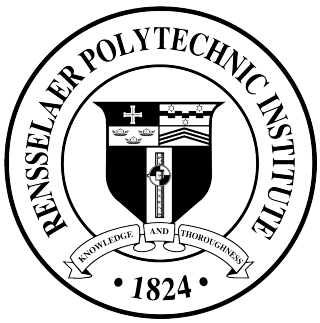
DEADLOCK FREEDOM IN ACTOR LANGUAGES

Andrew Wilkerson

Submitted in Partial Fullfillment of the Requirements
for the Degree of

MASTER OF SCIENCE

Approved by:
Carlos Varela, Chair
Ana Milanova
Stacy Patterson



Department of Computer Science
Rensselaer Polytechnic Institute
Troy, New York

[December 2023]

CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

We introduce a framework using session types for denoting the relationships between multiple actors in a system. We prove that ascribing to this framework guarantees deadlock freedom, and demonstrate tests of how an implementation of such would work in the SALSA language.

1. INTRODUCTION

Session types [1] can be used as a way to reason about the communications between two processes. For example, one process could `send(a)`, `send(b)`, then either `receive(c)` or `receive(d)`, then `end`. The process opposite it would, in turn, have the session type: `receive(a)`, `receive(b)`, (`send(c)` or `send(d)`), `end`. One important point to note is that these two processes have *dual session types*, meaning that for one process and any state it is in, the other process is in the opposite state. This creates a number of useful properties, including the guarantee of deadlock freedom [2].

While previous work has been done in adapting session types into Erlang [3], we seek to show the benefits of using an actor language with a significantly different structure, namely SALSA [4]. One of the largest differences is the fact that messages are received individually by message handlers in SALSA, meaning you are guaranteed to be able to receive a message of any type at any time, but are unable to restrict the types of messages you can receive. Contrast this with Erlang, in which upon receiving a message, the language allows the code to block until it receives another message only of a specific type. Additionally, SALSA uses *tokens* to elegantly wait for multiple messages being processed in parallel.

Deadlock is an important problem, as code that potentially enters deadlock may enter an erroneous state and be unable to inform the user that it is unable to process information. Session types are a good solution to this problem, as they model the communication topology between two agents, and deadlock is fundamentally a problem caused by badly formed communication topologies. Furthermore, session types will type check at compile time, meaning it can provide these guarantees without the user needing to run code.

The main contributions of this thesis are threefold. First, we introduce a language, which we call Session Typed FeatherWeight SALSA, that models session types and executes similarly to FeatherWeight SALSA [4]. Next, we define the semantics of this language and demonstrate how such a model would be able to detect deadlock. Finally, we prove that defining the program like so guarantees our type checker is sound, and finds any potential deadlock at compile time.

2. RELATED WORK

Mostrous and Vasconcelos [5] showed that it is possible to construct a program in an actor language in a way that can use session types. To do this, they defined a featherweight version of Erlang, and used the notion of unique references to verify the identity of the sender and recipient. They explained how each function in their language would translate into session types, and proved various properties about programs created under this framework. For example, they defined how communications match, or become balanced, and that under their environment this relationship would be preserved, thus guaranteeing type safety.

Neykova and Yoshida [6] showed how to use session types for a group of actors, rather than just a pair. Their research hinges on establishing an overhead organization framework that defines the session types between roles, and having the programs participating in the session type choose the role they plan to fulfill.

Reynolds [7] introduced the notion of *separation logic* as an extension of Hoare logic, specifically for reasoning about parallel programs. Separation logic denotes dependency on certain portions of shared memory, and how interactions between those portions affects the state of the other program threads.

Jacobs et. al. [8] used separation logic to form *connectivity graphs* based on a given program. Each process was represented by a vertex, with directed edges marking dependencies. From there, they were able to determine if the program had entered a deadlock state by checking the graph for cycles, as that would indicate a cyclic dependency.

Carbone and Debois [2] proved that utilizing session types could prevent deadlock, using the notion that a program that uses session types always maintains both progress and preservation. They used π *Calculus* and the notion of dual types to show that the dependency graph of a program would never enter deadlock if it adhered to session types.

Charalambides et. al. [9] introduced the notion of *parameterized session types* for multi-party interactions. They showed how languages may be able to reason about asynchronicity, while also verifying concurrent, parameterized programs. To do so, they created a language of their own that could demonstrate such properties, and proved that every facet of session types was upheld by how the language defined its interactions. From there, they formalized their type checking and showed that the types were projectable, or able to be reduced to simpler types, and proved the correctness of this result.

3. MOTIVATING EXAMPLE

We explore a version of the *dining philosophers* problem with only two chopsticks and two philosophers. The goal is for each philosopher to be able to continuously eat using the chopsticks on either side of them, while also allowing the other philosopher to grab them as well once they are finished. A common problem is for both philosophers to grab the left chopstick first, then wait for the right chopstick, which will never be put down by the other philosopher, causing both to starve. A common solution to this (per Dijkstra [10]) is to have one philosopher prioritize their left chopstick, and the other to prioritize their right.

Section 3.2 shows the complete code for a well-written implementation of the dining philosophers problem. The code as given ensures fairness between the two philosophers (one philosopher will not eat over and over while the other is waiting for chopsticks), as well as guaranteeing it is impossible for them to deadlock. We prove that this is the case, as well as show that swapping the priority of the second philosopher's chopsticks leads to the possibility for deadlock.

Note that in a more minimal example, the chopsticks would act similarly to semaphores, in that they would not queue up the next philosopher to grab automatically. However, the communication topology of the more minimal example is more complex as a result, as attempting to grab a chopstick results in either success or failure. Since our work focuses on ensuring properties about the communication topology, our example focuses on the implementation of the problem with more straightforward session types (the fairness version), as there is only one messaging result of attempting to grab a chopstick (success).

This program can be difficult to reason about, because it includes a variety of interesting portions. For example, the program is designed to continuously loop, rather than reach a terminal state; we will represent this in our session types using the Kleene star (*). In addition, the program utilizes tokens, meaning we must find a way to denote that the tokens have been received and the program can progress. This is done by treating the tokens themselves as dependencies within the graphical representation, which unblock upon reception of the corresponding message. Furthermore, the program features actors which use internal variables to decide which control flow to take upon receiving certain messages. We use Hoare logic [11] to guarantee each of these possibilities is accounted for. Finally, the program is made up of four actors, rather than the two typical for session types. In

order to reason about this, we represent the program as a whole using a multiparty session type [6], which is set up within the initialization. Additionally, we model the changes in the communication topology as a series of graphs where a cyclical dependency in one indicates potential deadlock [2].

Figure 3.1 shows a simplified version of the program, where only one philosopher is involved. Note that there is an ambiguous ordering of which chopstick is released first, yet both traces lead back to the same state.

3.1 In Practice

Looking back at our dining philosophers example, we hope to see that a small change in its setup (grabbing left instead of right) will lead to the type checker detecting deadlock and vice versa.

When we initialize Philosopher `p2`, swapping the order of the chopsticks has no effect on the session types themselves. However, when we check the initialization, we record the state of the Philosopher with respect to its relationships to the Chopsticks. Specifically, it records `c2` as the first chopstick and `c1` as the second (the opposite occurs in the well-written case).

Additionally, the first get message sent as `p2` must be to its first chopstick. The checker asserts the message sent first is the same as the first message of the session type, and thus will only be able to confirm a result if this is the case.

With this new setup, it will check all possible traces of the program. One such trace is shown in Figure 3.2, which deadlocks. Hence, the type checker confirms the program potentially enters deadlock, and can provide a trace that does so.

With the well-written setup, no such trace produces deadlock. A sample walk of the well-written program is shown in Figures 3.3 and 3.4. In this case, it encounters a previously cached state at every leaf node, meaning that no matter which trace it takes, it will always be able to recurse.

3.2 Dining Philosophers Code

```
behavior Chopstick{
    Philosopher pa;
```

```
Philosopher pb;
boolean a_holding;
boolean b_holding;
boolean a_waiting;
boolean b_waiting;

Chopstick(){}

void setPhilosophers(Philosopher a, Philosopher b){
    pa = a;
    pb = b;
    a_holding = false;
    b_holding = false;
    a_waiting = false;
    b_waiting = false;
    return null;
}

//@   requires SessionType[pa] get;
//@   requires SessionType[pb] get;
//@   ensures receive[pa] get;
//@   ensures send[pa] holding;
//@   ensures SessionType[pa] holding;
//@   ensures SessionType[pb] get;
//@ also
//@   requires SessionType[pa] get;
//@   requires SessionType[pb] holding;
//@   ensures receive[pa] get;
//@   ensures SessionType[pa] holding;
//@   ensures SessionType[pb] holding;
//@ also
//@   requires SessionType[pa] get;
```

```
//@   requires SessionType[pb] release;
//@   ensures receive[pa] get;
//@   ensures SessionType[pa] holding;
//@   ensures SessionType[pb] release;
//@ also

//@   requires SessionType[pa] get;
//@   requires SessionType[pb] get;
//@   ensures receive[pb] get;
//@   ensures send[pb] holding;
//@   ensures SessionType[pa] get;
//@   ensures SessionType[pb] holding;
//@ also
//@   requires SessionType[pa] holding;
//@   requires SessionType[pb] get;
//@   ensures receive[pb] get;
//@   ensures SessionType[pa] holding;
//@   ensures SessionType[pb] holding;
//@ also
//@   requires SessionType[pa] release;
//@   requires SessionType[pb] get;
//@   ensures receive[pb] get;
//@   ensures SessionType[pa] release;
//@   ensures SessionType[pb] holding;
void get(Philosopher p){
    a_holding = a_holding || (p==pa && !b_holding);
    b_holding = b_holding || (p==pb && !a_holding);
    a_waiting = p==pa && b_holding;
    b_waiting = p==pb && a_holding;

    if (p==pa && a_holding){
        token t0 = pa <- holding():waitfor();
```

```
        return null;
    } else if (p==pb && b_holding) {
        token t0 = pb <- holding():waitfor();
        return null;
    } else return null;
}

//@   requires SessionType[pa] release;
//@   requires SessionType[pb] get;
//@   ensures receive[pa] release;
//@   ensures SessionType[pa] get;
//@   ensures SessionType[pb] get;
//@ also
//@   requires SessionType[pa] release;
//@   requires SessionType[pb] holding;
//@   ensures receive[pa] release;
//@   ensures send[pb] holding;
//@   ensures SessionType[pa] get;
//@   ensures SessionType[pb] holding;
//@ also

//@   requires SessionType[pa] get;
//@   requires SessionType[pb] release;
//@   ensures receive[pb] release;
//@   ensures SessionType[pa] get;
//@   ensures SessionType[pb] get;
//@ also
//@   requires SessionType[pa] holding;
//@   requires SessionType[pb] release;
//@   ensures receive[pb] release;
//@   ensures send[pa] holding;
//@   ensures SessionType[pa] holding;
```

```
    //@   ensures SessionType[pb] get;
    void release(){
        a_holding = a_waiting;
        b_holding = b_waiting;
        a_waiting = false;
        b_waiting = false;

        if (a_holding) {
            token t0 = pa <- holding():waitfor();
            return null;
        } else if (b_holding) {
            token t0 = pb <- holding():waitfor();
            return null;
        } else return null;
    }
}

behavior Philosopher{
    Chopstick first;
    Chopstick second;
    int sticks;

    Philosopher(Chopstick first_, Chopstick second_){
        this.first = first_;
        this.second = second_;
        this.sticks = 0;
    }

    //@   requires SessionType[first] holding;
    //@   requires SessionType[second] get;
    //@   ensures receive[first] holding;
    //@   ensures send[second] get;
```

```
    //@   ensures SessionType[first] release;
    //@   ensures SessionType[second] get;
    //@ also
    //@   requires SessionType[first] release;
    //@   requires SessionType[second] holding;
    //@   ensures receive[second] holding;
    //@   ensures send[first] release;
    //@   ensures send[second] release;
    //@   ensures send[first] get:release[self, first]:release[
       self, second];
    //@   ensures SessionType[first] release;
    //@   ensures SessionType[second] release;
    void holding(){
        sticks = (sticks+1)%2;

        if (sticks==0) {
            token t0 = first <- release():waitfor();
            token t1 = second <- release():waitfor();
            token t2 = first <- get(self):waitfor(t0, t1);
            return null;
        } else
            token t0 = second <- get(self):waitfor();
        return null;
    }
}

Init{
        //@ Actor c1;
        c1 = new Chopstick();
        //@ Actor c2;
        c2 = new Chopstick();
        //@ Actor p1;
```

```
        p1 = new Philosopher(c1, c2);
        //@ Actor p2;
        p2 = new Philosopher(c1, c2);

        token t1 = c1 <- setPhilosophers(p1, p2):waitfor();
        token t2 = c2 <- setPhilosophers(p1, p2):waitfor();
        //@ SessionType(p1, c1) send get; receive holding; send
           release; *;
        //@ SessionType(c1, p1) receive get; send holding; receive
            release; *;

        //@ SessionType(p1, c2) send get; receive holding; send
           release; *;
        //@ SessionType(c2, p1) receive get; send holding; receive
            release; *;

        //@ SessionType(p2, c1) send get; receive holding; send
           release; *;
        //@ SessionType(c1, p2) receive get; send holding; receive
            release; *;

        //@ SessionType(p2, c2) send get; receive holding; send
           release; *;
        //@ SessionType(c2, p2) receive get; send holding; receive
            release; *;

        //@ Message(p1, c1) get;
        c1 <- get(p1):waitfor(t1, t2);
        //@ Message(p2, c1) get;
        c1 <- get(p2):waitfor(t1, t2);
}
```

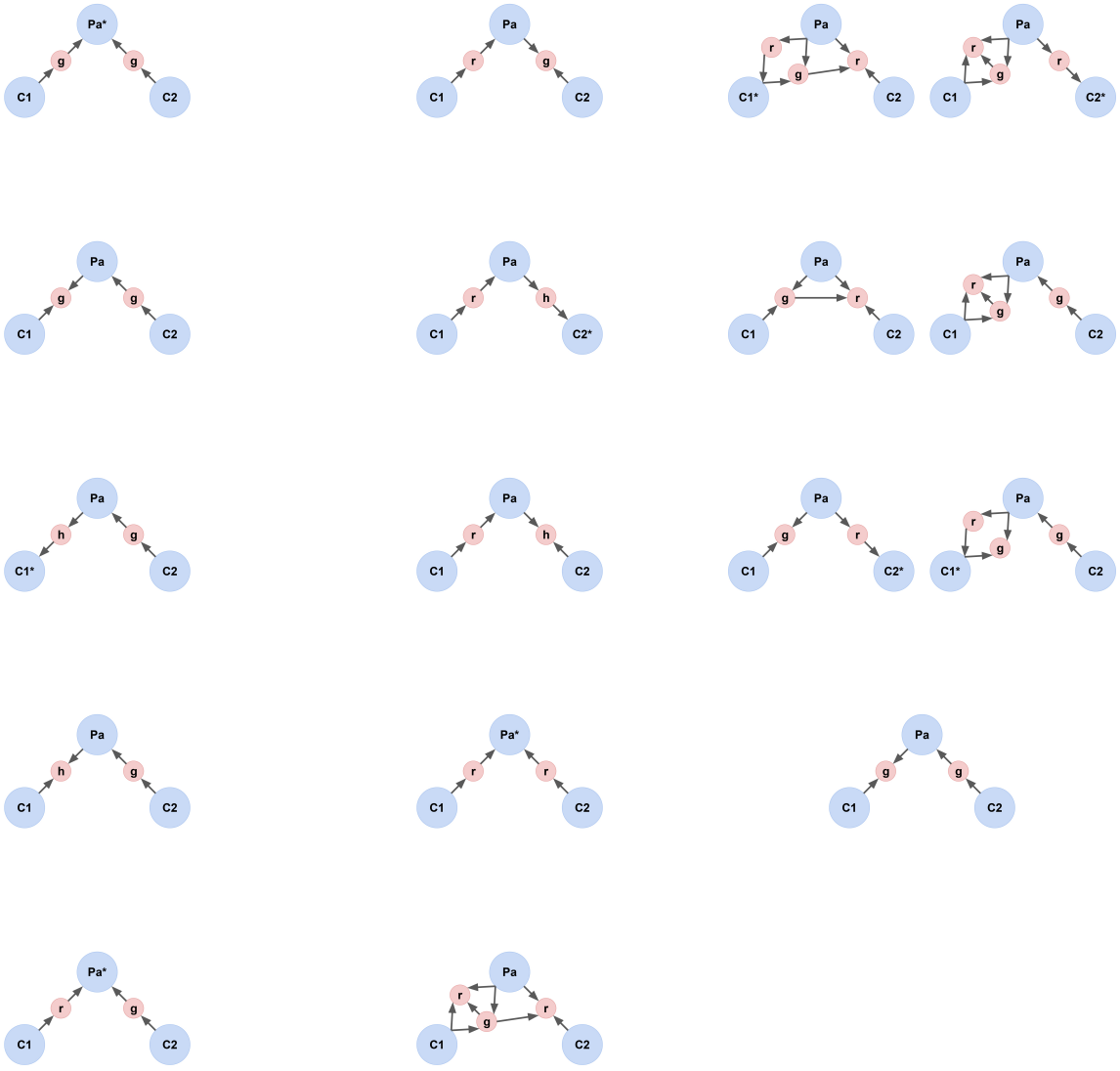


Figure 3.1: The chain of events of a single Philosopher in the DP example; top to bottom, left to right. The large blue nodes are actors, and include an asterisk while processing. The smaller red nodes are the session types between actors, with the label noting the type of message in the session type. Directed edges model a dependency; either an actor placing a message in transit, an actor waiting to place a message in transit, or an actor waiting to receive a message in transit. Note that due to multiple interleavings, the model splits before returning to unified state, and thereafter looping.

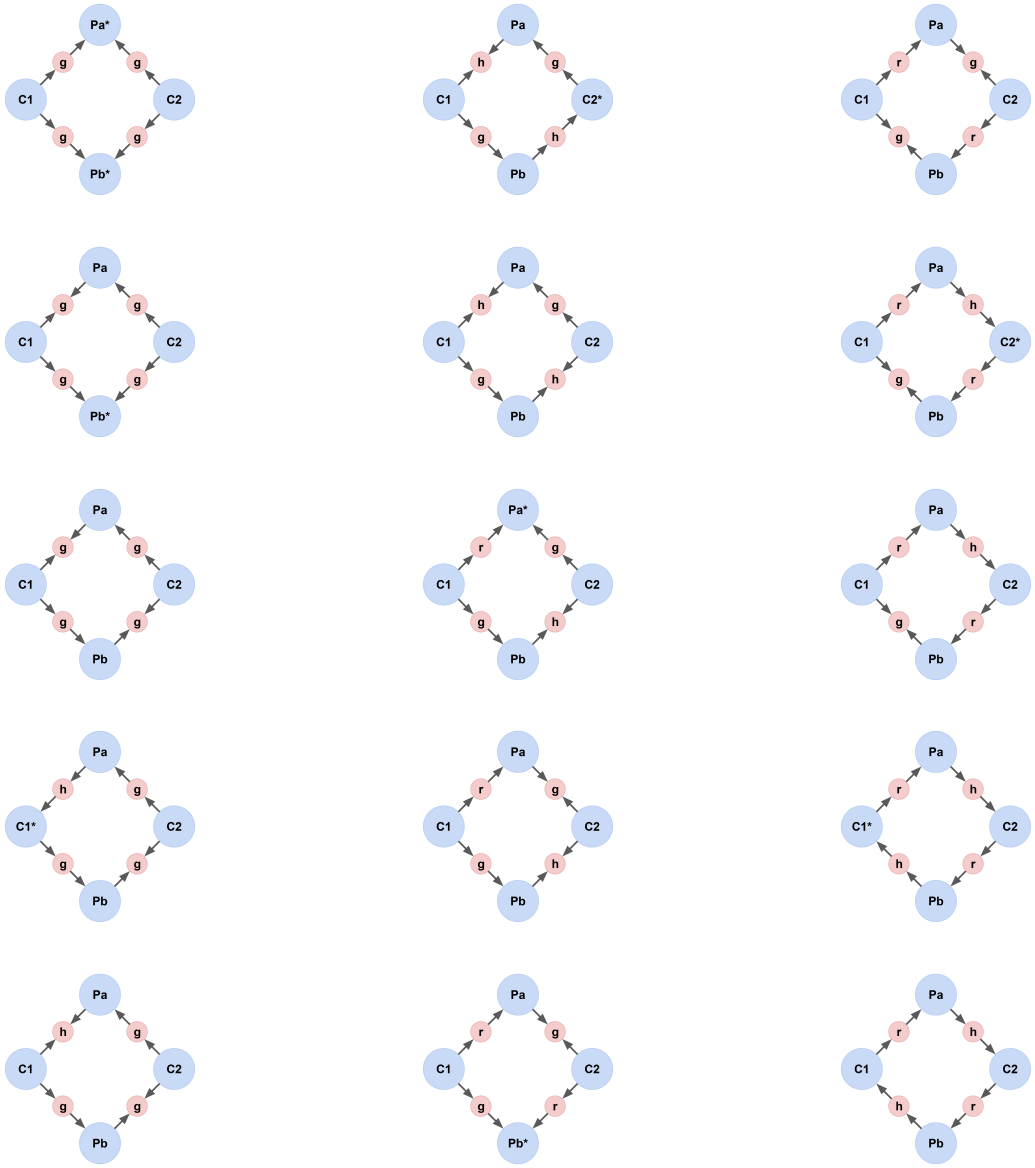


Figure 3.2: The chain of events leading to a possible deadlock scenario in the DP problem; top to bottom, left to right. The large blue nodes are actors, and include an asterisk while processing. The smaller red nodes are the session types between actors, with the label noting the type of message in the session type. Directed edges model a dependency; either an actor placing a message in transit, an actor waiting to place a message in transit, or an actor waiting to receive a message in transit. Note that the last frame contains a cyclic dependency.

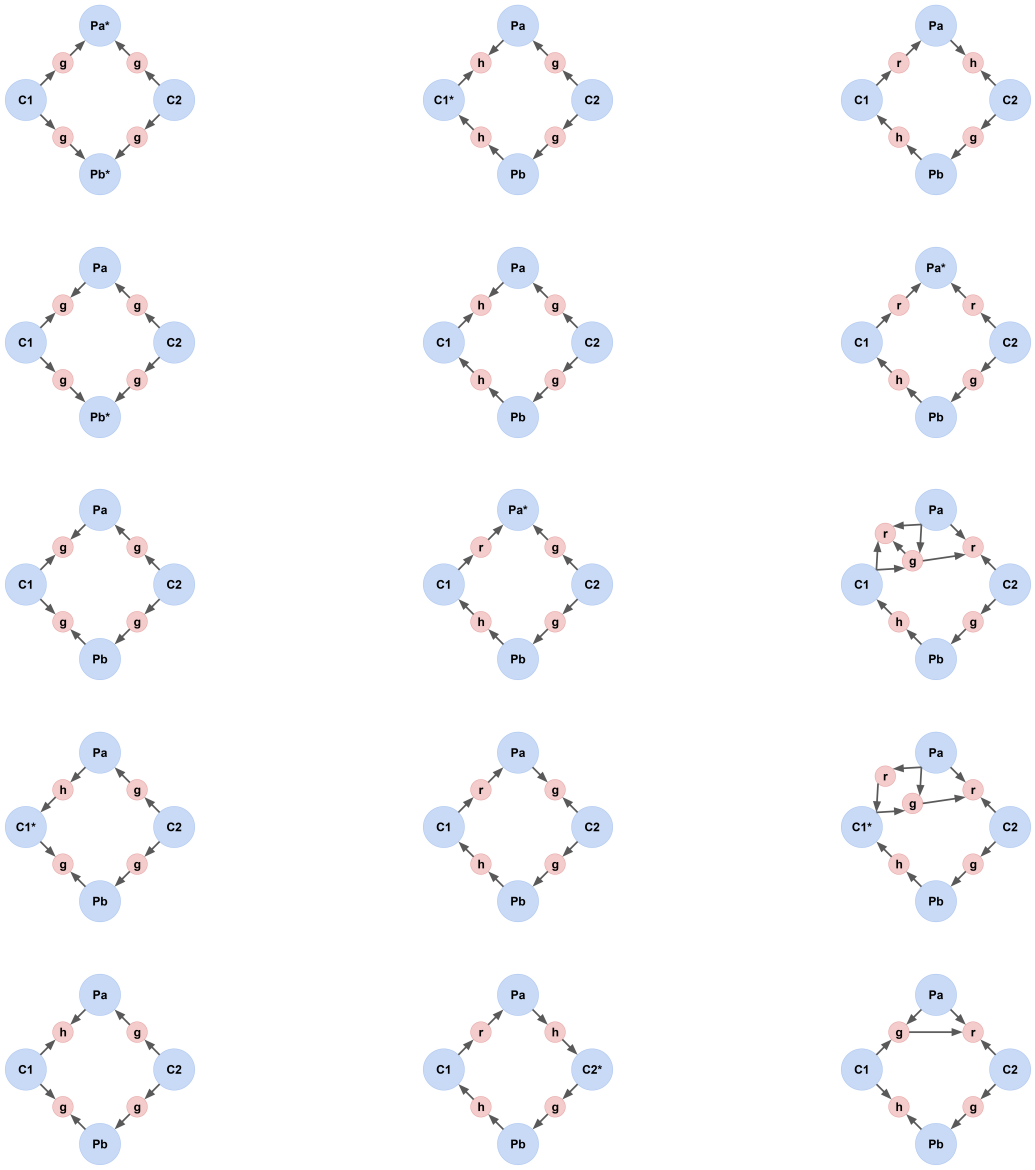


Figure 3.3: The chain of events of two Philosophers in the DP example; top to bottom, left to right. Second half of sequence on next page (Figure 3.4). The large blue nodes are actors, and include an asterisk while processing. The smaller red nodes are the session types between actors, with the label noting the type of message in the session type. Directed edges model a dependency; either an actor placing a message in transit, an actor waiting to place a message in transit, or an actor waiting to receive a message in transit.

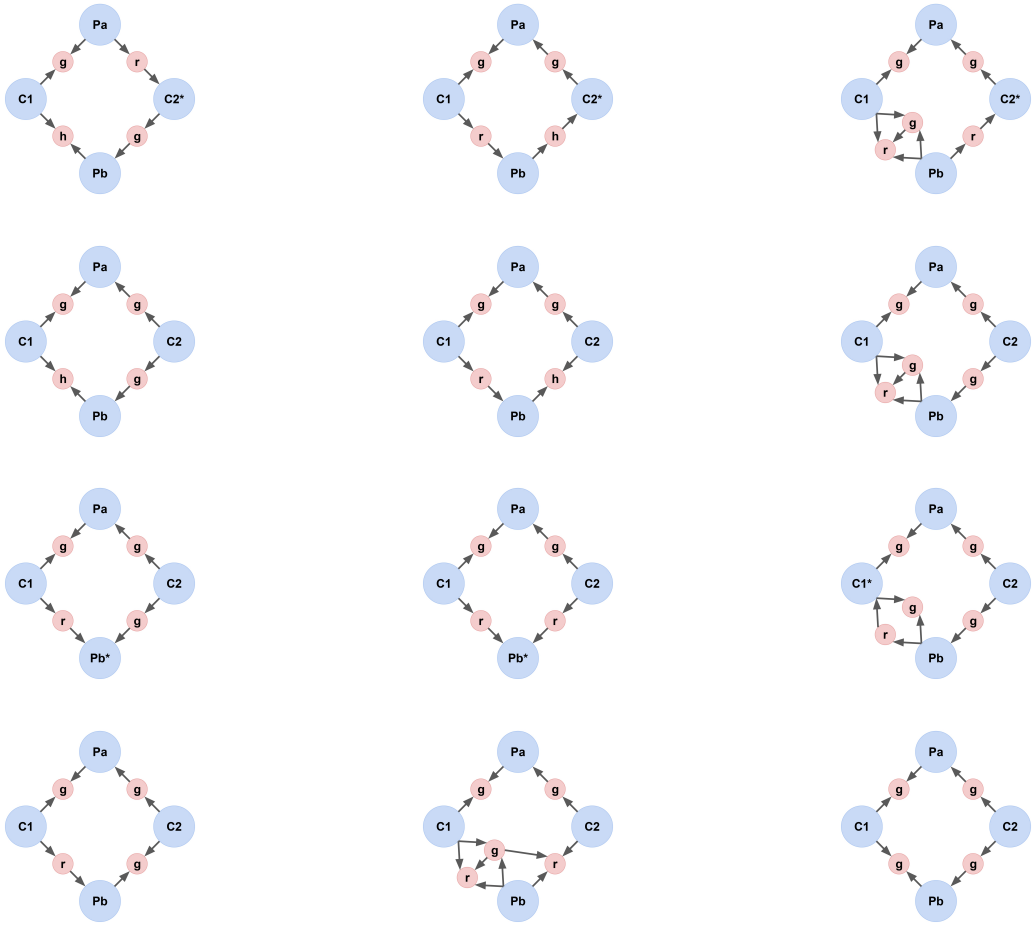


Figure 3.4: The chain of events of two Philosophers in the DP example; top to bottom, left to right. First half of sequence on previous page (Figure 3.3).The large blue nodes are actors, and include an asterisk while processing. The smaller red nodes are the session types between actors, with the label noting the type of message in the session type. Directed edges model a dependency; either an actor placing a message in transit, an actor waiting to place a message in transit, or an actor waiting to receive a message in transit.

4. LANGUAGE SYNTAX AND SEMANTICS

The first portion of the type checker program (Pseudocode in Section 4.3.1) is dedicated to making sure that the code itself matches the specifications, and confirming the Hoare triples found in each of the specifications. We make sure each annotation progresses exactly one session type by one stage (specifically, the stage corresponding to the message handler following it). Additionally, if the message handler has any branches, we guarantee there is

Table 4.1: Syntax for Session Typed FeatherWeight Salsa. Note that I and R symbolize initialization (including actor creation) and the annotations, respectively, and are formally defined in Table 4.3.

A	::=	{ true, false, null, ...}	Atoms
N	::=	{ 0, 1, 2, ...}	Natural numbers
X	::=	{ self, x, y, z, ...}	Variables
D	::=	{ a, b, c, ...}	Actor names
T	::=	{ t, t_0, t_1, ...}	Tokens
M	::=	{ m, n, o, ...}	Message names
B	::=	{ A, B, C, ...}	Behavior names
F	::=	{ +, *, ==, ...}	Primitive operations
V	::=	$A \mid N \mid X \mid D \mid T$	Values
E	::=		Expressions
		V	
	$\mid$	$F(E, ..., E)$	Function application
	$\mid$	$E; E$	Sequencing
	$\mid$	if (E) $\{E\}$ else E	Conditional execution
	$\mid$	token $T = E$ <-$M(E, ..., E)$	Message send
		:waitfor$(T, ..., T)$	
	$\mid$	return E	Set token; get ready
H	::=	$R\ B\ M(B\ X, ..., B\ X)$	Message handlers
		$\{X = E; ...; X = E; E; ...; E;\}$	
C	::=	$B(B\ X, ..., B\ X)$	Constructors
		$\{$this.$X = X; ...;$this.$X = X;\}$	
Be	::=	behavior B	Behaviors
		$\{B\ X; ...; B\ X; C\ H...H\}$	
P	::=	$Be\ Be\ ...\ Be\ I$	Programs

at least one annotation corresponding to each possible trace, such that all cases are covered.

The second portion (Pseudocode in Section 4.3.2) is dedicated to determining the types of messages each actor can process. For each method within an actor, it inspects each corresponding Hoare triple. It retains a list of the possible starting states from which a program can enter this method (i.e. at what step within a local session type have they progressed to), as well as the corresponding messages sent (including sender and recipient information) and the state the program ends on.

From there, the type checker generates a full set of all the possible actions the program can take, in no given order. It first collects data on the maximum number of actors and session types, then uses this data combined with the results of the previous step to create a chunk of data denoting a singular option. This chunk includes the starting states over the whole program before the message is consumed, the message consumed, the messages that would be sent following the control flow of this option, and the ending state that the program would be left in.

The third portion (Pseudocode in Section 4.3.3) takes this set of options, as well as the program's overall starting state and messages in transit, and recursively checks if it is possible for the program to consume an option. This occurs if the program's state and the option's starting state match (utilizing pattern matching if possible), as well as checking the received message has been sent and not yet received. If these conditions are met, the state of the session types are updated, as well as the list of messages in transit. Additionally, the fact that this state-message pair has been processed gets recorded, and any future copies of the same pair immediately return the same result. If there is ever a cyclical dependency within a strongly connected component without outgoing edges, the program alerts the user of the iteration order that caused this; otherwise, the program will eventually run out of states to check, thus guaranteeing the program is deadlock-free.

This type checker is guaranteed to terminate, despite having potentially exponential time. There are a finite number of traces (note that "trace" here refers to the static modeling of the communication topology, rather than the potentially infinite amount of dynamic traces which include runtime parameters), which means the tree detailing all walks of the program must also be finite when caching previous states. This does, however, grow exponentially, leading to a non-polynomial runtime in the worst case.

4.1 Session Typed FeatherWeight Salsa

We combine all these portions into one package (Pseudocode in Section 4.3.4), and run it on a modified featherweight version of the SALSA language, known as Session Typed FeatherWeight SALSA. Table 4.1 shows the syntax of this language, and Section 3.2 demonstrates how to convert and annotate the dining philosopher's problem into this framework. An important detail is that the only functions we need are primitive operations; all other functions and loops can be reduced to a message send. Note that since none of our primitive operations block the actor, this means all message handlers are guaranteed to eventually complete their processing and return the actor to the ready state.

Note that there are a few changes from the original implementation of FeatherWeight Salsa. Notably, all actor creation is done during the initialization of a program, which guarantees we work on a fixed set of actors and a fixed communication topology. The initialization is syntactic sugar for a unique actor which performs the same code, but is kept separate from the rest of the program in accordance with multiparty session types. If a program has an upper bound on the number of actors created and dynamic communications between them, this can be (inefficiently) modeled in our language by defining each possibility within the initialization and simulating the actor creations and communications. Additionally, we necessitate a set of requirements before each message handler, corresponding to the annotations for that message.

In terms of the semantics for this language, shown in Table 4.2, we largely borrow from FeatherWeight Salsa. In short, α is a map of actor names to actor states, μ is a multiset of messages, and β is the table of actor behaviors. Note that due to removing dynamic actor creation, the `(FS-Init)` rule is the only way new actors are made. The notation $\lfloor R \triangleright e \triangleleft \rfloor_a$ denotes that actor a is currently running, and is processing instruction e. The notations $\downarrow_t$ and $\uparrow_t$ denote a token being set and created, respectively. Lastly, the notation $\beta(B, m)$= B' m($\bar{B}$ $\bar{x}$) {$\bar{f} = \bar{e}$; $\bar{e}$';} $\forall v \in \bar{v}' \cup \bar{v}'' : v \notin T$ represents that behavior β has a message handler for m with the corresponding text.

4.2 Annotations

We expand on FeatherWeight Salsa [4], the syntax of which is shown in Table 4.3. We add an annotation before each message handler H, which we will label R. Each annotation includes the session types before the message handler, the type of message being received,

Table 4.2: Operational semantics for Session Typed FeatherWeight Salsa. The left column denotes the rule name, right column denotes parameters, and center column denotes the resulting reduction.

(FS-Seq) with (e $\rightarrow_s$ e')

$$\alpha,\ \lfloor R \triangleright e \triangleleft \rfloor_a \| \mu \| \beta \mapsto \alpha,\ \lfloor R \triangleright e' \triangleleft \rfloor_a \| \mu \| \beta$$

(FS-Send) with (fresh t'; t' $\in T$)

$$\alpha,\ \lfloor R \triangleright \text{token t = a' <- m}(\bar{v})\text{:waitfor}(\bar{t}) \triangleleft \rfloor_a \| \mu \| \beta \mapsto$$
$$\alpha,\ \lfloor R\{\text{t'/t}\} \triangleright \text{null} \triangleleft \rfloor_a \| \mu \uplus \{((\bar{t}) : a' \Leftarrow m(\bar{v}) \uparrow^{t'})\} \| \beta$$

(FS-Rtrn)

$$\alpha,\ \lfloor \text{B}(\bar{v})\text{:}R \triangleright \text{return v} \triangleleft \downarrow_t \rfloor_a \| \mu \| \beta \mapsto \alpha,\ \lfloor \text{B}(\bar{v}) \rfloor_a \| \mu\{v/t\} \| \beta$$

(FS-Recv) with ($\beta(B, m)$= B' m($\bar{B}$ $\bar{x}$){$\bar{f} = \bar{e}$; $\bar{e}$';} $\forall v \in \bar{v}' \cup \bar{v}'' : v \notin T$)

$$\alpha,\ \lfloor \text{B}(\bar{v}) \rfloor_a \| \{((\bar{v}'') : a \Leftarrow m(\bar{v}') \uparrow^t)\} \uplus \mu \| \beta \mapsto$$
$$\alpha,\ \lfloor (B(\bar{e}) : \bar{e}')\{(\bar{v}, \bar{v}', a)/(\bar{f}, \bar{x}, \text{self})\} \downarrow_t \rfloor_a \| \mu \| \beta$$

(FS-Init) with (fresh α; $\alpha \in X$ fresh μ; $\mu \in M$)

$$\emptyset \| \emptyset \| \beta \mapsto \alpha \| \mu \| \beta$$

any messages being sent, and finally the session types after the message handler.

In addition, we designate an Initializer actor I, which creates a finite set of actors, defines the session types between them, and sends the initial message(s) While we may be able to expand session types to dynamic actor creation in the future, doing so creates a multitude of issues beyond the scope of our current research, such as dynamic session type creation, recursive loop invariants, the usage of actor references as variables, and avoiding the Halting Problem.

Note: we assume the program makes progress whenever it recurses. If we did not have this assumption, we would have to account for cases such as livelock, where recursion occurs but no progress is made. Since this is outside the scope of our work, we exclude it from our analysis.

In terms of the semantics of our type system, defined in Table 4.4, γ represents the graph structure between actors, with $[\text{a} \triangleleft \text{m}]_b$ meaning actor a points towards message m with respect to its relationship with actor b. τ represents the graph structure for tokens, with m[a,b] $\triangleright$ m_1[c,d] representing that the message m from a to b relies on the token of message m_1 from c to d. Finally, σ represents the set of messages the system has yet to process being sent.

We break our type system into two main rules. The first, `(ST-Recv)`, details the

Table 4.3: Syntax of the annotations and initializer in Session Typed FeatherWeight Salsa. Note that this table reuses any definitions from Table 4.1.

Rst	::=	\\@ requires SessionType[X] M	Precondition Session Types
	|	Rst Rst	
En	::=	\\@ ensures receive[X] M	Received Message
	|	En Es	
Es	::=	\\@ ensures send[X] M	Sent Message(s)
Est	::=	\\@ ensures SessionType[X] M	Postcondition Session Types
	|	Est Est	
R	::=	Rst Rv E Est	Requirements
	|	Rst En Est	
	|	R \\@ also R	
Ac	::=	\\@ Actor X;	Actor creation
		X = new $B(E, ..., E)$;	
Im	::=	E <-$M(E, ..., E)$:waitfor$(T, ..., T)$	Initialization messages
St	::=	\\@ Session Type(X, X) Ms; *;	Session Type Declaration
Ms	::=	send M	Session Type Step
	|	receive M	
	|	Ms; Ms	
Am	::=	\\@ Message(X, X) M;	Activation messages
		E <-$M(E, ..., E)$:waitfor$(T, ..., T)$	
I	::=	Init{$Ac...Ac$ $Im...Im$ $St...St$ $Am...Am$}	Initialization Actor

Table 4.4: The typing rules of the session types, detailing the transitions possible from the graphical model. The left column denotes the rule name, right column denotes parameters, and center column denotes the resulting reduction.

(ST-Recv) with (m $\rightarrow$ m'; m[a,b]$\triangleright X \notin \tau$)

$$\gamma \cup [\text{a}\triangleright\text{m}]_b \cup [\text{b}\triangleright\text{m}]_a \parallel \tau \cup \overline{X \triangleright m[a,b]} \parallel \sigma \mapsto \gamma \cup [\text{a}\triangleleft\text{m'}]_b \cup [\text{b}\triangleright\text{m'}]_a \parallel \tau \parallel \sigma \cup \{\overline{m[a,c]}, \overline{\overline{m[a,c]} \triangleright m_1[d,e]}\}$$

(ST-Send) with ($\exists$ $\overline{m_1[c,d]}$)

$$\gamma \cup [\text{a}\triangleleft\text{m}]_b \parallel \tau \parallel \sigma \cup (\text{m[a,b]}, \overline{m[a,b] \triangleright m_1[c,d]}) \mapsto \gamma \cup [\text{a}\triangleright\text{m}]_b \parallel \tau \cup \overline{m[a,b] \triangleright m_1[c,d]} \parallel \sigma$$

Table 4.5: The static semantics, detailing the annotations' Hoare triples. The left column denotes the rule name, right column denotes parameters, and center column denotes the resulting reduction (pre and post conditions on top, code on bottom).

(HT-Recv) with ($m_1 \in M$)

$$\frac{\emptyset \mapsto \backslash\backslash @ \text{ ensures receive}[X]\ m_1}{B\ m_1(BX, ..., BX)\{X = E; ...; X = E; E; ...; E;\}}$$

(HT-Send) with ($m_1 \in M$)

$$\frac{\emptyset \mapsto \backslash\backslash @ \text{ ensures send}[X]\ m_1}{\text{token } T = E < -m_1(E, ..., E)}$$

(HT-If) with ($r_1, r_2 \in R$, $r_1 = R(e_1)$, $r_2 = R(e_2)$)

$$\frac{\text{pre}(r_1)\ \backslash\backslash @ \text{ also pre}(r_2) \mapsto \text{post}(r_1)\ \backslash\backslash @ \text{ also post}(r_2)}{\text{if } (E)\{e_1\}\text{else } e_2}$$

(HT-Types) with ($m_1 = m_2 \neq m_3$ or ($m_1 = m_3$ and $m_1 \mapsto m_2$))

$$\frac{\backslash\backslash @ \text{ requires SessionType}[X]\ m_1 \mapsto \backslash\backslash @ \text{ ensures SessionType}[X]\ m_2}{B\ m_3(BX, ..., BX)\{X = E; ...; X = E; E; ...; E;\}}$$

behavior when the system processes a message being received. It assumes that the subsequent message type of the system after m is m', and that message m has no tokens it relies on. The second, `(ST-Send)`, details the behavior of the system as it processes messages being sent. If it has tokens, this step only takes place after each of the corresponding messages has been sent, preserving an order for the tokens.

Additionally, we define the static semantics of our annotations. Since our annotations only detail the communication topology, only certain portions of code are relevant to the Hoare triples, namely the message handler name, and any message sends. `(HT-Recv)` ensures the message corresponding to the message handler is received, and `(HT-Send)` does the same with sent messages and the message send syntax. `(HT-If)` details that if a message handler includes branches (with $r_1 = R(e_1)$ denoting r_1 is the annotation for e_1), it must include at least one separate annotation for each branch, and if there are multiple annotations, there must be a branch corresponding to each one. Lastly, `(HT-Types)` ensures the session types outside the one being received do not get modified, and the one being received does get modified to its next stage.

4.3 Type Checker Pseudocode

Note: For the sake of brevity, any trivial edge cases (for example, empty program) are skipped within this pseudocode. This is not the case in actual implementations.

4.3.1 Annotation Checking

```
check_initialization(program){
    initialization = program.find("Init"); //Init Actor

    actors = [];
    for "\\@ Actor a" in initialization {
        assert("a = new B(...)" in initialization);
        actors.append(a);
    }

    s_types = [];
    //Actor a contains reference to b in its state
    contains_ref(a, b, string init){
        assert(behavior(a).contains_state(behavior(b)));
        return ("a = new B(..., b, ...)" or "a <- init_msg(..., b,
            ...)") in init;
    }
    for "\\@ Session Type (a,b) x; y; z; ...; *;" in
       intitialization {
        assert(contains_ref(a, b, initialization));
        for i in (x; y; z; ...;) {
            if (i == "send m")
                    {assert(behaviour(b).contains_handler(m))}
            if (i == "receive m")
                    {assert(behaviour(a).contains_handler(m))}
        }
        s_types.append((a,b) x; y; z; ...; *;);
    }
```

```
    for s_type in s_types {
        assert(s_types.contains(dual(s_type)));
    }

    msgs = [];
    for "\\@ Message (a,b) m" in initialization {
        assert("b <- m(...)" in initialization)
        assert(s_types[(a,b)][0] = "send m")
    }

    return (actors, s_types, msgs);
}

check_hoare_triples(program, actors, s_types){
    for behavior B {
    for message handler H {
    for annotation A{
        recv = A.find("\\@ ensures receive[a] msg_type(H)");
        sender = recv.get_sender();
        for "\\@ requires Session Type[a] m" in A {
            assert(s_types[self, a].contains(m));
            if (a == sender) {
                assert(m == msg_type(H));
                next_s_type = s_types[self, a][s_types[self, a].
                    loc[m]+1];
                assert("\\@ ensures Session Type[a] next_s_type"
                    in A);
            }
            else assert("\\@ ensures Session Type[a] m" in A);
        }
        sends = A.findall("\\@ ensures send[a] m:waitfor[m1[c,d],
            m2[e,f], ...]");
```

```
        sends.rewrite_as("token t = a<-m(...):waitfor(t1, t2, ...)
            ");
        assert(sends in trace T);
    }
    for trace T{
        sends = T.findall("token t = a<-m(...):waitfor(t1, t2,
            ...)");
        sends.rewrite_as("\\@ ensures send[a] m:waitfor[m1[c,d],
            m2[e,f], ...]");
        assert(sends in annotation A);
    }
    }
    }
}
```

4.3.2 Option Generation

```
generate_local_options(program, actors, s_types){
    options = []
    for behavior B {
        behavior_options = []
        for message handler H {
        for annotation A{
            recv = A.find("\\@ ensures receive[a] msg_type(H)");
            start = A.findall("\\@ requires Session Type[a] m");
            sends = A.findall("\\@ ensures send[a] m:waitfor[m1[c,
                d], m2[e,f], ...]");
            end = A.findall("\\@ ensures Session Type[a] m");
            behavior_options.append(Option(recv, start, sends, end
                ));
        }
        }
        options.append(behavior_options);
```

```
    }
    return options;
}

generate_global_options(options, actors, s_types){
    global_options = []
    for actor a {
        a_options = options[type(a)];
        st = s_types.get(a);
        for option in a_options{global_options.append(globalize(
           option, a, st));
        }
    }
    return global_options;
}
```

4.3.3 Model Checking

```
check_valid_option(option, state) {
    return state.s_types.contains(option.start) && state.msgs.
       contains(option.recv);
}

step(option, state) {
    state.s_types.update_progress(option.end);
    state.msgs.remove(option.recv);
    state.msgs.add(option.sends);
    return state;
}

check_model(options, state, prev_states){
    if (find_cycle(state)) return False;
    for option in options {
```

```
        if (check_valid_option(option, state) && ((option, state)
           not in prev_states)){
            prev_states.append((option, state));
            test = check_model(options, step(option, state),
               prev_states);
            if (!test) return False;
        }
    }
    return True;
}
```

4.3.4 Full Type Checker

```
type_check(program){
    actors, s_types, msgs = check_initialization(program);
    check_hoare_triples(program, actors, s_types);
    options = generate_global_options(generate_local_options(
       program, actors, s_types), actors, s_types);
    prev_states = [];
    check_model(options, State(actors, s_types, msgs), prev_states
       );
}
```

5. PROOFS

Definition 1 *(Deadlock)*: We define deadlock as a state in which a subset of actors are all waiting to receive messages only from other actors in the same subset, and the messages between them are not in transit.

Lemma 1 *(Cycles)*: Per our directed graphical representation, a cycle in a sink strongly connected component represents a deadlock.

Proof: Since each directed edge $[a \triangleleft m]_b$ marks the direction of a dependency, a cycle marks a cyclic dependency. If this is located in a sink component c, then the only outgoing edges of each node in c are to another node in c. Based on our definition of deadlock, this means all actors in c are waiting on messages from other actors in c. In addition, we can guarantee that the messages are not in transit, as all live messages are either sink nodes (as both the receiver and sender point to them immediately after `(ST-Send)`: $[a \triangleleft m]_b$ and $[b \triangleleft m]_a$ by structural induction on γ), or waiting on a token $m[a,b] \triangleright m_1[c,d]$. Since Lemma 2 shows these tokens will eventually get set as the program progresses, every message in transit will eventually become a sink node. Since we know there are no sink nodes in c, as well as no outgoing edges that could represent a token dependency, there must be no live messages in c, and therefore it must be in deadlock.

Corollary 1 *(Progress)*: At each successful step we look at, the program makes progress towards reaching its recursive state.

Proof: By the way each message requirement R is structured, it ensures the program receives exactly one message at that time step (corresponding to the message in the message handler). Since each session type has a finite number of steps to reach the Kleene Star and makes progress once it reaches that point, this will progress at least one session type forwards. Per our semantics, `(ST-Recv)` only occurs when message m is available to transition to m', which we have established as making progress. In doing so, it adds a finite number of messages to *sigma*, and since `(ST-Send)` decreases the size of σ by one, there will be a finite amount (or less) of `(ST-Send)` steps before the next `(ST-Recv)`, thus making progress.

Corollary 2 *(Preservation)*: At each step we look at, the structure of the graphical repre-

sentation of the program remains intact.

Proof: At every annotation, the only differences between the preconditions and post-conditions are an increment of a single session type, a single message gets received, and a variable number of messages are sent. Since our program updates the graph with the received message and the new session type, the graph itself remains intact. We also add the sent messages to the list of live messages, and mark their corresponding session types as having been sent. Per our semantics, the number of edges in γ remains constant at each step. In `(ST-Send)`, it only changes the direction of a single edge, and in `(ST-Recv)`, it replaces one pair of edges corresponding to m ($[a \triangleright m]_b \cup [b \triangleright m]_a$) with one pair of edges corresponding to m' ($[a \triangleleft m']_b \cup [b \triangleright m']_a$).

Lemma 2 *(Tokens)*: The unique nature of tokens prevents them from causing any deadlock.

Proof: In Session Typed FeatherWeight Salsa, each token is created 'fresh', meaning it has not been used anywhere up until that point. Since the only way to create a token is to attach it to a message, the same applies to messages that are waiting on tokens. By the Well-Ordering Principle, there must be an oldest live message m that was created before all other live messages. Since any tokens that m may have to wait for would be attached to older messages, those messages must not be live. Since message handlers cannot block while receiving a message, we can say that all of m's tokens will be set. This means that m can be received by its designated actor, and that progress can be made.

Lemma 3 *(Traces)*: The type checker looks at every possible trace of the program to determine if any lead to deadlock.

Proof: At every time step t of the type checker, we look at the list of live messages and choose one to analyze for deadlock. This essentially models what occurs if you perform a single `(ST-Recv)` step, followed by `(ST-Send)` until σ is empty. Doing so guarantees the largest amount of options for `(ST-Recv)`, meaning we cover all results (it moves tokens from σ to τ, and transitions more elements of γ to have edges pointing to the message, which `(ST-Recv)` needs two of, instead of pointing away). If no deadlock is found in any trace including time step (t+1), we backtrack to t and try every other message that can be received. This performs a depth-first search on the ordering of received messages (caching is used to prune the search in the event of duplicates). Since each trace is given by the possible

interleavings of message handlers, this will cover every possible trace.

Corollary 3 *(Receives)*: Each annotation progresses a single session type by one stage; namely, by receiving the message correlated to the message handler.

Proof: Due to the nature of Session Typed FeatherWeight Salsa, each actor only receives one message m at a time before becoming ready to receive any other message. Since the session types represent a series of sent and received messages, and each message can only correspond to one session type, receiving m must progress the session type between its sender and recipient by one stage exactly, and no other session type. This is modeled very succinctly in our semantics by definition of `(ST-Recv)`.

Corollary 4 *(Sends)*: Each annotation notes all possible messages sent depending on the branching of the message handler.

Proof: In our type checker, we analyze all possible traces of an individual message handler, depending on the branches it takes within if-statements. Since we make sure there is at least one annotation for each trace, we guarantee the user has marked the postconditions for each trace, since sent messages only matter for the postconditions. Per Definition 2 the difference in session types changes by one stage, and by definition the message received is that of the message handler. Therefore, the only remaining annotation that changes depending on the trace of a message handler is the sent messages. However, this is encapsulated in the definition of `(ST-Recv)`: $\sigma \cup \{\overline{m[a,c]}, \overline{\overline{m[a,c]} \triangleright m_1[d,e]}\}$

Lemma 4 *(Cycle Existence)*: If a program can enter deadlock, there is a trace in which the graphical model forms a cycle.

Proof: Since our model encapsulates every interaction (sends, receives, branches, etc.), and deadlock can only be formed by a cyclic dependency made up of these interactions, a trace corresponding to deadlock can only be represented by a cycle in the graph.

Proof 1 *(Deadlock)*: If a program can enter deadlock, the type checker finds a cycle in the model and vice versa.

Proof: Per Lemma 1 and Lemma 4, a cycle indicates deadlock, and the presence of deadlock indicates a cycle. Since our type checker analyzes every possible trace, it must

encounter the trace including a cycle at some point if the program can deadlock. If the program cannot deadlock, then no trace will include a cycle, and the type checker will terminate (due to caching) and confirm the program is deadlock-free.

6. CONCLUSION

We have shown that it is possible to represent SALSA programs as a sequence of session type interactions, and have proved that representing them as such can guarantee certain programs never deadlock, as well as show the sequence of actions that can lead to deadlock otherwise. The full code implementation of our work is located at:

https://github.com/wilkersona/deadlock_freedom

6.1 Future Work

This work focuses specifically on deadlock, but has the potential to be expanded towards situations such as livelock. Since in our work we assume an actor looping through its session types makes progress, adding notation for situations in which such loops make progress or not could potentially flag livelock as well. One idea of how to accomplish this would be to mark goal states in the annotations, and translate this information into the tree of communication traces.

We also do not explore spaces with a dynamic number of actors. This would necessitate the ability to create session types dynamically, which significantly increases the complexity of verifying programs on a global scale. This may be possible by defining the roles each session type is defined between, or having a subset of safe actors. However, being able to verify such properties at compile time is highly related to the Halting Problem, which is known to be impossible.

Currently we do not automatically generate the annotations nor verify that they encapsulate the corresponding program. Creating the session types and proving correctness would be incredibly helpful for larger, more complex programs.

REFERENCES

[1] K. Honda, V. T. Vasconcelos, and M. Kubo, "Language primitives and type discipline for structured communication-based programming," in *Lecture Notes in Computer Science*, vol. 1381, Heidelberg, Germany: Springer Verlag, 1998, pp. 122–138. DOI: 10.1007/bfb0053567.

[2] M. Carbone and S. Debois, "A graphical approach to progress for structured communication in web services," in *Proc. Interact, Concurrency Experience (ICE)*, vol. 38, Oct. 2010, pp. 13–27. DOI: 10.4204/eptcs.38.4.

[3] J. Armstrong, "The development of erlang," in *Proc. of the Second ACM SIGPLAN Int. Conf. on Functional Program. (ICFP '97)*, Amsterdam, The Netherlands, 1997, pp. 196–203. DOI: 10.1145/258948.258967.

[4] C. A. Varela, *Programming Distributed Computing Systems: A Foundational Approach*. Cambridge, MA, USA: MIT Press, 2013.

[5] D. Mostrous and V. T. Vasconcelos, "Session typing for a featherweight erlang," in *Proc. of the 13th Int. Conf. on Coordination Models and Lang. (COORDINATION '11)*, Reykjavik, Iceland, 2011, pp. 95–109.

[6] R. Neykova and N. Yoshida, "Multiparty session actors," in *Program. Lang. Approaches to Concurrency and Commun. cEntric Softw. (PLACES)*, vol. 155, Jun. 2014, pp. 32–37. DOI: 10.4204/eptcs.155.5.

[7] J. C. Reynolds, "Separation logic: A logic for shared mutable data structures," in *Proc. of the 17th Annual IEEE Symp. on Log. in Comput. Sci. (LICS '02)*, 2002, pp. 55–74.

[8] J. Jacobs, S. Balzer, and R. Krebbers, "Connectivity graphs: A method for proving deadlock freedom based on separation logic," *Proc. ACM Program. Lang.*, vol. 6, no. POPL, pp. 1–33, Jan. 2022. DOI: 10.1145/3498662.

[9] M. Charalambides, P. Dinges, and G. Agha, "Parameterized, concurrent session types for asynchronous multi-actor interactions," *Sci. Comput. Program*, vol. 115-116, pp. 100–126, Jan. 2016. DOI: https://doi.org/10.1016/j.scico.2015.10.006.

[10] E. W. Dijkstra, "Hierarchical ordering of sequential processes," *Acta Informatica*, vol. 1, no. 2, pp. 115–138, Jun. 1971. DOI: 10.1007/BF00289519.

[11] V. R. Pratt, "Semantical considerations on floyd-hoare logic," in *17th Annual Symp. on Found. of Comput. Sci. (SFCS)*, Oct. 1976, pp. 109–121. DOI: 10.1109/SFCS.1976.27.

APPENDIX A
EXAMPLE PROGRAM: THREAD RING

```
behavior ThreadRing {
        ThreadRing recipient;
        ThreadRing(){}

        void setRecipient(ThreadRing new_recipient){
                recipient = new_recipient;
                return null;
        }

        //@ ThreadRing loop;
        //@ requires SessionType [0] send loop;
        //@ requires SessionType [1] receive loop;
        //@ ensures receive[1] loop;
        //@ ensures send[0] loop;
        //@ requires SessionType [0] send loop;
        //@ requires SessionType [1] receive loop;
        //@     also
        //@ requires SessionType [0] receive loop;
        //@ requires SessionType [1] send loop;
        //@ ensures receive[0] loop;
        //@ ensures send[1] loop;
        //@ requires SessionType [0] receive loop;
        //@ requires SessionType [1] send loop;
        void loop(int n){
                token t0 = recipient <- loop(n+1):waitfor();
                return null;
        }
}
```

```
Init {
    //@ Actor tr0;
    tr0 = new ThreadRing();
    //@ Actor tr1;
    tr1 = new ThreadRing();
    //@ Actor tr2;
    tr2 = new ThreadRing();
    //@ Actor tr3;
    tr3 = new ThreadRing();
    //@ Actor tr4;
    tr4 = new ThreadRing();
    //@ Actor tr5;
    tr5 = new ThreadRing();
    //@ Actor tr6;
    tr6 = new ThreadRing();
    //@ Actor tr7;
    tr7 = new ThreadRing();

    //@ SessionType(tr0, tr1) send loop;*;
    token t0 = tr0 <- setRecipient(tr1):waitfor();
    //@ SessionType(tr1, tr2) send loop;*;
    token t1 = tr1 <- setRecipient(tr2):waitfor();
    //@ SessionType(tr2, tr3) send loop;*;
    token t2 = tr2 <- setRecipient(tr3):waitfor();
    //@ SessionType(tr3, tr4) send loop;*;
    token t3 = tr3 <- setRecipient(tr4):waitfor();
    //@ SessionType(tr4, tr5) send loop;*;
    token t4 = tr4 <- setRecipient(tr5):waitfor();
    //@ SessionType(tr5, tr6) send loop;*;
    token t5 = tr5 <- setRecipient(tr6):waitfor();
    //@ SessionType(tr6, tr7) send loop;*;
    token t6 = tr6 <- setRecipient(tr7):waitfor();
```

```
    //@ SessionType(tr7, tr0) send loop;*;
    token t7 = tr7 <- setRecipient(tr0):waitfor();

    //@ Message(tr7, tr0) loop;
    token t8 = tr0 <- loop(0):waitfor(t0, t1, t2, t3, t4, t5, t6,
       t7);
}
```