compromising the brevity and elegance of these languages. Furthermore, synchronous data flow languages generally permit sharing of channels, which, although part of many session-typed languages [Balzer and Pfenning 2017; Caires and Pfenning 2010], require further investigation in our setting. On the other hand, we support a number of additional types such as external choice ($\&$) for bidirectional communication and higher-order channel-passing ($A \multimap B$, $A \otimes B$). In the context of functional reactive programming, a Nakano-style [Nakano 2000] temporal modality has been used to ensure productivity [Krishnaswami and Benton 2011]. A difference in our work is that we consider concurrent processes and that our types prescribe the timing of messages.

Computational interpretations of $\bigcirc A$. A first computational interpretation of the next-time modality under a proofs-as-programs paradigm was given by Davies [1996]. The basis is natural deduction for a (non-linear!) intutionistic linear-time temporal logic with only the next-time modality. Rather than capturing cost, the programmer could indicate *staging* by stipulating that some subexpressions should be evaluated "at the next time". The natural operational semantics then is a logically-motivated form of *partial evaluation* which yields a residual program of type $\bigcirc A$. This idea was picked up by Feltman et al. [2016] to instead *split* the program statically into two stages where results from the first stage are communicated to the second. Again, neither linearity (in the sense of linear logic), nor any specific cost semantics appears in this work.

Other techniques. Inferring the cost of concurrent programs is a fundamental problem in resource analysis. Hoffmann and Shao [2015] introduce the first automatic analysis for deriving bounds on the worst-case evaluation cost of parallel first-order functional programs. Their main limitation is that they can only handle parallel computation; they don't support message-passing or shared memory based concurrency. Blelloch and Reid-Miller [1997] use pipelining [Paul et al. 1983] to improve the complexity of parallel algorithms. However, they use futures [Halstead 1985], a parallel language construct to implement pipelining without the programmer having to specify them explicitly. The runtime of algorithms is determined by analyzing the work and depth in a language-based cost model. The work relates to ours in the sense that pipelines can have delays, which can be data dependent. However, the algorithms they analyze have no message-passing concurrency or other synchronization constructs. Albert et al. [2015] devised a static analysis for inferring the parallel cost of distributed systems. They first perform a block-level analysis to estimate the serial cost, then construct a distributed flow graph (DFG) to capture the parallelism and then obtain the parallel cost by computing the maximal cost path in the DFG. However, the bounds they produce are modulo a points-to and serial cost analysis. Hence, an imprecise points-to analysis will result in imprecise parallel cost bounds. Moreover, since their technique is based on static analysis, it is not compositional and a whole program analysis is needed to infer bounds on each module. Recently, a bounded linear typing discipline [Ghica and Smith 2014] modeled in a semiring was proposed for resource-sensitive compilation. It was then used to calculate and control execution time in a higher-order functional programming language. However, this language did not support recursion.

10 CONCLUSION

We have developed a system of temporal session types that can accommodate and analyze concurrent programs with respect to a variety of different cost models. Types can vary in precision, based on desired and available information, and includes latency, rate, response time, and span of computations. It is constructed in a modular way, on top of a system of basic session types, and therefore lends itself to easy generalization. We have illustrated the type system through a number of simple programs on streams of bits, binary counters, lists, stacks, queues, and trees. Time reconstruction and subtyping go some way towards alleviating demands on the programmer and supporting program reuse. In ongoing work we are exploring an implementation with an eye toward practical aspects of time reconstruction and, beyond that, automatic resource analysis based on internal measures of processes such as the length of a list or the height of a tree—so far, we have carried out these analyses by hand.

ACKNOWLEDGMENTS

This article is based on research that has been supported, in part, by AFRL under DARPA STAC award FA8750-15-C-0082 and in part by the NSF Foundation under Grants No. 1718267 and 1812876. Any opinions, findings, and conclusions contained in this document are those of the authors and do not necessarily reflect the views of the sponsoring organizations.

REFERENCES

Elvira Albert, Jesús Correas, Einar Broch Johnsen, and Guillermo Román-Díez. 2015. Parallel Cost Analysis of Distributed Systems. In *Static Analysis*, Sandrine Blazy and Thomas Jensen (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 275–292.

Martin Avanzini, Ugo Dal Lago, and Georg Moser. 2015. Analysing the Complexity of Functional Programs: Higher-Order Meets First-Order. In *29th Int. Conf. on Functional Programming (ICFP'15)*.

Stephanie Balzer and Frank Pfenning. 2017. Manifest Sharing with Session Types. In *International Conference on Functional Programming (ICFP)*. ACM, 37:1–37:29.

Gérard Berry and Georges Gonthier. 1992. The ESTEREL Synchronous Programming Language: Design, Semantics, Implementation. *Sci. Comput. Program.* 19, 2 (Nov. 1992), 87–152. https://doi.org/10.1016/0167-6423(92)90005-V

Guy E. Blelloch and Margaret Reid-Miller. 1997. Pipelining with Futures. In *Proceedings of the Ninth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA '97)*. ACM, New York, NY, USA, 249–259. https://doi.org/10.1145/258492.258517

Laura Bocchi, Weizhen Yang, and Nobuko Yoshida. 2014. Timed Multiparty Session Types. In *CONCUR 2014 – Concurrency Theory*, Paolo Baldan and Daniele Gorla (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 419–434.

Luís Caires, Jorge A. Pérez, Frank Pfenning, and Bernardo Toninho. 2013. Behavioral Polymorphism and Parametricity in Session-Based Communication. In *Proceedings of the European Symposium on Programming (ESOP'13)*, M.Felleisen and P.Gardner (Eds.). Springer LNCS 7792, Rome, Italy, 330–349.

Luís Caires and Frank Pfenning. 2010. Session Types as Intuitionistic Linear Propositions. In *Proceedings of the 21st International Conference on Concurrency Theory (CONCUR 2010)*. Springer LNCS 6269, Paris, France, 222–236.

Luís Caires, Frank Pfenning, and Bernardo Toninho. 2016. Linear Logic Propositions as Session Types. *Mathematical Structures in Computer Science* 26, 3 (2016), 367–423.

Iliano Cervesato and Andre Scedrov. 2009. Relating State-Based and Process-Based Concurrency through Linear Logic. *Information and Computation* 207, 10 (Oct. 2009), 1044–1077.

Norman Danner, Daniel R. Licata, and Ramyaa Ramyaa. 2015. Denotational Cost Semantics for Functional Languages with Inductive Types. In *29th Int. Conf. on Functional Programming (ICFP'15)*.

Ankush Das, Jan Hoffmann, and Frank Pfenning. 2017. Work Analysis with Resource-Aware Session Types. *CoRR* abs/1712.08310 (2017). arXiv:1712.08310 http://arxiv.org/abs/1712.08310

Rowan Davies. 1996. A Temporal Logic Approach to Binding-Time Analysis. In *Proceedings of the Eleventh Annual Symposium on Logic in Computer Science*, E. Clarke (Ed.). IEEE Computer Society Press, New Brunswick, New Jersey, 184–195. http://www.cs.cmu.edu/afs/cs/user/rowan/www/papers/multbta.ps.Z

Nicolas Feltman, Carlo Angiuli, Umut Acar, and Kayvon Fatahalian. 2016. Automatically Splitting a Two-Stage Lambda Calculus. In *Proceedings of the 25th European Symposium on Programming (ESOP)*, P. Thiemann (Ed.). Springer LNCS 9632, Eindhoven, The Netherlands, 255–281.

Jérôme Fortier and Luigi Santocanale. 2013. Cuts for Circular Proofs: Semantics and Cut Elimination. In *22nd Conference on Computer Science Logic (LIPIcs)*, Vol. 23. 248–262.

Simon J. Gay and Malcolm Hole. 2005. Subtyping for Session Types in the π-Calculus. *Acta Informatica* 42, 2–3 (2005), 191–225.

Dan R. Ghica and Alex I. Smith. 2014. Bounded Linear Types in a Resource Semiring. In *Proceedings of the 23rd European Symposium on Programming Languages and Systems - Volume 8410*. Springer-Verlag New York, Inc., New York, NY, USA, 331–350. https://doi.org/10.1007/978-3-642-54833-8_18

Stéphane Gimenez and Georg Moser. 2016. The Complexity of Interaction. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '16)*. ACM, New York, NY, USA, 243–255. https://doi.org/10.1145/2837614.2837646

Dennis Griffith. 2016. *Polarized Substructural Session Types.* Ph.D. Dissertation. University of Illinois at Urbana-Champaign.

Dennis Griffith and Elsa L. Gunter. 2013. Liquid Pi: Inferrable Dependent Session Types. In *Proceedings of the NASA Formal Methods Symposium*. Springer LNCS 7871, 186–197.

Sumit Gulwani, Krishna K. Mehra, and Trishul M. Chilimbi. 2009. SPEED: Precise and Efficient Static Estimation of Program Computational Complexity. In *36th ACM Symp. on Principles of Prog. Langs. (POPL'09)*. 127–139.

N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. 1991. The synchronous data flow programming language LUSTRE. *Proc. IEEE* 79, 9 (Sep 1991), 1305–1320. https://doi.org/10.1109/5.97300

Robert H. Halstead, Jr. 1985. MULTILISP: A Language for Concurrent Symbolic Computation. *ACM Trans. Program. Lang. Syst.* 7, 4 (Oct. 1985), 501–538. https://doi.org/10.1145/4472.4478

Jan Hoffmann, Ankush Das, and Shu-Chun Weng. 2017. Towards Automatic Resource Bound Analysis for OCaml. In *44th Symposium on Principles of Programming Languages (POPL'17)*.

Jan Hoffmann and Zhong Shao. 2015. Automatic Static Cost Analysis for Parallel Programs. In *Proceedings of the 24th European Symposium on Programming on Programming Languages and Systems - Volume 9032*. Springer-Verlag New York, Inc., New York, NY, USA, 132–157. https://doi.org/10.1007/978-3-662-46669-8_6

Kohei Honda, Vasco T. Vasconcelos, and Makoto Kubo. 1998. Language Primitives and Type Discipline for Structured Communication-Based Programming. In *7th European Symposium on Programming Languages and Systems (ESOP'98)*. Springer LNCS 1381, 122–138.

Naoki Kobayashi. 2002. A Type System for Lock-Free Processes. *Information and Computation* 177 (2002), 122–159.

Neelakantan R. Krishnaswami and Nick Benton. 2011. Ultrametric Semantics of Reactive Programs. In *26th IEEE Symposium on Logic in Computer Science, (LICS'11).* 257–266.

Ugo Dal Lago and Marco Gaboardi. 2011. Linear Dependent Types and Relative Completeness. In *26th IEEE Symp. on Logic in Computer Science (LICS'11).* 133–142.

Julien Lange and Nobuko Yoshida. 2017. On the Undecidability of Asynchronous Session Subtyping. In *Proceedings of the 20th International Conference on Foundations of Software Science and Computation Structures (FoSSaCS).* Springer LNCS 10203, 441–457.

Hugo A. López, Carlos Olarte, and Jorge A. Pérez. 2009. Towards a Unified Framework for Declarative Structure Communications. In *Proceedings of the Workshop on Programming Language Approaches to Concurrency and Communication-Centric Software (PLACES)*, A. Beresford and S. Gay (Eds.). EPTCS 17, 1–15.

Hiroshi Nakano. 2000. A Modality for Recursion. In *15th IEEE Symposium on Logic in Computer Science (LICS'00).* 255–266.

Rumyana Neykova, Laura Bocchi, and Nobuko Yoshida. 2014. Timed Runtime Monitoring for Multiparty Conversations. In *3rd International Workshop on Behavioural Types (BEAT 2014).*

W. Paul, U. Vishkin, and H. Wagener. 1983. Parallel dictionaries on 2–3 trees. In *Automata, Languages and Programming*, Josep Diaz (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 597–609.

Frank Pfenning and Dennis Griffith. 2015. Polarized Substructural Session Types. In *Proceedings of the 18th International Conference on Foundations of Software Science and Computation Structures (FoSSaCS 2015)*, A. Pitts (Ed.). Springer LNCS 9034, London, England, 3–22. Invited talk.

Amir Pnueli. 1977. The Temporal Logic of Programs. In *Proceedings of the 18th Symposium on Foundations of Computer Science (FOCS'77).* IEEE Computer Society, 46–57.

Marc Pouzet. 2006. Lucid Synchrone Release, version 3.0 Tutorial and Reference Manual. (2006).

Neda Saeedloei and Gopal Gupta. 2014. Timed π-Calculus. In *8th International Symposium on Trustworthy Global Computing - Volume 8358 (TGC 2013).* Springer-Verlag New York, Inc., New York, NY, USA, 119–135. https://doi.org/10.1007/978-3-319-05119-2_8

Miguel Silva, Mário Florido, and Frank Pfenning. 2016. Non-Blocking Concurrent Imperative Programming with Session Types. In *Fourth International Workshop on Linearity.*

Bernardo Toninho, Luís Caires, and Frank Pfenning. 2013. Higher-Order Processes, Functions, and Sessions: A Monadic Integration. In *Proceedings of the European Symposium on Programming (ESOP'13)*, M.Felleisen and P.Gardner (Eds.). Springer LNCS 7792, Rome, Italy, 350–369.

Bernardo Toninho, Luís Caires, and Frank Pfenning. 2014. Corecursion and Non-Divergence in Session-Typed Processes. In *Proceedings of the 9th International Symposium on Trustworthy Global Computing (TGC 2014)*, M. Maffei and E. Tuosto (Eds.). Springer LNCS 8902, Rome, Italy, 159–175.

Philip Wadler. 2012. Propositions as Sessions. In *Proceedings of the 17th International Conference on Functional Programming (ICFP 2012).* ACM Press, Copenhagen, Denmark, 273–286.

Ezgi Çiçek, Gilles Barthe, Marco Gaboardi, Deepak Garg, and Jan Hoffmann. 2017. Relational Cost Analysis. In *44th Symposium on Principles of Programming Languages (POPL'17).*