1 Extended Multiparty Session Type

An open automaton is a model of a software component or a process that can be modularized. In an open automaton's perspective, another open automaton that can communicate with it is of either of three types: its parent, one of its siblings, one of its children. We add a type self, which indicates the open automaton itself. We assume that the names of open automata in these types. A straightforward way is to index the parent by ∞ (the "farthest" automaton that can be reached), siblings by $\mathbb{N}^+ = \{1, 2, \ldots\}$ (the external), children by $\mathbb{N}^- = \{-1, -2, \ldots\}$ (the internal) and self by 0. The set of all indices is denoted by $\mathbb{Z} \cup \{\infty\}$. Let us take the convention $0 \cdot \infty = 0$ due to the convenience of using it in the constraint of the syntax.

1.1 Syntax

We want to make use of global type as a description of communication for transitions of open automata. In this way, given an open automaton A, a projection should be applicable to checking if an open automaton A' can be composed to a hole of A. However, since the projection is a binary session type, it cannot be projected anymore. Hence, if A' itself can have holes, we cannot further check if another open automaton can be composed to one of its hole. Therefore, we will develop a class of extended multiparty session types with the purpose that the projection of an extended multiparty session type to one of its party is also an extended multiparty session type. Then we will later define the notions of a sequence of transitions compatible with an extended multiparty session type, and of an open automaton compatible with an extended multiparty session type. From now on, if there is no ambiguity arising, we will simply call a extended multiparty session type a type and an open automaton an automaton.

$$\begin{array}{rcll} T & ::= & S & \text{(session type)} \\ & | & D & \text{(data type)} \end{array}$$

$$\begin{array}{rcll} S & ::= & \texttt{end} & \text{(termination)} \\ & | & L.S & \text{(sequence)} \\ & | & \mu\alpha.S & \text{(recursion)} \\ & | & \alpha & \text{(type variable)} \\ & | & (S + S) & \text{(nondeterminism)} \\ & | & (L_1|L_2|\ldots|L_n).S & \text{(asynchrony)} \end{array} \qquad \begin{array}{rcl} L & ::= & p \to q(D) \\ & | & \texttt{skip} \\ p, q & \in & \mathbb{Z} \cup \{\infty\}, pq \geq 0 \\ D & ::= & \ell|\texttt{bool}|\texttt{int}|\texttt{string}|\ldots \\ \ell & \in & \mathcal{L} \end{array}$$

Figure 1: The syntax of extended multiparty session type

The syntax of the extended multiparty session type is shown in Figure 1. The first four rules are the same as those in literature of multiparty session types [1]. We add a rule for `skip`. The rule for nondeterminism signifies the fact that at a configuration, there can be multiple possible transitions. We assume a set $\mathcal{L}$ of labels. In this way, we can use this rule as a generalization of the branching rule [1]. The rule for asynchrony is to model the fact that each transition of an open automaton can have more than one hole actions. This means we wait for the hole actions to execute asynchronously until they are done, to go along the transition. Denote by $\mathcal{S}$ the set of extended multiparty session types.

1.2 Type Equivalence

We define the subsequence relation $\subset$ in $\mathcal{P}(\mathcal{S}^2)$ as in Figure 2. Denote by $\not\subset$ the relation $\mathcal{P}(\mathcal{S}^2) \setminus \subset$. Intuitively, given $S_1 \subset S_2$, if an open automaton A can execute a sequence of transitions compatible with S_2, then there is a subsequence of these transitions which is compatible with S_1. Using the subsection relation, we define type equivalence as the equivalence relation $\equiv$ on $\mathcal{S}$ satisfying the rules given in Figure 3. The equivalence relation means that, given $S_1 \equiv S_2$, the automaton A can execute a sequence of transitions compatible with S_1 if and only if it can execute a sequence of transitions compatible with S_2.

$$\begin{array}{lcl} S & \subset & S \\ L & \subset & L.S \\ S & \subset & L.S \\ S_j & \subset & p \to q\{\ell_i : S_i\}_{i\in I}, \forall j \in I \\ S & \subset & \mu\alpha.S \end{array} \qquad \begin{array}{lcl} S_1 & \subset & (S_1 + S_2) \\ S_2 & \subset & (S_1 + S_2) \\ L_j & \subset & (L_1|L_2|\ldots|L_n).S, j \in \{1,\ldots,n\} \\ S & \subset & (L_1|L_2|\ldots|L_n).S \end{array}$$

$$\frac{S_1 \subset S_2 \quad S_2 \subset S_3}{S_1 \subset S_3}$$

Figure 2: The subsequence relation

$$\begin{array}{lcll} S & \equiv & S.\texttt{skip} & [\textsc{LSkip}] \\ S & \equiv & \texttt{skip}.S & [\textsc{RSkip}] \\ S.(S_1+S_2) & \equiv & S.S_1 + S.S_2 & [\textsc{Dist}] \\ \mu\alpha.S & \equiv & S, \alpha \not\subset S & [\textsc{Jump}] \\ S.\mu\alpha.S.\alpha & \equiv & \mu\alpha.S.\alpha & [\textsc{Fold}] \end{array} \qquad \begin{array}{c} \dfrac{S_1 \equiv S_2}{S.S_1 \equiv S.S_2} \\ \dfrac{S_1 \equiv S_2}{S_1.S \equiv S_2.S} \end{array}$$

Figure 3: The equivalence relation

1.3 Subtype

The operational semantics of multiparty session types is a directed graph having each transition of the form `end`, L or $L_1|\ldots|L_n$. Let $\mathrm{gr}(S)$ be the graph generated by S. A trace of $\mathrm{gr}(S)$ is a sequence $L_1, \ldots, L_n$, where there is an edge from L_i to L_{i+1}, L_1 is the initial node and L_n is a terminal node. A path is a sequence of edges connecting a trace. A type S_1 is a subtype of S_2, denoted by $S_1 \prec S_2$, if every path of S_1 is a path of S_2.

1.4 Projection

A projection is the conversion of a component's perspective on how to communicate to the its child's perspective. The projection only depends on the piece $p \to q$ every time it appears in a session type. In particular, a sub-component can only see a type $p \to q(D)$ if it is either p or q. Hence, let us first ly define an index projection function for each $p \in \mathbb{N}^-$ $\downharpoonright_p$: $(\mathbb{Z} \cup \{\infty\})^2 \to (\mathbb{Z} \cup \{\infty\})^2 \cup \{\varnothing\}$, mapping each pair $m \to n$ to either a pair $m' \to n'$, or the empty tuple $\varnothing$.

$$p \to q \downharpoonright_p \; = \begin{cases} 0 \to \infty, & \text{if } q = 0 \\ 0 \to -q, & \text{if } q < 0 \\ \varnothing, & \text{otherwise} \end{cases} \qquad q \to p \downharpoonright_p \; = \begin{cases} \infty \to 0, & \text{if } q = 0 \\ -q \to 0, & \text{if } q < 0 \\ \varnothing, & \text{otherwise} \end{cases}$$

$$m \to n \downharpoonright_p = \varnothing, \text{ if } m, n \neq p.$$

We slightly abuse the notation $\downharpoonright_p$ to define inductively the projection function $\downharpoonright_p$: $\mathcal{S} \to \mathcal{S}$ for each $p \in \mathbb{N}^-$.

2 Typed Open Automata

Originally, each transition of an open automaton has the form $\dfrac{\beta_j^{j\in J}, g, \phi}{s_1 \xrightarrow{\alpha} s_2}$, where

- $s_1 \xrightarrow{\alpha} s_2$ denotes a move from a state s_1 to a state s_2 emitting an action α

$$\begin{aligned}
\texttt{end} \downharpoonright_p &= \texttt{end}\\
L.S \downharpoonright_p &= L \downharpoonright_p .S \downharpoonright_p\\
(\mu\alpha.S) \downharpoonright_p &= \mu\alpha.S \downharpoonright_p\\
(S+S) \downharpoonright_p &= (S \downharpoonright_p + S \downharpoonright_p)\\
(L|L).S \downharpoonright_p &= (L \downharpoonright_p | L \downharpoonright_p).S \downharpoonright_p
\end{aligned}$$

$$\begin{aligned}
p \to q\{\ell_i : S_i\}_{i\in I} \downharpoonright_p &= \begin{cases} p \to q \downharpoonright_p \{\ell_i : S_i \downharpoonright_p\}_{i\in I}, \text{ if } p \to q \downharpoonright_p \neq \varnothing \\ \texttt{skip}, \text{ otherwise}\end{cases}\\
p \to q(D) \downharpoonright_p &= \begin{cases} p \to q \downharpoonright_p (D), \text{ if } p \to q \downharpoonright_p \neq \varnothing \\ \texttt{skip}, \text{ otherwise}\end{cases}
\end{aligned}$$

Figure 4: The projection function

- $\beta_j^{j\in J}$ is a set of hole actions, identified by a hole name and an action name. For example, it the automaton has hole P, then we can write a hole action as $P \mapsto \texttt{send}(m)$ to indicate that P emits the action `send` with parameter m.
- g is a predicate indicating if a move can execute once the hole actions execute.
- ϕ is a reassignment of the variables of the automaton if the move happens successfully.

If a hole P executes an internal computation, an unobservable action τ is usually emitted during the move. However, we note that the parent component should only be able to see the message passing actions of a child component i.e. a hole. Once a specify automaton is composed to the hole can the parent see all its internal actions. Therefore, we modify each hole action to the form $p \to q(m : D)$ for a message m of type D sent from p to q, or $p \to q(\ell)$ for a label sent from p to q. We come to the definition of a typed open automaton.

Definition 2.1 *A typed open automaton is a tuple $A = \langle S, s_0, E, V, \phi_0, T\rangle$, where*

- *S is the set of states*
- *$s_0 \in S$ is the initial state*
- *$E \subset S$ is the set of terminal states*
- *V is the set of variables*
- *$\psi_0 : V \to \mathcal{P}$ is the initial assignment*
- *T is the set of transitions. Each $t \in T$ has the form $\dfrac{\beta_j^{j\in J}, g, \psi}{s \xrightarrow{\alpha} s'}$, where*
  - *$s, s' \in S$ and α is an emitted action.*
  - *each β_j has the form $p \to q(m : D)$ or $p \to q(\ell)$ such that $p, q \in \mathbb{Z} \cup \infty$ and $\ell \in \mathcal{L}$.*
  - *g is a predicate over V*
  - *$\psi : V \to \mathcal{E}_V$ is a reassignment*

We can ignore the emitted action and write $s \xrightarrow{\beta_j^{j\in J}, g, \psi} s'$. A pair (s, ϕ), where $s \in S$ and $\phi : V \to \mathcal{P}$ is called a configuration of the automaton.

Remark 2.2. A transition has no hole actions indicates an internal computing. A transition with more than one action indicates asynchrony.

Example 2.3. Consider the producer-consumer communication through a three-slot buffer. This can be modelled as an automaton $A = \langle S, s_0, E, V, \phi_0, T\rangle$, where

- $S = E = \{s_0\}$
- $V = \{M, f, l\}$
- $\psi_0 : M \mapsto [0,0,0], f \mapsto 0, l \mapsto 0$
- $T = \left\{ s_0 \xrightarrow{\{-1\to 0(m:\texttt{int})\}((l+1)\%3\neq f)\left\{\substack{M[l]\leftarrow m \\ l\leftarrow (l+1)\%3}\right\}} s_0, s_0 \xrightarrow{\{0\to -2(M[f]:\texttt{int})\}(f\neq l)\{f\leftarrow f+1\}} s_0 \right\}$

In Example 2.3, we have used the index -1 for the producer and -2 for the consumer. The automaton acts as a *mediator*. On one hand, it receives a message from the producer, decide if the message can still be stored. On the other hand, if there is a message available in the buffer, it lets the consumer receive.

3 Type Inferred by an Automaton

Taking the transitions of an automaton, if we only look at the states and hole actions, we can generate a directed graph where

- Each node is a *state* of the automaton
- Each edge is inferred from the hole actions, where $p \to q(m : D)$ becomes $p \to q(D)$ and $p \to q(\ell)$ is kept the same.

The type with this graph is called the weak type of the automaton. On the other hand, if we take into account the guard and reassignment, we can generate a graph whose nodes are *configurations* of the automaton. *If such generating process halts*, we get the strong type of the automaton.

Proposition 3.1 *For every type S, there exists an automaton A with no variables inferring S. We also have that S is the strong type of A.*

Note that open automata with no holes are already Turing-complete. The reason is that each transition simulates a GOTO statement, which is conditioned by the guard, and there are also variable reassignments. On the other hand, session type supporting up to recursion is not Turing-complete. Hence the generating process from an open automaton to a type does not always halt. However, I will try to show that maybe

- Non-halting means that the automaton has an infinite-sized variable.
- If not, we can abstract one more level to "match" different configurations, so that the set of reachable abstracted configurations is finite.

This suggests that we can use open automata to verify further properties than data type compatibility. For example, adding lists of transferred frames to check if they arrive in order.

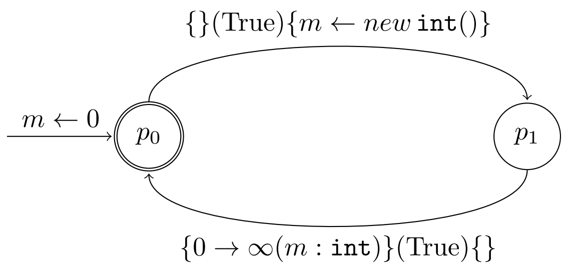
Taking Example 2.3, let $U = -1 \to 0(\texttt{int})$ and $V = 0 \to -2(\texttt{int})$, then the weak type of A is

$$W = \mu\alpha.(U + V).\alpha,$$

and the strong type of A is

$$S = \mu s_0.(U.\mu s_1.(V.s_0 + U.\mu s_2.(V.s_1 + U\mu s_3.V.s_2))).$$

Note that the type S is generated from an automaton with four states and one variable M. We should develop a calculus to show that $S \prec W$ because the definition based on directed graph is not computable. Maybe there is already such calculus in type theory textbooks.



$\{\}(\text{True})\{m \leftarrow \textit{new}\, \texttt{int}()\}$
$m \leftarrow 0$
p_0
p_1
$\{0 \to \infty(m : \texttt{int})\}(\text{True})\{\}$

Figure 5: An automaton implementing the producer in Example 2.3

Let $U' = 0 \to \infty(\texttt{int})$ and $V' = \infty \to 0(\texttt{int})$, we have the following projections

$$W \downharpoonleft_{-1} = \mu\alpha.U'.\alpha$$

$$\begin{aligned}
S \downharpoonleft_{-1} &= \mu s_0.(U'.\mu s_1.(s_0 + U'.\mu s_2.(s_1 + U'\mu s_3.s_2))) \\
&\equiv \mu s_0.(U'.\mu s_1.(s_0 + U'.\mu s_2.(s_1 + U's_2))) && [\textsc{Jump}] \\
&\equiv \mu s_0.(U'.\mu s_1.(s_0 + U'.\mu s_2.s_1 + U'\mu s_2.U's_2)) && [\textsc{Dist}] \\
&\equiv \mu s_0.(U'.\mu s_1.(s_0 + U'.\mu s_2.s_1 + \mu s_2.U's_2)) && [\textsc{Fold}] \\
&\equiv \mu s_0.(U'.\mu s_1.(s_0 + U'.s_1 + \mu s_2.U's_2)) && [\textsc{Jump}] \\
&\dots \\
&\equiv \mu\alpha.U'.\alpha
\end{aligned}$$

Although the strong and weak types are different, their projections onto a child component are the same. This means that any producer able to produce message should be compatible to be compose to the child component.

In addition, strong inference is similar to that we have simulated the automaton, so it ensures deadlock freedom.

4 Composition

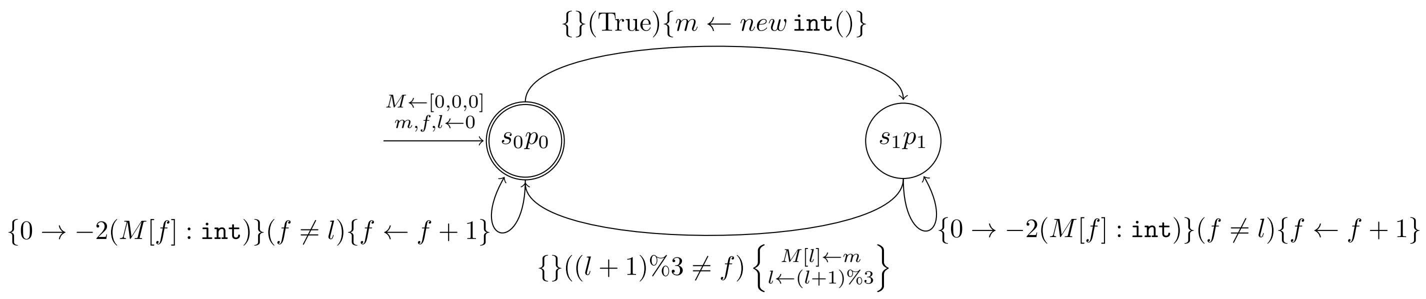
To compose an automaton A' to a hole of A, the idea is to compose all compatible actions, reserve all internal actions (of both A and A'), and rename internal and external communications if there is any conflict. I will elaborate more.

Example 4.1. Consider an implementation of the producer as the open automaton P depicted in Figure 5. The strong or weak type of P is $\mu\alpha.U'.\alpha$, so it can be composed to the internal component -1 of A.

Consider Figure 7. The type of P is now $\mu\alpha. - 1 \to 0(\texttt{int}).U'.\alpha$. However, P should also be able to be composed to A.

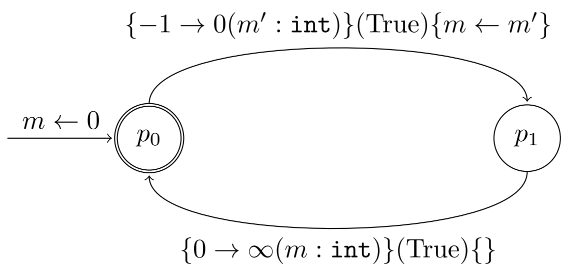
5 Plans

1. Refine the notion of composition and compatibility
2. Find conditions where strong inference is decidable
3. I think that for every open automaton, there exists a single-state automaton equivalent to it (just by let one more variable for state). But automata with more states and fewer variables are of



$\{\}(\text{True})\{m \leftarrow \textit{new}\,\texttt{int}()\}$
$M \leftarrow [0,0,0]$
$m,f,l \leftarrow 0$
s_0p_0
s_1p_1
$\{0 \to -2(M[f] : \texttt{int})\}(f \neq l)\{f \leftarrow f+1\}$
$\{\}((l+1)\%3 \neq f)\left\{\begin{smallmatrix} M[l] \leftarrow m \\ l \leftarrow (l+1)\%3 \end{smallmatrix}\right\}$
$\{0 \to -2(M[f] : \texttt{int})\}(f \neq l)\{f \leftarrow f+1$

Figure 6: The composed automaton



$\{-1 \to 0(m' : \texttt{int})\}(\text{True})\{m \leftarrow m'\}$
$m \leftarrow 0$
p_0
p_1
$\{0 \to \infty(m : \texttt{int})\}(\text{True})\{\}$

Figure 7: A modification of P having internal communication

course easier to verify. I will investigate whether this has something to do with bisimulation after two objectives above are done.