Reversible Session-Based Concurrency in Haskell (Draft of a Full Research Paper)*

Folkert de Vries† and Jorge A. Pérez

University of Groningen, The Netherlands

Abstract. Under a reversible semantics, computation steps can be undone. For message-passing, concurrent programs, reversing computation steps is a challenging and delicate task; one typically aims at a formal semantics which is *causally-consistent*. Prior work has addressed this challenge in the context of a process model of multiparty protocols (choreographies) following a so-called *monitors-as-memories* approach. In this paper, we describe our ongoing work on implementing this specific reversible operational semantics in Haskell.

Keywords: Reversible computation · Message-passing concurrency · Haskell.

1 Introduction

This paper describes ongoing work aimed at developing a Haskell implementation of *reversible, message-passing concurrency*. Our implementation is framed within a rather prolific line of research, which aims at establishing rigorous semantic foundations for reversible computing in the challenging concurrent setting (see, e.g., the survey [4]). Our key interest is the interplay of reversibility and message-passing concurrency, which is typically governed by *protocols* among possibly distributed partners.

In a programming language with a reversible semantics, computation steps can be undone. Reversing a sequential program is not hard: we should record enough information about standard forward steps in case we wish to return to a prior state. Reversing a concurrent program is more difficult: since in a concurrent program control resides in more than one point, we require carefully designed *memories* that not only record information about prior actions, but also about the *causal dependencies* between actions originated in different threads. For this reason, in a reversible, concurrent setting it is highly desirable to have a semantics which is *causally consistent*. Causal consistency ensures that reversible steps lead

* This work was partially supported by the EU COST action IC1405 (Reversible Computation – Extending Horizons of Computing).

† Main author - BSc student.

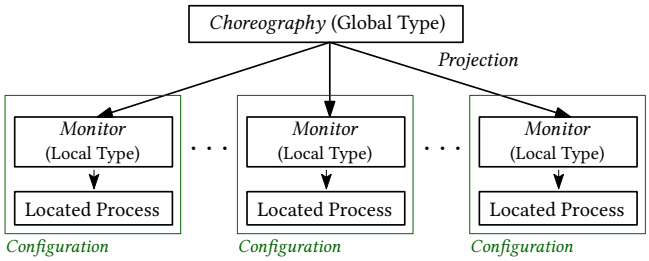

Fig. 1. The process model of multiparty communications defined in [5].

to system states that could been have reached by performing forward steps only. That is, causally consistent reversibility does not lead to extraneous states, not reachable through ordinary forward computations.

The quest for causally consistent reversible semantics for (message-passing) concurrency has led to a number of valuable proposals (cf. [5] and references therein). One common drawback in several of those works is that the memories used are rather heavy, and so the resulting reversible semantics are overly complex. This is a particularly notorious drawback in the work of Mezzina and Pérez in [5], which addresses reversibility in the context of concurrent processes whose communication actions are governed by *choreographies* defined by *multiparty session types* [1]. The reversible semantics developed in [5] is causally consistent; however, it is far from clear whether it can be implemented as actual tools for the analysis of message-passing, concurrent programs.

In this paper, we describe a Haskell implementation of the reversible semantics proposed in [5]. More precisely, we present a Haskell interpreter of message-passing programs written in the reversible process framework in [5]. This allows us to assess in practice the benefits and features of the memories and mechanisms deployed in [5] to enforce causally consistent reversibility. In this process, we have found the use of a functional programming language—and in particular, of Haskell—a natural choice. Haskell has a strong history in language design and its mathematical nature allows us to stick closely to the formal semantics.

2 The Process Model

This section summarizes the reversible process model proposed by Mezzina and Pérez in [5], which provides the formal basis for the Haskell implementation that we present in the following section (the main contribution of the present paper).

Fig. 1 depicts the ingredients of our two-level model of *choreographies* and *configurations/processes*. Choreographies are defined in terms of *global types*, which describe a protocol among two or more participants. A global type can be *projected* onto each participant so as to obtain a *local type*, i.e., a session type that abstracts a participant's contribution to the global protocol. (We often use

'choreographies' and 'global types' as synonyms.) The semantics of global types is given in terms of forward and backward transition systems (Fig. 3). There is a *configuration* for each protocol participant: it includes a *located process* that specifies asynchronous communication behavior, subject to a *monitor* that enables forward/backward steps at run-time based on the local type. The semantics of configurations is given in terms of forward and backward reduction relations (Figs. 5 and 6). Below, we use colors to improve readability: elements in blue belong to a forward semantics; elements in red belong to a backward semantics.

2.1 Global and Local Types

2.1.1 Syntax Let us write $\mathtt{p}, \mathtt{q}, \mathtt{r}, \mathtt{A}, \mathtt{B} \ldots$ to denote (protocol) *participants*. The syntax of global types $(G, G', \ldots)$ and local types $(T, T', \ldots)$ follows standard lines [1]:

$$G, G' \;::=\; \mathtt{p} \to \mathtt{q} : \langle U \rangle . G \;\big|\; \mathtt{p} \to \mathtt{q} : \{l_i : G_i\}_{i \in I} \;\big|\; \mu X . G \;\big|\; X \;\big|\; \mathtt{end}$$

$$U, U' \;::=\; \mathtt{bool} \;\big|\; \mathtt{nat} \;\big|\; \cdots \;\big|\; T \to \diamond$$

$$T, T' \;::=\; \mathtt{p}!\langle U \rangle . T \;\big|\; \mathtt{p}?\langle U \rangle . T \;\big|\; \mathtt{p} \oplus \{l_i : T_i\}_{i \in I} \;\big|\; \mathtt{p} \& \{l_i : T_i\}_{i \in I} \;\big|\; \mu X . T \;\big|\; X \;\big|\; \mathtt{end}$$

Global type $\mathtt{p} \to \mathtt{q} : \langle U \rangle . G$ says that $\mathtt{p}$ may send a value of type U to $\mathtt{q}$, and then continue as G. Given a finite index set I and pairwise different *labels* l_i, global type $\mathtt{p} \to \mathtt{q} : \{l_i : G_i\}_{i \in I}$ says that $\mathtt{p}$ may choose label l_i, communicate this selection to $\mathtt{q}$, and then continue as G_i. In these two types we assume that $\mathtt{p} \neq \mathtt{q}$. Global recursive and terminated protocols are denoted $\mu X.G$ and $\mathtt{end}$, respectively. The set $\mathtt{pa}(G)$ contains the participants in G. Value types U include basic first-order values (constants), but also *higher-order* values: abstractions from names to processes. (We write $\diamond$ to denote the type of processes.) Local types $\mathtt{p}!\langle U \rangle . T$ and $\mathtt{p}?\langle U \rangle . T$ denote, respectively, an output and input of value of type U by $\mathtt{p}$. We use α to denote type prefixes $\mathtt{p}?(U)$, $\mathtt{p}!\langle U \rangle$. Type $\mathtt{p} \& \{l_i : T_i\}_{i \in I}$ says that $\mathtt{p}$ offers different behaviors, available as labeled alternatives; conversely, type $\mathtt{p} \oplus \{l_i : T_i\}_{i \in I}$ says that $\mathtt{p}$ may select one of such alternatives. Terminated and recursive local types are denoted $\mathtt{end}$ and $\mu X.T$, respectively.

As usual, we consider only recursive types $\mu X.G$ (and $\mu X.T$) in which X occurs guarded in G (and T). We shall take an equi-recursive view of (global and local) types: we consider two types with the same regular tree as equal.

Global and local types are related by *projection* (Fig. 2): the projection of global type G onto participant $\mathtt{r}$ is written $G{\downarrow_{\mathtt{r}}}$. Intuitively, $G{\downarrow_{\mathtt{r}}}$ denotes the (local) contribution of $\mathtt{r}$ to the overall choreographic behavior that G declaratively specifies.

2.1.2 Semantics of Choreographies The semantics of global types (Fig. 3) comprises forward and backward transition rules. To express backward steps, we

$$(\mathtt{p} \to \mathtt{q} : \langle U \rangle.G)\!\downarrow_{\mathtt{r}} = \begin{cases} \mathtt{q}!\langle U \rangle.(G\!\downarrow_{\mathtt{r}}) & \text{if } \mathtt{r} = \mathtt{p} \\ \mathtt{p}?\langle U \rangle.(G\!\downarrow_{\mathtt{r}}) & \text{if } \mathtt{r} = \mathtt{q} \\ (G\!\downarrow_{\mathtt{r}}) & \text{if } \mathtt{r} \neq \mathtt{q}, \mathtt{r} \neq \mathtt{p} \end{cases}$$

$$(\mathtt{p} \to \mathtt{q} : \{l_i : G_i\}_{i \in I})\!\downarrow_{\mathtt{r}} = \begin{cases} \mathtt{q} \oplus \{l_i : (G_i\!\downarrow_{\mathtt{r}})\}_{i \in I} & \text{if } \mathtt{r} = \mathtt{p} \\ \mathtt{p} \& \{l_i : G_i\!\downarrow_{\mathtt{r}}\}_{i \in I} & \text{if } \mathtt{r} = \mathtt{q} \\ (G_1\!\downarrow_{\mathtt{r}}) & \text{if } \mathtt{r} \neq \mathtt{q}, \mathtt{r} \neq \mathtt{p} \text{ and } \\ & \forall i, j \in I. G_i\!\downarrow_{\mathtt{r}} = G_j\!\downarrow_{\mathtt{r}} \end{cases}$$

$$(\mu X.G)\!\downarrow_{\mathtt{r}} = \begin{cases} \mu X.G\!\downarrow_{\mathtt{r}} & \text{if } \mathtt{r} \text{ occurs in } G \\ \mathsf{end} & \text{otherwise} \end{cases}$$

$$X\!\downarrow_{\mathtt{r}} = X \qquad \mathsf{end}\!\downarrow_{\mathtt{r}} = \mathsf{end}$$

Fig. 2. Projection of a global type G onto a participant $\mathtt{r}$.

$$\text{(FVal1)}\ \mathbb{G}[\mathtt{p} \to \mathtt{q} : \langle U \rangle.G] \hookrightarrow \mathbb{G}[\mathtt{p} \to \mathtt{q} : \langle U \rangle.G]$$

$$\text{(FVal2)}\ \mathbb{G}[\mathtt{p} \to \mathtt{q} : \langle U \rangle.G] \hookrightarrow \mathbb{G}[\mathtt{p} \to \mathtt{q} : \langle U \rangle.G]$$

$$\text{(FCho1)}\ \mathbb{G}[\mathtt{p} \to \mathtt{q} : \{l_i : G_i\}_{i \in I}] \hookrightarrow \mathbb{G}[\mathtt{p} \to \mathtt{q} : \{l_i : G_i\, ;\, l_j : G_j\}_{i \in I \setminus j}]$$

$$\text{(FCho2)}\ \mathbb{G}[\mathtt{p} \to \mathtt{q} : \{l_i : G_i\, ;\, l_j : G_j\}_{i \in I \setminus j}] \hookrightarrow \mathbb{G}[\mathtt{p} \to \mathtt{q} : \{l_i : G_i\, ;\, l_j : G_j\}_{i \in I \setminus j}]$$

$$\text{(BVal1)}\ \mathbb{G}[\mathtt{p} \to \mathtt{q} : \langle U \rangle.G] \rightharpoonup \mathbb{G}[\mathtt{p} \to \mathtt{q} : \langle U \rangle.G]$$

$$\text{(BVal2)}\ \mathbb{G}[\mathtt{p} \to \mathtt{q} : \langle U \rangle.G] \rightharpoonup \mathbb{G}[\mathtt{p} \to \mathtt{q} : \langle U \rangle.G]$$

$$\text{(BCho1)}\ \mathbb{G}[\mathtt{p} \to \mathtt{q} : \{l_i : G_i\, ;\, l_j : G_j\}_{i \in I \setminus j}] \rightharpoonup \mathbb{G}[\mathtt{p} \to \mathtt{q} : \{l_i : G_i\, ;\, l_j : G_j\}_{i \in I \setminus j}]$$

$$\text{(BCho2)}\ \mathbb{G}[\mathtt{p} \to \mathtt{q} : \{l_i : G_i\, ;\, l_j : G_j\}_{i \in I \setminus j}] \rightharpoonup \mathbb{G}[\mathtt{p} \to \mathtt{q} : \{l_i : G_i\, ;\, l_j : G_j\}_{i \in I \setminus j}]$$

Fig. 3. Semantics of Global Types (Forward & Backwards).

require some auxiliary notions. *Global contexts*, ranged over by $\mathbb{G}, \mathbb{G}', \ldots$ with holes $\bullet$, record previous actions, including the choices discarded and committed:

$$\mathbb{G} \quad ::= \quad \bullet \;\big|\; \mathbb{G}[\mathtt{p} \to \mathtt{q} : \langle U \rangle.\mathbb{G}] \;\big|\; \mathbb{G}[\mathtt{p} \to \mathtt{q} : \{l_i : G_i\, ;\, l_j : \mathbb{G}\}_{i \in I \setminus j}]$$

We also use *global types with history*, ranged over by $\mathsf{H}, \mathsf{H}', \ldots$, to record the current protocol state. This state is denoted by the *cursor* $\mathbin{\frown}$:

$$\begin{aligned} \mathsf{H}, \mathsf{H}' \quad ::= \quad & \mathbin{\frown} G \;\big|\; G \mathbin{\frown} \;\big|\; \mathtt{p} \to \mathbin{\frown} \mathtt{q} : \langle U \rangle.G \;\big|\; \mathtt{p} \to \mathtt{q} : \langle U \rangle.\mathbin{\frown} G \\ & \big|\; \mathtt{p} \to \mathbin{\frown} \mathtt{q} : \{l_i : G_i\, ;\, l_j : G_j\}_{i \in I \setminus j} \;\big|\; \mathtt{p} \to \mathtt{q} : \{l_i : G_i\, ;\, l_j : \mathbin{\frown} G_j\}_{i \in I \setminus j} \end{aligned}$$

Intuitively, directed exchanges such as $\mathtt{p} \to \mathtt{q} : \langle U \rangle.G$ have three *intermediate states*, characterized by the decoupled involvement of $\mathtt{p}$ and $\mathtt{q}$ in the intended asynchronous model. The *first state*, denoted $\overset{\frown}{\mathtt{p}} \to \mathtt{q} : \langle U \rangle.G$, describes the situation prior to the exchange. The *second state* represents the point in which $\mathtt{p}$ has sent a value of type U but this message has not yet reached $\mathtt{q}$; this is denoted $\mathtt{p} \to \overset{\frown}{\mathtt{q}} : \langle U \rangle.G$. The *third state* represents the point in which $\mathtt{q}$ has received the message from $\mathtt{p}$ and the continuation G is ready to execute; this is denoted by $\mathtt{p} \to \mathtt{q} : \langle U \rangle.\overset{\frown}{G}$. These intuitions extend to $\mathtt{p} \to \mathtt{q} : \{l_i : G_i\}_{i \in I}$, with the following caveat: the second state should distinguish the choice made by $\mathtt{p}$ from the discarded alternatives; we write $\mathtt{p} \to \overset{\frown}{\mathtt{q}} : \{l_i : G_i \,;\, l_j : G_j\}_{i \in I \setminus j}$ to describe that $\mathtt{p}$ has selected l_j and that this choice is still to be received by $\mathtt{q}$. Once this occurs, a state $\mathtt{p} \to \mathtt{q} : \{l_i : G_i \,;\, l_j : \overset{\frown}{G_j}\}_{i \in I \setminus j}$ is reached.

These intuitions come in handy to describe the forward and backward transition rules in Fig. 3. For a forward directed exchange of a value, Rule (FVal1) formalizes the transition from the first to the second state; Rule (FVal2) denotes the transition from the second to the third state. Rules (FCho1) and (FCho2) are their analogues for the forward directed communication of a label. Rules (BVal1) and (BVal2) undo the step performed by Rules (FVal1) and (FVal2), respectively. Also, Rules (BCho1) and (BCho2) undo the step performed by Rules (FCho1) and (FCho2), respectively.

2.2 Processes and Configurations

2.2.1 Syntax The syntax of processes and configurations is given in Fig. 4. The syntax of configurations builds upon that of processes.

Names a, b, c (resp. s, s') range over shared (resp. session) names. We use session names indexed by participants, denoted $s_{[\mathtt{p}]}, s_{[\mathtt{q}]}$. Names n, m are session or shared names. First-order values v, v' include base values and constants. Variables are denoted by x, y, and recursive variables are denoted by X, Y. The syntax of values V includes shared names, first-order values, but also name abstractions (higher-order values) $\lambda x.\, P$, where P is a process. As shown in [2], abstraction passing suffices to express name passing (*delegation*).

Process terms include prefixes for sending and receiving values V, written $u!\langle V \rangle.P$ and $u?(x).P$, respectively. Given a finite index set I, processes $u \triangleleft \{l_i.P_i\}_{i \in I}$ and $u \triangleright \{l_i : P_i\}_{i \in I}$ implement selection and branching (internal and external labeled choices, respectively). The selection $u \triangleleft \{l_i.P_i\}_{i \in I}$ is actually a non-deterministic choice over I. Here we consider parallel composition of processes $P \mid Q$ and recursion $\mu X.P$ (which binds the recursive variable X in process P). Process $V\, u$ is the application which substitutes name u on the abstraction V. Constructs for name restriction $(\nu\, n)P$ and inaction $\mathbf{0}$ are standard. Session restriction $(\nu\, s)P$ simultaneously binds all the participant endpoints in P. We write $\mathtt{fv}(P)$ and $\mathtt{fn}(P)$ to denote the sets of free variables and names in P. We assume V in $u!\langle V \rangle.P$ does not include free recursive variables X. If $\mathtt{fv}(P) = \emptyset$, we call P *closed*.

$$
\begin{array}{rcl}
u, w & ::= & n \mid x, y, z \qquad n, n' \ ::= \ a, b \mid s_{[\mathtt{p}]} \\
v, v' & ::= & \mathtt{tt} \mid \mathtt{ff} \mid \cdots \\
V, W & ::= & a, b \mid x, y, z \mid v, v' \mid \lambda x.\, P \\
P, Q & ::= & u!\langle V \rangle.P \mid u?(x).P \mid u \triangleleft \{l_i.P_i\}_{i \in I} \mid u \triangleright \{l_i : P_i\}_{i \in I} \\
 & & \mid P \mid Q \mid X \mid \mu X.P \mid V\, u \mid (\nu\, n)P \mid \mathbf{0} \\
M, N & ::= & \ell\, \{a!\langle x \rangle.P\} \mid \ell\, \{a?(x).P\} \mid M \mid N \mid (\nu\, n)M \mid \mathbf{0} \\
 & & \mid \boxed{\ell_{[\mathtt{p}]} : \wr \mathtt{C} \ ; \ P \int} \mid \boxed{s_{[\mathtt{p}]} \lfloor H \cdot \widetilde{x} \cdot \sigma \rfloor^{\spadesuit}} \mid \boxed{s : (h_i \star h_o)} \mid \boxed{k \lfloor (V\, u)\, , \ell \rfloor} \\
\mathtt{C}, \mathtt{C}' & ::= & \mathbf{0} \mid u \triangleleft \{l_i.P_i\}_{i \in I} \mid u \triangleright \{l_i : P_i\}_{i \in I} \mid \mathtt{C}_1, \mathtt{C}_2 \\
\spadesuit & ::= & \blacklozenge \mid \lozenge \qquad h \ ::= \ \epsilon \mid h \circ (\mathtt{p}\, , \mathtt{q}\, , m) \qquad m \ ::= \ V \mid l \\
\alpha & ::= & \mathtt{q}?(U) \mid \mathtt{q}!\langle U \rangle \\
T, S & ::= & \mathtt{end} \mid \alpha.S \mid \mathtt{q} \oplus \{l_i : S_i\}_{i \in I} \mid \mathtt{q} \& \{l_i : S_i\}_{i \in I} \\
H, K & ::= & \frown S \mid S \frown \mid \alpha_1. \cdots .\alpha_n. \frown S \mid \mathtt{q} \oplus \{l_i : S_i \ ; \ l_j : H_j\}_{i \in I} \mid \mathtt{q} \& \{l_i : S_i \, , \, l_j : H_j\}_{i \in I}
\end{array}
$$

Fig. 4. Syntax of processes P, Q, configurations M, N, stacks $\mathtt{C}, \mathtt{C}'$, local types T, S, local types with history H, K. Constructs given in boxes appear only at run-time.

Building upon processes, the syntax of configurations $M, N, \ldots$ includes constructs for *session initiation*:

- configuration $\ell\, \{a!\langle x \rangle.P\}$ denotes the *request* of a service identified with a implemented in P as x;
- conversely, configuration $\ell\, \{a?(x).P\}$ denotes service *acceptance*.

In these constructs, identifiers $\ell, \ell', \ldots$ denote a *location* or *site*. Locations indexed by participants, useful in run-time expressions, are denoted $\ell_{[\mathtt{p}]}, \ell_{[\mathtt{q}]}$. Configurations also include inaction $\mathbf{0}$, parallel composition $M \mid N$, name restriction $(\nu\, n)M$, as well as the following *run-time elements*:

- *Running processes* are of the form $\ell_{[\mathtt{p}]} : \wr \mathtt{C} \ ; \ P \int$, where ℓ is a location that hosts a process P and a *(process) stack* $\mathtt{C}$ that implements participant $\mathtt{p}$. A process stack is a list of processes, useful to record/reinstate the discarded alternatives in a choice.

- *Monitors* are of the form $s_{[\mathtt{p}]}\left\lfloor H \cdot \widetilde{x} \cdot \sigma \right\rfloor^{\spadesuit}$ where s is the session being monitored, $\mathtt{p}$ is a participant, H is a history session type, $\widetilde{x}$ is a set of free variables, and the *store* σ records the value of such variables (see Def. 1). These elements allow us to track the current protocol and state of the monitored process.
  Also, each monitor has a *tag* $\spadesuit$, which can be either *empty* (denoted '$\Diamond$') or *full* (denoted '$\blacklozenge$'). When created all monitors have an empty tag; a full tag indicates that the running process associated to the monitor is currently involved in a decoupled reversible step. We often omit the empty tag (so we write $s_{[\mathtt{p}]}\left\lfloor H \cdot \widetilde{x} \cdot \sigma \right\rfloor$ instead of $s_{[\mathtt{p}]}\left\lfloor H \cdot \widetilde{x} \cdot \sigma \right\rfloor^{\Diamond}$) and write $s_{[\mathtt{p}]}\left\lfloor H \cdot \widetilde{x} \cdot \sigma \right\rfloor^{\blacklozenge}$ to emphasize the reversible (red) nature of a monitor with full tag.
- Following [3], we have *message queues* of the form $s : (h_i \star h_o)$, where s is a session, h_i is the input part of the queue, and h_o is the output part of the queue. Each queue contains messages of the form $(\mathtt{p}, \mathtt{q}, m)$ (read: "message m is sent from $\mathtt{p}$ to $\mathtt{q}$"). The effect of an output prefix in a process is to place the message in its corresponding output queue; conversely, the effect of an input prefix is to obtain the first message from its input queue. Messages in the queue are *never consumed*: a process reads a message $(\mathtt{p}, \mathtt{q}, m)$ by moving it from the (tail of) queue h_o to the (top of) queue h_i. This way, the delimiter '$\star$' distinguishes the *past* of the queue from its *future*.
- We use *running functions* of the form $k\left\lfloor (V\,u)\,,\,\ell \right\rfloor$ to reverse applications $V\,u$. While k is a fresh identifier (key) for this term, ℓ is the location of the running process that contains the application.

We shall write $\mathcal{P}$ and $\mathcal{M}$ to indicate the set of processes and configurations, respectively. We call *agent* an element of the set $\mathcal{A} = \mathcal{M} \cup \mathcal{P}$. We let P, Q to range over $\mathcal{P}$; also, we use L, M, N to range over $\mathcal{M}$ and A, B, C to range over $\mathcal{A}$.

2.2.2 A Decoupled Semantics for Configurations We define a reduction relation on configurations, coupled with a structural congruence on processes and configurations. Our reduction semantics defines a *decoupled* treatment for reversing communication actions within a protocol. Reduction is thus defined as $\longrightarrow \subset \mathcal{M} \times \mathcal{M}$, whereas structural congruence is defined as $\equiv \subset \mathcal{P}^2 \cup \mathcal{M}^2$. We require auxiliary definitions for *contexts*, *stores*, and *type contexts*.

Evaluation contexts are configurations with one hole '$\bullet$', defined by:

$$\mathbb{E} \;::=\; \bullet \;\big|\; M \mid \mathbb{E} \;\big|\; (\nu\, n)\, \mathbb{E}$$

General contexts $\mathbb{C}$ are processes or configurations with one hole $\bullet$: they are obtained by replacing one occurrence of $\mathbf{0}$ (as a process or as a configuration) with $\bullet$. A congruence on processes and configurations is an equivalence $\Re$ that is closed under general contexts: $P\,\Re\,Q \implies \mathbb{C}[P]\,\Re\,\mathbb{C}[Q]$ and $M \Re N \implies \mathbb{C}[M]\,\Re\,\mathbb{C}[N]$. We rely on a notion of structural congruence on processes and configurations,

denoted $\equiv$ and defined in [5]. A relation $\Re$ on configurations is *evaluation-closed* if it satisfies the following rules:

$$(\textsc{Ctx})\ \frac{M\,\Re\,N}{\mathbb{E}[M]\,\Re\,\mathbb{E}[N]} \qquad (\textsc{Eqv})\ \frac{M \equiv M' \qquad M'\,\Re\,N' \qquad N' \equiv N}{M\,\Re\,N}$$

The state of monitored processes is formalized as follows:

Definition 1. *A store σ maps variables to values. Given a store σ, a variable x, and a value V, the* update $\sigma[x \mapsto V]$ *and the* reverse update $\sigma \setminus x$ *are defined as:*

$$\sigma[x \mapsto V] = \begin{cases} \sigma \cup \{(x,V)\} & \text{if } x \notin \mathtt{dom}(\sigma) \\ \textit{undefined} & \textit{otherwise} \end{cases} \qquad \sigma \setminus x \quad = \begin{cases} \sigma_1 & \text{if } \sigma = \sigma_1 \cup \{(x,V)\} \\ \sigma & \textit{otherwise} \end{cases}$$

Together with local types with history, the following notion of type context allows us to keep the current protocol state inside monitors:

Definition 2. *Let $k, k', \ldots$ denote fresh name identifiers. We define type contexts as (local) types with one hole, denoted "$\bullet$":*

$$\begin{aligned} \mathbb{T},\mathbb{S} \ ::= \ & \bullet \mid \mathtt{q} \oplus \{l_w : \mathbb{T}\,;\ l_i : S_i\}_{i \in I \setminus w} \mid \mathtt{q}\&\{l_w : \mathbb{T}\,,\ l_i : S_i\}_{i \in I \setminus w} \\ & \mid \alpha.\mathbb{T} \mid k.\mathbb{T} \mid (\ell, \ell_1, \ell_2).\mathbb{T} \end{aligned}$$

Type contexts $k.\mathbb{T}$ and $(\ell, \ell_1, \ell_2).\mathbb{T}$ will be instrumental in formalizing reversibility of name applications and thread spawning, respectively, which are not described by local types. This way, we will often have monitors of the form $s_{[\mathtt{p}]} \lfloor \mathbb{T}\,[\frown S] \cdot \widetilde{x} \cdot \sigma \rfloor^{\spadesuit}$, where $\mathbb{T}$ and S describe past and future protocol steps for $\mathtt{p}$, respectively.

Abstraction passing can implement a form of *session delegation*, for received abstractions $\lambda x.\, P$ can contain free session names (indexed by participant identities). The following definition identifies those names:

Definition 3. *Let h and $\mathtt{p}$ be a queue and a participant, respectively. Also, let $\{(\mathtt{q}_1\,,\,\mathtt{p}\,,\,\lambda x_1.\, P_1), \ldots, (\mathtt{q}_k\,,\,\mathtt{p}\,,\,\lambda x_k.\, P_k)\}$ denote the (possibly empty) set of messages in h containing abstractions sent to $\mathtt{p}$. We write* $\mathbf{roles}(\mathtt{p}, h)$ *to denote the set of participant identities occurring in $P_1, \ldots, P_k$.*

The reduction relation $\longrightarrow$ is defined as the union of the forward and backward reduction relations, denoted $\twoheadrightarrow$ and $\rightsquigarrow$, respectively. That is, $\longrightarrow\, = \,\twoheadrightarrow \cup \rightsquigarrow$. Relations $\twoheadrightarrow$ and $\rightsquigarrow$ are the smallest evaluation-closed relations satisfying the rules in Figs. 5 and 6. We indicate with $\longrightarrow^*$, $\twoheadrightarrow^*$, and $\rightsquigarrow^*$ the reflexive and transitive closure of $\longrightarrow$, $\twoheadrightarrow$ and $\rightsquigarrow$, respectively. We first discuss the forward reduction rules (Fig. 5):

$$\text{(Init)}\ \frac{\mathtt{pa}(G)=\{\mathtt{p}_1,\cdots,\mathtt{p}_n\} \qquad \forall \mathtt{p}_i \in \mathtt{pa}(G).\, G{\downarrow_{\mathtt{p}_i}} = T_i}{\ell_1\,\{a!\langle x_1:T_1\rangle.P_1\} \mid \prod_{i\in\{2,\cdots,n\}} \ell_i\,\{a?(x_i:T_i).P_i\}}$$
$$\twoheadrightarrow$$
$$(\nu\, s)\left(\prod_{i\in\{1,\cdots,n\}} \ell_{i\,[\mathtt{p}_i]} : \jmath\mathbf{0} \fatsemi P_i\{s_{[\mathtt{p}_i]}/x_i\}\jmath \mid s_{[\mathtt{p}_i]}\left\lfloor \frown T_i \cdot x_i \cdot [x_i \mapsto a]\right\rfloor \mid s : (\epsilon \star \epsilon)\right)$$

$$\text{(Out)}\ \frac{\mathtt{p}=\mathtt{r} \vee \mathtt{p}\in \mathbf{roles}(\mathtt{r},h_i)}{\ell_{[\mathtt{r}]} : \jmath\mathtt{C} \fatsemi s_{[\mathtt{p}]}!\langle V\rangle.P\jmath \mid s_{[\mathtt{p}]}\left\lfloor \mathbb{T}\,[\frown \mathtt{q}!\langle U\rangle.S]\cdot\widetilde{x}\cdot\sigma\right\rfloor \mid s : (h_i \star h_o)}$$
$$\twoheadrightarrow$$
$$\ell_{[\mathtt{r}]} : \jmath\mathtt{C} \fatsemi P\jmath \mid s_{[\mathtt{p}]}\left\lfloor \mathbb{T}\,[\mathtt{q}!\langle U\rangle.\frown S]\cdot\widetilde{x}\cdot\sigma\right\rfloor \mid s : (h_i \star h_o \circ (\mathtt{p}\,,\,\mathtt{q}\,,\,\sigma(V)))$$

$$\text{(In)}\ \frac{\mathtt{p}=\mathtt{r} \vee \mathtt{p}\in \mathbf{roles}(\mathtt{r},h_i)}{\ell_{[\mathtt{r}]} : \jmath\mathtt{C} \fatsemi s_{[\mathtt{p}]}?(y).P\jmath \mid s_{[\mathtt{p}]}\left\lfloor \mathbb{T}\,[\frown \mathtt{q}?\langle U\rangle.S]\cdot\widetilde{x}\cdot\sigma\right\rfloor \mid s : (h_i \star (\mathtt{q}\,,\,\mathtt{p}\,,\,V)\circ h_o)}$$
$$\twoheadrightarrow$$
$$\ell_{[\mathtt{r}]} : \jmath\mathtt{C} \fatsemi P\jmath \mid s_{[\mathtt{p}]}\left\lfloor \mathbb{T}\,[\mathtt{q}?\langle U\rangle.\frown S]\cdot\widetilde{x}, y\cdot\sigma[y\mapsto V]\right\rfloor \mid s : (h_i \circ (\mathtt{q}\,,\,\mathtt{p}\,,\,V)\star h_o)$$

$$\text{(Sel)}\ \frac{\mathtt{p}=\mathtt{r} \vee \mathtt{p}\in \mathbf{roles}(\mathtt{r},h_i) \qquad w\in J \qquad J\subseteq I}{\ell_{[\mathtt{r}]} : \jmath\mathtt{C} \fatsemi s_{[\mathtt{p}]} \triangleleft \{l_i.P_i\}_{i\in I}\jmath \mid s_{[\mathtt{p}]}\left\lfloor \mathbb{T}\,[\frown \mathtt{q}\oplus\{l_j : S_j\}_{j\in J}]\cdot\widetilde{x}\cdot\sigma\right\rfloor \mid s : (h_i \star h_o)}$$
$$\twoheadrightarrow$$
$$\ell_{[\mathtt{r}]} : \jmath\mathtt{C}, s_{[\mathtt{p}]} \triangleleft \{l_i.P_i\}_{i\in I\setminus w} \fatsemi P_w\jmath \mid s_{[\mathtt{p}]}\left\lfloor \mathbb{T}\left[\mathtt{q}\oplus\{l_j : S_j\,,\, l_w : \frown S_w\}_{j\in J\setminus w}\right]\cdot\widetilde{x}\cdot\sigma\right\rfloor \mid s : (h_i \star h_o \circ (\mathtt{p}\,,\,\mathtt{q}\,,\,l_w))$$

$$\text{(Bra)}\ \frac{\mathtt{p}=\mathtt{r} \vee \mathtt{p}\in \mathbf{roles}(\mathtt{r},h_i) \qquad w\in I \qquad I\subseteq J}{\ell_{[\mathtt{r}]} : \jmath\mathtt{C} \fatsemi s_{[\mathtt{p}]} \triangleright \{l_i : P_i\}_{i\in I}\jmath \mid s_{[\mathtt{p}]}\left\lfloor \mathbb{T}\,[\frown \mathtt{q}\&\{l_j : S_j\}_{j\in J}]\cdot\widetilde{x}\cdot\sigma\right\rfloor \mid s : (h_i \star (\mathtt{q}\,,\,\mathtt{p}\,,\,l_w)\circ h_o)}$$
$$\twoheadrightarrow$$
$$\ell_{[\mathtt{r}]} : \jmath\mathtt{C}, s_{[\mathtt{p}]} \triangleright \{l_i : P_i\}_{i\in I\setminus w} \fatsemi P_w\jmath \mid s_{[\mathtt{p}]}\left\lfloor \mathbb{T}\left[\mathtt{q}\&\{l_j : S_j\,,\, l_w : \frown S_w\}_{j\in J\setminus w}\right]\cdot\widetilde{x}\cdot\sigma\right\rfloor \mid s : (h_i \circ (\mathtt{q}\,,\,\mathtt{p}\,,\,l_w)\star h_o)$$

$$\text{(Beta)}\ \frac{\sigma(V)=\lambda x.\,P}{\ell_{[\mathtt{p}]} : \jmath\mathtt{C} \fatsemi (V\,w)\jmath \mid s_{[\mathtt{p}]}\left\lfloor \mathbb{T}\,[\frown S]\cdot\widetilde{x}\cdot\sigma\right\rfloor \twoheadrightarrow (\nu\, k)\left(\ell_{[\mathtt{p}]} : \jmath\mathtt{C} \fatsemi P\{\sigma(w)/x\}\jmath \mid k\left\lfloor (V\,w)\,,\,\ell\right\rfloor \mid s_{[\mathtt{p}]}\left\lfloor \mathbb{T}\,[k.\frown S]\cdot\widetilde{x}\cdot\sigma\right\rfloor\right)}$$

$$\text{(Spawn)}\ \frac{}{\ell_{[\mathtt{p}]} : \jmath\mathtt{C} \fatsemi P \mid Q\jmath \mid s_{[\mathtt{p}]}\left\lfloor \mathbb{T}\,[\frown S]\cdot\widetilde{x}\cdot\sigma\right\rfloor \twoheadrightarrow (\nu\,\ell_1,\ell_2)\left(\ell_{[\mathtt{p}]} : \jmath\mathtt{C} \fatsemi \mathbf{0}\jmath \mid \ell_{1\,[\mathtt{p}]} : \jmath\mathbf{0} \fatsemi P\jmath \mid \ell_{2\,[\mathtt{p}]} : \jmath\mathbf{0} \fatsemi Q\jmath \mid s_{[\mathtt{p}]}\left\lfloor \mathbb{T}\,[(\ell,\ell_1,\ell_2).\frown S]\cdot\widetilde{x}\cdot\sigma\right\rfloor\right)}$$

Fig. 5. Decoupled semantics for configurations: Forward reduction ($\twoheadrightarrow$).

▶ Rule (Init) initiates a choreography G with n participants. Given the composition of one service request and $n-1$ service accepts (all along a, available

in different locations ℓ_i), this rule sets up the run-time elements: running processes and monitors—one for each participant, with empty tag (omitted)—and the empty session queue. A unique session identifier (s in the rule) is also created. The processes are inserted in their respective running structures, and instantiated with an appropriate session name. Similarly, the local types for each participant are inserted in their respective monitor, with the cursor $\overset{\frown}{}$ at the beginning.

- Rule (Out) starts the output of value V from $\mathtt{p}$ to $\mathtt{q}$. Given an output-prefixed process as running process, and a monitor with a local type supporting an output action, reduction adds the message $(\mathtt{p}, \mathtt{q}, \sigma(V))$ to the output part of the session queue (where σ is the current store). Also, the cursor within the local type is moved accordingly. In this rule (but also in several other rules), premise $\mathtt{p} = \mathtt{r} \vee \mathtt{p} \in \mathbf{roles}(\mathtt{r}, h_i)$ allows performing actions on names previously received via abstraction passing.
- Rule (In) allows a participant $\mathtt{p}$ to receive a value V from $\mathtt{q}$: it simply takes the first element of the output part of the queue and places it in the input part. The cursor of the local type and state in the monitor for $\mathtt{p}$ are updated accordingly.
- Rule (Sel) is the forward rule for labeled selection, which in our case entails a non-deterministic choice between pairwise different labels indexed by I. We require that I is contained in J, i.e., the set that indexes the choice according to the choreography. After reduction, the selected label (l_w in the rule) is added to the output part of the queue, and the continuation P_w is kept in the running process; to support reversibility, alternatives different from l_w are stored in the stack $\mathtt{C}$ with their continuations. The cursor is also appropriately updated in the monitor.
- Rule (Bra) is similar to Rule (Sel): it takes a message containing a label l_w as the first element in the output part of the queue, and places it into the input part. This entails a selection between the options indexed by I; the continuation P_w is kept in the running process, and all those options different from l_w are kept in the stack. Also, the local type in the monitor is updated accordingly.
- Rule (Beta) handles name applications. Reduction creates a fresh identifier (k in the rule) for the running function, which keeps (i) the structure of the process prior to application, and (ii) the identifier of the running process that "invokes" the application. Notice that k is recorded also in the monitor: this is necessary to undo applications in the proper order. To determine the actual abstraction and the name applied, we use σ.
- Rule (Spawn) handles parallel composition. Location ℓ is "split" into running processes with fresh identifiers (ℓ_1, ℓ_2 in the rule). This split is recorded in the monitor.

Now we comment on the backward rules (Fig. 6) which, in most cases, change the monitor tags from $\Diamond$ into $\blacklozenge$:

- Rule (RInit) undoes session establishment. It requires that local types for every participant are at the beginning of the protocol, and empty session queue and process stacks. Run-time elements are discarded; located service accept/requests are reinstated.
- Rule (RollS) starts to undo an input-output synchronization between `p` and `q`. Enabled when there are complementary session types in the two monitors, this rule changes the monitor tags from ◊ to ⧫. This way, the undoing of input and output actions occurs in a decoupled way. Rule (RollC) is the analog of (RollS) but for synchronizations originated in labeled choices.
- Rule (ROut) undoes an output. This is only possible for a monitor tagged with ⧫, exploiting the first message in the input queue. After reduction, the process prefix is reinstated, the cursor is adjusted, the message is removed from the queue, the monitor is tagged again with ◊. Rule (RIn) is the analog of Rule (ROut). In this case, we also need to update the state of store σ.
- Rule (RBra) undoes the input part of a labeled choice: the choice context is reinstated; the cursor is moved; the last message in the input part of the queue is moved to the output part.
- Rule (RSel) is the analog of (RBra), but for the output part of the labeled choice. The non-deterministic selection is reinstated.
- Rule (RBeta) undoes β-reduction, reinstating the application. The running function disappears, using the information in the monitor (k in the rule). Rule (RSpawn) undoes the spawn of a parallel thread, using the identifiers in the monitor.

3 Our Haskell Implementation

We set out to implement the language, types and semantics given above. The end goal is to implement the two stepping functions

```
forward :: Location -> Participant -> Session Value ()
backward :: Location -> Participant -> Session Value ()
```

Where `Session` contains an `ExecutionState` holding among other things a store of variables, and can fail producing an `Error`.

```
type Session value a = StateT (ExecutionState value) (Except Error) a
```

Additionally we need to provide a program for every participant, a monitor for every participant and a global message queue. All three of those need to be able to move forward and backward.

The rest of this section is structured as follows: Section 3.1 describes the implementation of global and local types, 3.2 describes the process calculus and 3.3

gives a more convenient syntax for writing programs in that calculus. 3.4 covers how reversibility is implemented, and finally 3.5 combines all the pieces.

The code shown in this section is available at https://github.com/folkertdev/reversible-debugger

3.1 Global and Local Types

The Global type describes interactions between participants. The definition of global types is given by

```
type GlobalType u = Fix (GlobalTypeF u)

data GlobalTypeF u next
    = Transaction
        { from :: Participant, to :: Participant, tipe :: u, continuation ::  next }
    | Choice
        { from :: Participant, to :: Participant, options :: Map String next }
    | R next
    | V
    | Wk next
    | End
    deriving (Show, Functor)
```

The recursive constructors are taken from [6]. `R` introduces a recursion point, `V` jumps back to a recursion point and `Wk` weakens the recursion, making it possible to jump to a less tightly-binding `R`.

A global type can be projected onto a participant, resulting in that participant's local type. The local type describes interactions between a participant and the central message queue. Specifically, sends and receives, and offers and selects. We use the three buyer example from [5] as a running example here. The projection of `globalType` onto `A` is equivalent to this pseudo-code of `derivedTypeForA`.

```
data MyParticipants = A | B | C | V deriving (Show, Eq, Ord)

data MyType = Title | Price | Share | Ok | Thunk | Address | Date
    deriving (Show, Eq, Ord)

globalType :: GlobalType.GlobalType MyParticipants MyType
globalType = GlobalType.globalType $ do
    GlobalType.transaction A V Title
    GlobalType.transactions V [A, B] Price
    GlobalType.transaction A B Share
    GlobalType.transactions B [A, V] Ok
```

```
    GlobalType.transaction B C Share
    GlobalType.transaction B C Thunk
    GlobalType.transaction B V Address
    GlobalType.transaction V B Date
    GlobalType.end

derivedTypeForA :: LocalType MyType
derivedTypeForA = do
    send V Title
    receive V Price
    send B Share
    receive B Ok
```

3.2 A Language

We need a language to use with our types. It needs at least instructions for the four participant-queue interactions, a way to assign variables, and a way to define and apply functions.

```
type Participant = String
type Identifier = String

type Program value = Fix (ProgramF value)

data ProgramF value next
    -- transaction primitives
    = Send { owner :: Participant, value :: value, continuation :: next }
    | Receive { owner :: Participant, variableName :: Identifier, continuation :: next  }

    -- choice primitives
    | Offer Participant (List (String, next))
    | Select Participant (List (String, value, next))

    -- other constructors
    | Parallel next next
    | Application Identifier value
    | Let Identifier value next
    | IfThenElse value next next
    | Literal value
    | NoOp
    deriving (Eq, Show, Functor)


data Value
```

```
    = VBool Bool
    | VInt Int
    | VString String
    | VUnit
    | VIntOperator Value IntOperator Value
    | VComparison Value Ordering Value
    | VFunction Identifier (Program Value)
    | VReference Identifier
    | VLabel String
    deriving (Eq, Show)
```

In the definition of `ProgramF`, the recursion is factored out and replaced by a type parameter. The `Fix` type reintroduces the ability to create to create arbitrarily deep trees of instructions. The advantage of using `Fix` is that we can use recursion schemes - like folds - on the structure.

Given a `LocalType` and a `Program`, we can now step forward through the program. For each instruction, we check the session type to see whether the instruction is allowed.

3.3 An eDSL with the free monad

We create an embedded domain-specific language (eDSL) using the free monad to write our example programs. This method allows us to use haskell's do-notation, which is much more convenient than writing programs with `Fix`.

The free monad is a monad that comes for free given some functor. The `ProgramF` type is specifically created in such a way that it is a functor in `next`. The idea then is to use the free monad on our `ProgramF` data type to be able to build a nice DSL.

For the transformation from `Free (ProgramF value) a` back to `Fix (ProgramF value)` we need also need some state: a variable counter that allows us to produce new unique variable names.

```
newtype HighLevelProgram a =
    HighLevelProgram (StateT (Location, Participant, Int) (Free (ProgramF Value)) a)
        deriving (Functor, Applicative, Monad, MonadState (Location, Participant, Int))

uniqueVariableName :: HighLevelProgram Identifier
uniqueVariableName = do
    (location, participant, n) <- State.get
    State.put (location, participant, n + 1)
    return $ "var" ++ show n
```

```
send :: Value -> HighLevelProgram ()
send value = do
    (_, participant, _) <- State.get
    HighLevelProgram $ lift $ liftFree (Send participant value ())

receive :: HighLevelProgram Value
receive = do
    (_, participant, _) <- State.get
    variableName <- uniqueVariableName
    HighLevelProgram $ lift $ liftFree (Receive participant variableName ())
    return (VReference variableName)

terminate :: HighLevelProgram a
terminate = HighLevelProgram (lift $ Free NoOp)
```

We can now write a correct implementation of `A`s local type.

```
aType :: LocalType MyType
aType = do
    send V Title
    receive V Price
    send B Share
    receive B Ok

alice = H.compile "Location1" "A" $ do
    let share = VInt 42
    H.send (VString "address" )
    price <- H.receive
    H.send share
    ok <- H.receive
    H.terminate
```

`HighLevelProgram` is transformed into a `Program` by evaluating the `StateT`, this puts the correct owner and unique variable names into the tree, and then transforming `Free` to `Fix` by putting `NoOp` at the leaves where needed.

```
freeToFix :: Free (ProgramF value) a -> Program value
freeToFix (Pure n) = Fix NoOp
freeToFix (Free x) = Fix (fmap freeToFix x)

compile :: Location -> Participant -> HighLevelProgram a -> Program value
compile location participant (HighLevelProgram program) =
    freeToFix $ runStateT program (location, participant, 0)
```

3.3.1 Ownership The `owner` field for send, receive, offer and select is makes sure that instructions in closures are attributed to the correct participant.

```
bob = H.compile "Location1" "B" $ do
    thunk <-
        H.function $ \_ -> do
            H.send (VString "Lucca, 55100")
            d <- H.receive
            H.terminate

    price <- H.receive
    share <- H.receive
    let verdict = price `H.lessThan` VInt 79
    H.send verdict
    H.send verdict
    H.send share
    H.send thunk


carol = H.compile "Location1" "C" $ do
    h <- H.receive
    code <- H.receive
    H.applyFunction code VUnit
```

Here `B` creates a function that performs a send and receive. Because the function is created by `B`, the owner of these statements is `B`, even when the function is sent to and eventually evaluated by `C`.

The design of the language and semantics poses some further issues. With the current mechanism of storing applications, functions have to be named. Hence `H.function` cannot produce a simple value, because it needs to assign to a variable and thereby update the state.

It essential that all references in the function body are dereferenced before sending. References that remain are invalid at the receiver, causing undefined behavior.

3.4 Reversibility

Every forward step needs an inverse. The inverse needs to contain all the information needed to recreate the instruction and local type that performed the forward step.

```
type TypeContext program value a = Fix (TypeContextF program value a)

data TypeContextF program value a f
```

```
  = Hole
  | LocalType (LocalTypeF a ()) f
  | Selected
      { owner :: Participant
      , offerer :: Participant
      , selection :: Zipper (String, value, program, LocalType a)
      , continuation :: f
      }
  | Offered
      { owner :: Participant
      , selector :: Participant
      , picked :: Zipper (String, program, LocalType a)
      , continuation :: f
      }
  | Branched
      { condition :: value
      , verdict :: Bool
      , otherBranch :: program
      , continuation :: f
      }
  | Application Identifier Identifier f
  | Assignment
      { visibleName :: Identifier
      , internalName :: Identifier
      , continuation :: f
      }
  | Literal a f
  deriving (Eq, Show, Generic, Functor)
```

For **send/receive** and **offer/select**, the instructions that modify the queue, we must also roll the queue. Additionally, both participants must be synchronized. Synchronization ensures that both parties roll their parts, but the rolling can still happen in a decoupled way. The synchronization is a dynamic check that throws an error message if either participant is not in the expected state.

Rolling a **let** removes the assigned variable from the store. While strictly necesarry to maintain causal consistency, it is good practice.

Function applications are treated exactly as in the formal semantics: A reference is stored to the function and its arguments, so we can recreate the application later.

Given a `LocalType` and a `Program` we can now step through the program while producing a trace through the execution. At any point, we can move back to a previous state.

3.5 Putting it all together

At the start we described the `Session` type.

```
type Session value a = StateT (ExecutionState value) (Except Error) a
```

We now have all the pieces we need to define the execution state. Based on the error conditions that arise in moving forward and backward, we can also define a meaningful `Error` type.

3.5.1 The Monitor A participant is defined by its monitor and its program. The monitor contains various metadata about the participant: variables, the current state of the type and some other information to be able to move backward.

In the formal semantics we have defined a monitor as a tagged triplet containing a history session type, a set of free variables and a store assigning these variables to values. In the haskell implementation, the tag is moved into the history type `LocalTypeState`. The set of free variables and the store is merged into a dictionary. The set of variables is the set of keys of the dictionary.

```
data Monitor value tipe =
    Monitor
        { _localType :: LocalTypeState (Program value) value tipe
        , _recursiveVariableNumber :: Int
        , _recursionPoints :: List (LocalType tipe)
        , _store :: Map Identifier value
        , _applicationHistory :: Map Identifier (Identifier, value)
        }
        deriving (Show, Eq)
```

Additionally, the application history is folded into the monitor as this is a participant-specific piece of data. Similarly, the monitor keeps track of recursion points to restore the local type when a recursive step is reversed.

3.5.2 ExecutionState The execution state contains the monitors and the programs, but also

- A variable and application counter to generate unique new names
- The central message queue.
- The `_isFunction` field, a function that extracts the function body used for function application.

```haskell
data Queue a = Queue
    { history :: List ( Participant, Participant, a)
    , current :: List (Participant, Participant, a)
    }
    deriving (Eq, Show)

data ExecutionState value =
    ExecutionState
        { _variableCount :: Int
        , _applicationCount :: Int
        , _participants :: Map Participant (Monitor value String)
        , _locations :: Map Location (Map Participant (Program value))
        , _queue :: Queue value
        , _isFunction :: value -> Maybe (Identifier, Program value)
        }
```

3.5.3 Error generation There are a lot of potential failure conditions in this system. A small error somewhere in either the global type or the program can quickly move program and type out of sync.

Having descriptive error messages that provide a lot of context makes it easier to fix these issues.

```haskell
data Error
    = UndefinedParticipant Participant
    | UndefinedVariable Participant Identifier
    | SynchronizationError String
    | LabelError String
    | QueueError String Queue.QueueError
    deriving (Eq, Show)
```

Not all errors are fatal. For instance, it is possible that one participant expects to receive the value, but it's not been sent yet. This will give a queue error - because the correct value is not in the queue - but the program and the type are perfectly fine. Implementing a workflow for error recovery is left as future work.

3.5.4 Stepping functions We can now implement the stepping functions

```haskell
forward :: Location -> Participant -> Session ()
backward :: Location -> Participant -> Session ()
```

A sketch of the implementation for forward:

- Given a `Location` and a `Participant` we can query the `ExecutionState` to get a `Program` and a `Monitor`.
- We pattern match on the program and the local type. If then match then we get into the business logic, otherwise we throw an error.
- The business logic must construct a new program and a new monitor. In the below example, a variable assignment is evaluated. Assignments don't affect the local type so the type is matchted against the wildcard `_`.

Next the assignment is added to the history type, then the monitor is updated with this new local type and an updated store with the new variable. The remaining program is updated by renaming the variable.

```
(Let visibleName value continuation, _) -> do
    variableName <- uniqueVariableName

    let newLocalType =
            LocalType.createState
                (LocalType.assignment visibleName variableName previous)
                fixedLocal

        renamedValue = renameValue visibleName variableName value
        newMonitor =
            monitor
                { _store = Map.insert variableName renamedValue (_store monitor)
                , _localType = newLocalType
                }

    setParticipant location participant
        ( newMonitor, renameVariable visibleName variableName continuation )
```

4 Concluding Remarks and Future Work

Our next step is to fully integrate recursion and choice. The combination of recrusion and choice with more than 2 participants won't always work because the choice is not sent to other participants. When the selected branch recurses, the other participants won't know about it. This quickly results in errors.

We're also looking at providing a better user interface for the project as a whole and for resolving errors in particular.

With regards to theory, we would like to be able to step though a program by individual global protocol actions. The local types and programs steps would automatically be generated from a global action. Additionally, we're experimenting with making informed decisions when one branch of a choice fails.

References

1. K. Honda, N. Yoshida, and M. Carbone. Multiparty Asynchronous Session Types. In *POPL'08*, pages 273–284. ACM, 2008.
2. D. Kouzapas, J. A. Pérez, and N. Yoshida. On the relative expressiveness of higher-order session processes. In P. Thiemann, editor, *ESOP 2016*, volume 9632 of *LNCS*, pages 446–475. Springer, 2016.
3. D. Kouzapas, N. Yoshida, R. Hu, and K. Honda. On asynchronous eventful session semantics. *Mathematical Structures in Computer Science*, 26(2):303–364, 2016.
4. I. Lanese, C. A. Mezzina, and F. Tiezzi. Causal-consistent reversibility. *Bulletin of the EATCS*, 114, 2014.
5. C. A. Mezzina and J. A. Pérez. Causally consistent reversible choreographies: a monitors-as-memories approach. In W. Vanhoof and B. Pientka, editors, *Proceedings of the 19th International Symposium on Principles and Practice of Declarative Programming, Namur, Belgium, October 09 - 11, 2017*, pages 127–138. ACM, 2017.
6. F. v. Walree. Session types in cloud haskell. 2017.

$$\text{(RInit)}\ \frac{\mathtt{pa}(G)=\{\mathtt{p}_1,\cdots,\mathtt{p}_n\} \qquad \forall \mathtt{p}_i \in \mathtt{pa}(G).\, G{\downarrow_{\mathtt{p}_i}} = T_i \qquad Q_i = P_i\{s_{[\mathtt{p}_i]}/x_i\}}{(\nu\, s)\left(\prod_{i\in\{1,\cdots,n\}} \ell_{i[\mathtt{p}_i]} : \mathbb{0}\ ;\ Q_i \int \ \mid s_{[\mathtt{p}_i]}\left\lfloor \frown T_i \cdot x_i \cdot [x_i \mapsto a]\right\rfloor^{\Diamond} \mid s : (\epsilon \star \epsilon)\right) \rightsquigarrow \ell_1\,\{a!\langle x_1 : T_1\rangle.P_1\} \mid \prod_{i\in\{2,\cdots,n\}} \ell_i\,\{a?(x_i : T_i).P_i\}}$$

$$\text{(RollS)}\ \frac{}{s_{[\mathtt{p}]}\left\lfloor \mathbb{T}\,[\mathtt{q}?\langle U\rangle.\frown T]\cdot\widetilde{x}\cdot\sigma_1\right\rfloor^{\Diamond} \mid s_{[\mathtt{q}]}\left\lfloor \mathbb{S}\,[\mathtt{p}!\langle U\rangle.\frown S]\cdot\widetilde{y}\cdot\sigma_2\right\rfloor^{\Diamond} \mid s : (h_i \star h_o) \rightsquigarrow s_{[\mathtt{p}]}\left\lfloor \mathbb{T}\,[\mathtt{q}?\langle U\rangle.\frown T]\cdot\widetilde{x}\cdot\sigma_1\right\rfloor^{\blacklozenge} \mid s_{[\mathtt{q}]}\left\lfloor \mathbb{S}\,[\mathtt{p}!\langle U\rangle.\frown S]\cdot\widetilde{y}\cdot\sigma_2\right\rfloor^{\blacklozenge} \mid s : (h_i \star h_o)}$$

$$\text{(RollC)}\ \frac{}{\begin{array}{c} s_{[\mathtt{p}]}\left\lfloor \mathbb{T}\left[\mathtt{q}\&\{l_z : \frown S_z\,,\, l_w : S_w\}_{z\in J\setminus w}\right]\cdot\widetilde{x}\cdot\sigma_1\right\rfloor^{\Diamond} \mid \\ s_{[\mathtt{q}]}\left\lfloor \mathbb{S}\left[\mathtt{p}\oplus\{l_z : \frown S_z\,,\, l_w : S_w\}_{z\in J\setminus w}\right]\cdot\widetilde{y}\cdot\sigma_2\right\rfloor^{\Diamond} \mid s : (h_i \star h_o) \end{array} \rightsquigarrow \begin{array}{c} s_{[\mathtt{p}]}\left\lfloor \mathbb{T}\left[\mathtt{q}\&\{l_z : \frown S_z\,,\, l_w : S_w\}_{z\in J\setminus w}\right]\cdot\widetilde{x}\cdot\sigma_1\right\rfloor^{\blacklozenge} \mid \\ s_{[\mathtt{q}]}\left\lfloor \mathbb{S}\left[\mathtt{p}\oplus\{l_z : \frown S_z\,,\, l_w : S_w\}_{z\in J\setminus w}\right]\cdot\widetilde{y}\cdot\sigma_2\right\rfloor^{\blacklozenge} \mid s : (h_i \star h_o) \end{array}}$$

$$\text{(ROut)}\ \frac{\mathtt{p}=\mathtt{r} \ \vee\ \mathtt{p}\in \mathbf{roles}(\mathtt{r}, h_i)}{\ell_{[\mathtt{r}]} : \mathbb{C}\ ;\ P \int \ \mid s_{[\mathtt{p}]}\left\lfloor \mathbb{T}\,[\mathtt{q}!\langle U\rangle.\frown S]\cdot\widetilde{x}\cdot\sigma\right\rfloor^{\blacklozenge} \mid s : (h_i \star (\mathtt{p},\mathtt{q},V)\circ h_o) \rightsquigarrow \ell_{[\mathtt{r}]} : \mathbb{C}\ ;\ s_{[\mathtt{p}]}!\langle V\rangle.P \int \ \mid s_{[\mathtt{p}]}\left\lfloor \mathbb{T}\,[\frown \mathtt{q}!\langle U\rangle.S]\cdot\widetilde{x}\cdot\sigma\right\rfloor^{\Diamond} \mid s : (h_i \star h_o)}$$

$$\text{(RIn)}\ \frac{\mathtt{p}=\mathtt{r} \ \vee\ \mathtt{p}\in \mathbf{roles}(\mathtt{r}, h_i)}{\ell_{[\mathtt{r}]} : \mathbb{C}\ ;\ P \int \ \mid s_{[\mathtt{p}]}\left\lfloor \mathbb{T}\,[\mathtt{q}?\langle U\rangle.\frown S]\cdot\widetilde{x}, y\cdot\sigma\right\rfloor^{\blacklozenge} \mid s : (h_i \circ (\mathtt{q},\mathtt{p},V) \star h_o) \rightsquigarrow \ell_{[\mathtt{r}]} : \mathbb{C}\ ;\ s_{[\mathtt{p}]}?(y).P \int \ \mid s_{[\mathtt{p}]}\left\lfloor \mathbb{T}\,[\frown \mathtt{q}?\langle U\rangle.S]\cdot\widetilde{x}\cdot\sigma \setminus y\right\rfloor^{\Diamond} \mid s : (h_i \star (\mathtt{q},\mathtt{p},V)\circ h_o)}$$

$$\text{(RBra)}\ \frac{\mathtt{p}=\mathtt{r} \ \vee\ \mathtt{p}\in \mathbf{roles}(\mathtt{r}, h_i) \qquad w\in I \qquad I\subseteq J}{\begin{array}{c} \ell_{[\mathtt{r}]} : \mathbb{C}, s_{[\mathtt{p}]} \triangleright \{l_i : P_i\}_{i\in I\setminus\{w\}}\ ;\ P \int \ \mid s_{[\mathtt{p}]}\left\lfloor \mathbb{T}\left[\mathtt{q}\&\{l_j : S_j\,,\, l_w : \frown S_w\}_{j\in J\setminus w}\right]\cdot\widetilde{x}\cdot\sigma\right\rfloor^{\blacklozenge} \\ \mid s : (h_i \circ (\mathtt{q}\,,\,\mathtt{p}\,,\,l_w) \star h_o) \end{array} \rightsquigarrow \begin{array}{c} \ell_{[\mathtt{r}]} : \mathbb{C}\ ;\ s_{[\mathtt{p}]} \triangleright \{l_i : P_i, l_w : P\}_{i\in I\setminus\{w\}} \int \ \mid s_{[\mathtt{p}]}\left\lfloor \mathbb{T}\left[\frown \mathtt{q}\&\{l_j : S_j\}_{j\in J}\right]\cdot\widetilde{x}\cdot\sigma\right\rfloor^{\Diamond} \mid \\ s : (h_i \star (\mathtt{q}\,,\,\mathtt{p}\,,\,l_w)\circ h_o) \end{array}}$$

$$\text{(RSel)}\ \frac{\mathtt{p}=\mathtt{r} \ \vee\ \mathtt{p}\in \mathbf{roles}(\mathtt{r}, h_i) \qquad w\in I \qquad I\subseteq J}{\ell_{[\mathtt{r}]} : \mathbb{C}, s_{[\mathtt{p}]} \triangleleft \{l_i.P_i\}_{i\in I}\ ;\ P \int \ \mid s_{[\mathtt{p}]}\left\lfloor \mathbb{T}\left[\mathtt{q}\oplus\{l_j : S_j\,,\, l_w : \frown S_w\}_{j\in J\setminus w}\right]\cdot\widetilde{x}\cdot\sigma\right\rfloor^{\blacklozenge} \mid s : (h_i \star (\mathtt{p}\,,\,\mathtt{q}\,,\,l_w)\circ h_o) \rightsquigarrow \ell_{[\mathtt{r}]} : \mathbb{C}\ ;\ s_{[\mathtt{p}]} \triangleleft \{l_w.P\} + s_{[\mathtt{p}]} \triangleleft \{l_i.P_i\}_{i\in I} \int \ \mid s_{[\mathtt{p}]}\left\lfloor \mathbb{T}\,[\frown \mathtt{q}\oplus\{l_j : S_j\}_{j\in J}]\cdot\widetilde{x}\cdot\sigma\right\rfloor^{\Diamond} \mid s : (h_i \star h_o)}$$

$$\text{(RBeta)}\ \frac{}{(\nu\, k)\left(\ell_{[\mathtt{p}]} : \mathbb{C}\ ;\ Q \int \mid k\left\lfloor (V\,w)\,,\,\ell\right\rfloor \mid s_{[\mathtt{p}]}\left\lfloor \mathbb{T}\,[k.\frown S]\cdot\widetilde{x}\cdot\sigma\right\rfloor\right) \rightsquigarrow \ell_{[\mathtt{p}]} : \mathbb{C}\ ;\ (V\,w) \int \ \mid s_{[\mathtt{p}]}\left\lfloor \mathbb{T}\,[\frown S]\cdot\widetilde{x}\cdot\sigma\right\rfloor}$$

$$\text{(RSpawn)}\ \frac{}{\begin{array}{c} (\nu\,\ell_1,\ell_2)\left(\ell_{[\mathtt{p}]} : \mathbb{C}\ ;\ \mathbb{0} \int \mid \ell_{1[\mathtt{p}]} : \mathbb{0}\ ;\ P \int \ \mid \ell_{2[\mathtt{p}]} : \mathbb{0}\ ;\ Q \int \ \mid s_{[\mathtt{p}]}\left\lfloor \mathbb{T}\,[(\ell,\ell_1,\ell_2).\frown S]\cdot\widetilde{x}\cdot\sigma\right\rfloor\right) \rightsquigarrow \\ \ell_{[\mathtt{p}]} : \mathbb{C}\ ;\ P \mid Q \int \ \mid s_{[\mathtt{p}]}\left\lfloor \mathbb{T}\,[\frown S]\cdot\widetilde{x}\cdot\sigma\right\rfloor \end{array}}$$

Fig. 6. Decoupled semantics for configurations: Backwards reduction ($\rightsquigarrow$).