Implementing a Message-Passing Interpretation of the Semi-Axiomatic Sequent Calculus (Sax)⋆

Adrian Francalanza[1] , Gerard Tabone[1] , and Frank Pfenning[2]

[1] University of Malta, Msida, Malta
{adrian.francalanza,gerard.tabone}@um.edu.mt
[2] Carnegie Mellon University, Pittsburgh, PA, USA
fp@cs.cmu.edu

Abstract. We present language implementation based on a formulation of sessions types for message-passing programs in terms of an adjoint intuitionistic logic. This logical formulation can naturally describe asynchronous concurrency and can handle linear, affine, multicast and replicated types. This allows the resulting language to express a variety of common programming idioms such as service replicable, broadcast communication and message cancellations within the same programming, while still guaranteeing safety. Our tool consists of a type-checker and an interpreter. It is implemented in the Go language, leveraging its concurrency features in order to investigate the implementability of the operational interpretation proposed by the adjoint logic formulation. We assess the performance of our concurrent interpreter and show that it scales adequately to the number of concurrent processes executed.

Keywords: behavioural types · concurrency · language implementation

1 Introduction

The Semi-Axiomatic Sequent Calculus (Sax) [10, 28] is a logical framework that blends features of the sequent calculus with axiomatic presentations of intuitionistic logic, replacing non-invertible rules by corresponding axioms. The framework has been shown to elegantly handle a variety of substructural modalities such as linear, affine, multicast and replication within one uniform formalism. Sax also induces a natural operational interpretation in terms of active and passive parallel processes that interact *asynchronously*. Numerous variants of such an interpretation have been studied for a variety of computational models, ranging from shared memory concurrency [10], futures [32], and message-passing concurrency [31]. Every proposed interpretation is shown to observe standard

---

requirement such as progress and preservation, in addition to other operational properties such as confluence and deadlock-freedom.

This paper focusses on the message-passing operational interpretation of SAX, and investigates the *implementability* of the proposed operational model. Concretely we build a type-checker that automates the verification of process terms modelling session type specifications [20], according to the substructural type system developed in the aforementioned paper. This gives us a language for expressing *safe* message-passing concurrency that departs from the strict linearity constraints: we can flexibly express a variety of common programming idioms such as replicable services broadcast communication and message cancellations. We also build an interpreter that executes typed programs according to the concurrent reduction semantics given in [31]. Our interpreter targets the Go programming language which natively supports message-based concurrency via channels and goroutines [11]. More precisely, our implementation supports two execution options (using unbuffered and buffered channels respectively) which allows us to better asses the implementability of the asynchronous semantics proposed.

Structure and Contribution. After reviewing the static and dyamic semantics of our language (sec. 2) we outline the design decisions leading to our type-checker (sec. 3). This is followed by a discussion on the implementation of the interpreter (sec. 4). We finally evaluate our implementation in sec. 5. The accompanying tool, called GRITS, is available at https://github.com/gertab/Grits (archived [36]).

2   SAX for Message-Passing Concurrency

The asynchronous message-passing language proposed by Pruiksma et. al. [31] centers around the intuitionistic judgement eq. (1) below. It defines an interface specification for process P, asserting that it *provides* the behaviour described by the proposition B on the channel denoted by variable y, assuming that some $Q_1,...,Q_k$ processes (to which it is *client*) each provide a behaviour described by A_i on channel x_i respectively.

$$x_1 : A_1, ..., \ x_k : A_k \vdash P :: (y : B) \tag{1}$$

At runtime, the variables $x_1,...,x_k,y$ in eq. (1) are instantiated to dynamically-allocated channel names $a_1,...,a_k,b$, resulting in the process arrangement of fig. 1. The behaviour described by the channel propositions in eq. (1) have a dual interpretation. Process R in fig. 1 is a *client* on channel b whereas processes $Q_1,...,Q_k$ *provide* on channels $a_1,...,a_k$ to which process P is a client. In the sequel, we let identifiers $u,v,w,...$ range over both names, $a,b,c,d,...$ and variables $x,y,z,...$.

Types. Channel propositions are *session types*, expressed as linear logic connectives [4] indexed by a specific *mode* of truth, m (which is elided when implicit).
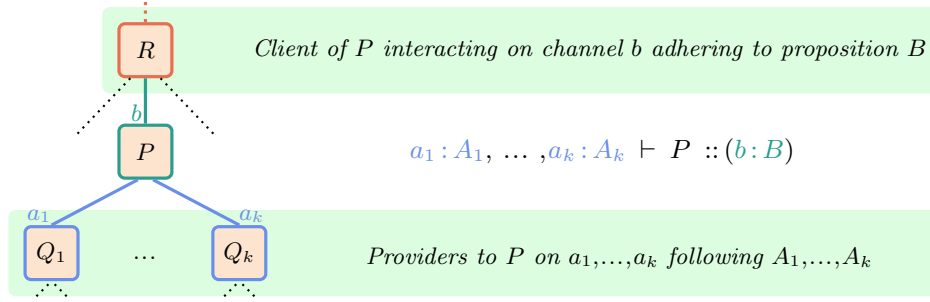
Fig. 1: Hierarchical structure of processes, from P_2's perspective

$$A^m, B^m \in \text{Type} ::= A^m \otimes B^m \quad | \; A^m \multimap B^m \quad | \oplus \{l : A_l^m\}_{l \in L} \quad | \; \&\{l : A_l^m\}_{l \in L}$$
$$| \; \mathbf{1}^m \qquad\quad | \; t^m \qquad\quad\; | \uparrow_m^n A^m \qquad\;\; | \downarrow_n^m A^m$$
$$m, n, o \in \text{Mode} ::= \mathsf{l} \;\; \textit{(linear)} \quad | \; \mathsf{a} \;\; \textit{(affine)} \quad | \; \mathsf{m} \;\; \textit{(multicast)} \quad | \; \mathsf{r} \;\; \textit{(replicable)}$$

A tensor, $A^m \otimes B^m$, represents the *sending* of channel of type A^m along with the continuation channel of type B^m. Implication, $A^m \multimap B^m$, is its dual, representing the *receipt* of a channel A^m with the continuation of type B^m. A labelled n-ary sum-type, $\oplus \{l : A_l^m\}_{l \in L}$, represents the *internal choice* from a range of labels $l \in L$ transferred along with the continuation channel having type A_l, whereas the *external choice*, $\&\{l : A_l^m\}_{l \in L}$, receives such a label $l \in L$ and continuation channel at type A_l. The unit type, $\mathbf{1}^m$, and the recursion variable, t^m, are standard; a collection of *contractive* [16] equi-recursive type definitions, $t^m = A^m$, is assumed.

All logical connectives combine types at the *same mode*, except upshifts, $\uparrow_m^n A^m$ and downshifts, $\downarrow_n^m A^m$. Four modes are considered, representing the possible combinations of contraction and weakening. This induces a mode pre-order, $m \succeq n$ (see axioms below), whenever m has *more* substructural properties than n.

$$\mathsf{r} \succeq \mathsf{m} \qquad\qquad \mathsf{r} \succeq \mathsf{a} \qquad\qquad \mathsf{m} \succeq \mathsf{l} \qquad\qquad \mathsf{a} \succeq \mathsf{l}$$

Whereas linear and replicable are bottom and top elements, affine allows only weakening whereas multicast only permits contraction. Shifts, are subject to the following mode ordering constraints: $\uparrow_n^m A^n$ and $\downarrow_n^m A^m$ require that $m \succeq n$.

Syntax. Our process syntax follows closely the one in [31]. The constructs inducing interaction (*i.e.,* send, select, close and cast) are all *asynchronous* (*i.e.,* without any continuation process). Forwarding, spawning, splitting and dropping are structural constructs (the latter two are implicit in [31]). The processes assume a collection of named process definitions $p(\overline{x}) = P$ which are invoked using $p(\overline{u})$ (where p ranges over process names).

Type	Cont.	Process	Cont.	Description
$c : A \otimes B$	$-$	send $c\langle a,d\rangle$	$-$	provider sends $a : A$, $d : B$ on c
		$\langle x,y\rangle \leftarrow$ recv $c;P$	$P[a,d/x,y]$	client receives $a : A$, $d : B$ on c
$c : A \multimap B$	$d : B$	$\langle x,y\rangle \leftarrow$ recv $c;P$	$P[a,d/x,y]$	provider receives $a : A$, $d : B$ on c
		send $c\langle a,d\rangle$	$-$	client sends $a : A$, d on c
$c : \oplus\{l : A_l\}_{l \in L}$	$-$	$c.k\langle d\rangle$	$-$	provider selects $k \in L$ with $d{:}A_k$ on c
		case $c\,(l\langle y\rangle \Rightarrow P_l)_{l \in L}$	$P_k[d/y]$	client branches to $k \in L$ with $d{:}A_k$ on c
$c : \&\{l : A_l\}_{l \in L}$	$d : A_k$	case $c\,(l\langle y\rangle \Rightarrow P_l)_{l \in L}$	$P_k[d/y]$	provider branches to $k \in L$ with $d{:}A_k$ on c
		$c.k\langle d\rangle$	$-$	client selects $k \in L$ with $d{:}A_k$ on c
$c : \downarrow_n^m A^m$	$-$	cast $c\langle d\rangle$	$-$	provider upshifts to $d : A^m$ on c
		$y \leftarrow$ shift $c;P$	$P[d/y]$	client upshifts to $d : A^m$ on c
$c : \uparrow_n^m A^n$	$-$	$y \leftarrow$ shift $c;P$	$P[d/y]$	provider downcasts to $d : A^n$ on c
		cast $c\langle d\rangle$	$-$	client downcasts to $d : A^n$ on c
$c : \mathbf{1}$	$-$	close c	$-$	provider terminates on c
		wait $c;P$	P	client receives termination on c

Tbl. 1: Session types mapped to processes

$$
\begin{aligned}
P,Q \in \text{PROC} ::= {}& \text{send } u\langle v,w\rangle && \textit{(send)} && \mid \langle x,y\rangle \leftarrow \text{recv } u;P && \textit{(receive)} \\
& \mid u.l\langle v\rangle && \textit{(select)} && \mid \text{case } u\,(l\langle y\rangle \Rightarrow P_l)_{l \in L} && \textit{(branch)} \\
& \mid \text{close } u && \textit{(close)} && \mid \text{wait } u;P && \textit{(wait)} \\
& \mid \text{cast } u\langle v\rangle && \textit{(cast)} && \mid x \leftarrow \text{shift } u;P && \textit{(shift)} \\
& \mid \text{fwd } u\ v && \textit{(forward)} && \mid x \leftarrow \text{new } P;Q && \textit{(spawn)} \\
& \mid \langle x,y\rangle \leftarrow \text{split } u;P && \textit{(split)} && \mid \text{drop } u;P && \textit{(drop)} \\
& \mid p(\overline{u}) && \textit{(call)}
\end{aligned}
$$

Three *synchronous* processes (*i.e.,* receive, branch and shift) together with spawn and split, bind variables in the respective continuation processes (*e.g.* receive binds variables x and y in P). To facilitate the mechanisation of typechecking, we follow Sano et al. [33], Crary [7] and require every bound variable to be used linearly (exactly once) in the binding scope. Process spawning, *i.e.,* $x \leftarrow$ new $P;Q$, partially breaks this syntactic constraint and allows x to be used linearly in P and in Q to generate a more standard formulation of the CUT rule (see below).

Type System. Based on eq. (1), the SAX type system for static and runtime terms[3] takes the form eq. (2) and observes *mode independence*: $\forall i \in 1..k \ . \ m_i \geq n$.

$$u_1 : A_1^{m_1},\ldots,\ u_k : A_k^{m_k} \vdash P :: (v : B^n) \tag{2}$$

---

[3] Static terms are closed (*i.e.,* no free variables) and do not contain any names.

Type environments, Γ, range over sequences of antecedents, $u_i : A_i^{m_i}$, subject to exchange. The structural rules are fairly standard, augmented with mode considerations. *E.g.* Cut (below) embodies computation via the interaction between the spawning (client) process, Q, and the spawnee (provider), P, via the dynamically allocated channel x. To preserve mode independence, all antecedent modes typing P should be ordered w.r.t. the mode of channel x, $\Gamma \succeq m$, which should also preserve independence w.r.t. the channel that Q provides on, $m \succeq n$. Rules Drp and Spl explicitly link weakening and contraction to structural terms.

$$\frac{}{u : A^m \vdash \mathsf{fwd}\ w\ u :: (w : A^m)}\ \text{Id} \qquad \frac{\Gamma \succeq m \succeq n \quad \Gamma \vdash P :: (x : A^m) \quad \Gamma', x : A^m \vdash Q :: (u : B^n)}{\Gamma, \Gamma' \vdash x \leftarrow \mathsf{new}\ P; Q :: (u : B^n)}\ \text{Cut}$$

$$\frac{m \in \{\mathsf{a},\mathsf{r}\} \quad \Gamma \vdash P :: (w : B^n)}{\Gamma, u : A^m \vdash \mathsf{drop}\ u; P :: (w : B^n)}\ \text{Drp} \qquad \frac{m \in \{\mathsf{m},\mathsf{r}\} \quad \Gamma, x : A^m, y : A^m \vdash P :: (w : B^n)}{\Gamma, u : A^m \vdash \langle x, y \rangle \leftarrow \mathsf{split}\ u; P :: (w : B^n)}\ \text{Spl}$$

There is a left and right rule for every logical connective. Crucially, in Sax, right rules of positive connectives and left rules of negative connectives are axioms, capturing the asynchronous nature of the constructs inducing interaction. We detail the typing rules for the tensor and implication connectives below and outline the relationship for the remaining connectives in tbl. 1.

$$\frac{}{u : A^m, v : B^m \vdash \mathsf{send}\ w\langle u, v \rangle :: (w : A^m \otimes B^m)}\ \otimes \text{R} \qquad \frac{\Gamma, x : A^m, y : B^m \vdash P :: (w : C^n)}{\Gamma, u : A^m \otimes B^m \vdash \langle x, y \rangle \leftarrow \mathsf{recv}\ u; P :: (w : C^n)}\ \otimes \text{L}$$

$$\frac{\Gamma, x : A^m \vdash P :: (y : B^m)}{\Gamma \vdash \langle x, y \rangle \leftarrow \mathsf{recv}\ w; P :: (w : A^m \multimap B^m)}\ \multimap \text{R} \qquad \frac{}{u : A^m, w : A^m \multimap B^m \vdash \mathsf{send}\ w\langle u, v \rangle :: (v : B^m)}\ \multimap \text{L}$$

For completeness, we list below the remaining rules used by the type system. The environment Σ is fixed and left implicit; it stores the typing information for all process definitions. For a comprehensive discussion of these rules, see [31].

$$\frac{\Sigma(p) = \overline{y : A^m} \vdash P :: (x : B^n)}{\overline{u : A^m} \vdash p(w, \overline{u}) :: (w : B^n)}\ \text{Call} \qquad \frac{}{\cdot \vdash \mathsf{close}\ w :: (w : \mathbf{1}^m)}\ \text{1R} \qquad \frac{\Gamma \vdash P :: (w : A^n)}{\Gamma, u : \mathbf{1}^m \vdash \mathsf{wait}\ u; P :: (w : A^n)}\ \text{1L}$$

$$\frac{l \in L}{u : A_l^m \vdash w.l\langle u \rangle :: (w : \oplus\{l : A_l\}_{l \in L}^m)}\ \oplus \text{R} \qquad \frac{\Gamma, y_l : A_l^m \vdash P_l :: (w : B^n) \quad \text{for each } l \in L}{\Gamma, u : \oplus\{l : A_l\}_{l \in L}^m \vdash \mathsf{case}\ u\ (l\langle y_l \rangle \Rightarrow P_l)_{l \in L} :: (w : B^n)}\ \oplus \text{L}$$

$$\frac{\Gamma \vdash P_l :: (y_l : A_l^m) \quad \text{for each } l \in L}{\Gamma \vdash \mathsf{case}\ w\ (l\langle y_l \rangle \Rightarrow P_l)_{l \in L} :: (w : \&\{l : A_l\}_{l \in L}^m)}\ \&\text{R} \qquad \frac{l \in L}{u : \&\{l : A_l\}_{l \in L}^m \vdash u.l\langle w \rangle :: (w : A_l^m)}\ \&\text{L}$$

$$\frac{\Gamma \vdash P :: (y : A^n)}{\Gamma \vdash y \leftarrow \mathsf{shift}\ w; P :: (w : \uparrow_n^m A^n)}\ \uparrow \text{R} \qquad \frac{}{u : \uparrow_n^m A^n \vdash \mathsf{cast}\ u\langle w \rangle :: (w : A^n)}\ \uparrow \text{L}$$

$$\frac{}{u : A^m \vdash \mathsf{cast}\ w\langle u \rangle :: (w : \downarrow_n^m A^m)}\ \downarrow \text{R} \qquad \frac{\Gamma, x : A^m \vdash P :: (w : B^o)}{\Gamma, u : \downarrow_n^m A^m \vdash x \leftarrow \mathsf{shift}\ u; P :: (w : B^o)}\ \downarrow \text{L}$$

Abstract Types	Concrete Types	Abstract Types	Concrete Types
$A \otimes B$	`A * B`	$\mathbf{1}$	`1`
$A \multimap B$	`A -* B`	t	`t`
$\oplus\{l : A_l\}_{l \in L}$	`+{l1 : A1, ...}`	$\uparrow_m^n A^m$	`m /\ n A`
$\&\{l : A_l\}_{l \in L}$	`&{l1 : A1, ...}`	$\downarrow_n^m A^m$	`m \/ n A`
$t^m = T$	`type t = m T`	l,a,m,r	`lin, aff, mul, rep`

Tbl. 2: Abstract and concrete mapping for types

Runtime. Closed processes execute concurrently by interacting on names assigned to channels. A running process takes the form

$$\mathsf{prc}(N; a; P)$$

The channel on which process P provides is referred to *externally* (by other processes) via the set of names $N = \{b_1,\ldots,b_n\}$ (a unique name per reference), and *internally* by P using name a (prc acts as a name binder for a in P). P may in turn contain other names to refer to channels provided by other processes. The semantics is given in terms of reduction rules; we use "_" in lieu of names, a, or sets of names, N, when irrelevant (and unchanged between redex and reduct).

(CUT) $\mathsf{prc}(\_; \_; x \leftarrow \mathsf{new}\ P; Q) \longrightarrow \mathsf{prc}(\{b\}; c; P[c/x]), \mathsf{prc}(\_; \_; Q[b/x])$

(SND) $\mathsf{prc}(\{a\}; b; \mathsf{send}\ b\langle c,d\rangle), \mathsf{prc}(\_; \_; \langle x,y\rangle \leftarrow \mathsf{recv}\ a; P) \longrightarrow \mathsf{prc}(\_; \_; P[c,d/x,y])$

(RCV) $\mathsf{prc}(\{a\}; b; \langle x,y\rangle \leftarrow \mathsf{recv}\ b; P), \mathsf{prc}(\_; d; \mathsf{send}\ a\langle c,d\rangle) \longrightarrow \mathsf{prc}(\_; d; P[c,d/x,y])$

For instance, CUT spawns a new process $P[c/x]$ while allocating two names, one internal, c, and one external, b (used by the spawning process Q) to refer to the channel provided by the spawnee $P[c/x]$. SND describes the sending by a provider on a channel known by a client via the (external) name a; since communication is asynchronous, the provider terminates after the interaction. Dually, RCV describes the receipt of a message by a process providing on a channel known externally via name a. See Pruiksma et. al. [31] for details.

3   Static Type-Checker

We implement a tool called GRITS, that defines a language for describing message-passing programs that satisfy SAX type specifications and runtime, and execute them. It is implemented in Go [11] and is publicly available on GitHub [36].

Programs are written in a syntax similar to the one in sec. 2, with a few minor discrepancies. Types are written with the syntax mapping in tbl. 2. Moreover processes are allowed to refer to the name of the channel they provide on via the keyword `self`.[4] The structure of a GRITS program follows the general structure

---

[4] Processes fully observe the linearity syntactic constraint since, in $x \leftarrow \mathsf{new}\ P; Q$, variable x is used linearly in Q exclusively, but not in P which uses `self` instead.

outlined below. It starts with a series of type declarations, line 1, followed by a series of *named* process template declarations, line 3, and a single main process that starts the computation, line 5.

```
1   type t = m A                      //  t^m = A^m
2   ...
3   let p(x1:A1,...,xk:Ak):B = P   //  p(x_1,...,x_k) = P  with  x_1:A_1,...,x_k:A_k ⊢ P :: (self:B)
4   ...
5   exec q()
```

Typechecking follows closely the typing rules outlined in sec. 2. Its automation is facilitated by the fact that most typing rules are syntax directed. The only exception is rule CUT; its premises require the analysis to statically guess:

1. the type associated to the channel x provided by the spawned process P
2. how to split the antecedents across the two premises

GRITS solves the first issue by allowing spawning to include type information and be written as $x : A \leftarrow \mathsf{new}\ P; Q$. The second issue is solved by limiting the typing of P to analytic cuts (snips [10]). Concretely, the syntax of P is limited to either asynchronous constructs (*i.e.,* send, select, close and cast) or process calls, since the antecedents in their (axiom) typing rules are precisely determined from the structure of the term. More complex variations for P can always be packaged as a separate process definition and then used via a process call.

Expressivity. The program excerpt below is adapted from [31, Ex. 10]. The `map` process declaration takes two parameters: a map (`f`) and the original list of numbers (`l`). The mapping function `f` is declared at a *replicable mode* since it will be copied and used over each element of the list. Since the list elements are linear numbers, the map has to first shift to linear mode before it is applied.

```
1   type nat = lin +{zero : 1, succ : nat}
2   type listNat = lin +{cons : nat * listNat, nil : 1}
3   type mapType = lin /\ rep (nat -* nat)
```

The type declarations `nat` and `listNat` define natural number and lists recursively, whereas `mapType` defines the type of a replicable map function.

```
4   let map(f:mapType, l:listNat):listNat =
5       case l (
6           cons<l'> => <curr, l''> <- recv l';
7               <f', f''> <- split f;   // map channel 'f' copied
8               fl : lin (nat -* nat) <- new cast f'<self>;
9               curr_upd : nat <- new send fl<curr, self>;
10              k'' : listNat <- new map(f'', l'');
11              k' : nat * listNat <- new send self<curr_upd,k''>;
12              self.cons<k'>
13          | nil<l'> => drop f;        // map channel 'f' unused
14              self.nil<l'>
15      )
16
17  let mapByInc() : mapType  =
18      s <- shift self;
19      <toAdd, result> <- recv s;
20      self.succ<toAdd>                // increment by one 'succ'
```

In the `cons` case of the `map` definition, the process creates two copies of the replicable mapping process (line 7) before downcasting one to `lin` and applying

it to the current element of the list (lines 8, 9). The remaining lines apply the
map recursively to the tail of the list and reconstruct a new list with the mapped
values. In the `nil` case, the mapping process is left unused (line 13). From lines
17 to 20 a replicable mapping process is defined behaving as a successor.

```
21   let main() : listNat =      // main process
22       l : listNat <- new simpList();
23       f : mapType <- new mapByInc();
24       map(f, l)
25
26   exec main() // launch main process
```

The main process initialises a `nat` list process (code elided), launches a replicable
mapping process, and spawns a client `map` process to the former two.

```
27   prc[l] : listNat = simpleList()
28   prc[f] : mapType = mapByInc()
29   prc[b] : listNat = map(f, l)     // f:mapType, l:listNat ⊢ map(f, l) :: (self:listNat)
```

For debugging and modelling purposes, GRITS allow the execution to start from
a particular snapshot, instead of having to launch all execution from one root
process. The alternative launching code above describes three processes that are
already running in parallel providing on channels named `l`, `f` and `b` (lines 27-29).

```
30   assuming l : listNat    // instead of  prc[l] : listNat = simpleList()
31   prc[f] : mapType =  mapByInc()
32   prc[b] : listNat = map(f, l)
```

GRITS also allows programs to be developed compositionally, by eliding parts
of the computation. For instance, line 30 in the excerpt above does *not* specify
the precise code of the process providing at channel `l`, instead describing its
interface specification. This still permits the program to be typechecked.

Bank Service Example. The services offered by a hypothetical bank are for-
malised by the affine type `bankType` (lines 1 to 2 below). Two choices are ini-
tially offered: a `login` option and another one for general queries (`gen_query`)
regarding opening times (details omitted). A `login` request is answered by either
an `auth` (authenticated) or an `not_auth` response; the latter response allows the
user to retry logging again. An authenticated user can initiate a transaction.
For this, the interaction must shift into linear mode ($\uparrow_{\mathsf{l}}^{\mathsf{a}}$ `transaction`) to force
transaction termination; the labels `start` and `finish` delimit a (*dummy*) trans-
action.

```
1   type bankType = aff &{ login      : authType
2                          gen_query : ... }
3   type authType = aff +{ auth      : lin /\ aff transaction ,
4                          not_auth : bankType }
5   type transaction = lin +{ start : +{ finish : 1 } }
```

The behaviour dictated by the type `bankType` is implemented and typechecked
using GRITS. The `bank` process waits to receive either a `login` or `gen_query` la-
bel (line 7). An authenticated user (line 8) is handled by `authService()` (line 8),
where a shift (into linear mode, line 14) is performed before executing the trans-
action (lines 15-17).

```
 6   let bank() : bankType =
 7       case self (
 8           login<s>      => auth <- new authService();
 9                            self.auth<auth>
10         | gen_query<s> => ...
11       )
12
13   let authService() : lin /\ aff transaction =
14       s' <- shift self;             // handle transaction in linear mode
15       s'' : lin 1 <- new close self;
16       s''' : lin +{finish : 1} <- new self.finish<s''>;
17       self.start<s'''>
```

The execution launched below models an *indecisive* user (`user1`, line 18), who initiates an interaction but promptly cancels it (line 19); this is permitted by the `bankService`'s affine mode.

```
18   prc[bankService] : bankType = bank()
19   prc[user1] : lin 1 = drop bankService;
20                        close self
```

Conversely, a different user (`user2`, line 23) requests a login and waits to be authenticated (line 25). After shifting modes (line 26), the interaction with `bankService` proceeds in linear mode (lines 26-31).

```
21   prc[user2] : lin 1 =
22       print _attempt_login_;                          //stdout notification
23       b : authType <- new bankService.login<self>;
24       case b (
25           authenticated<b'> =>
26               t :  transaction <- new cast b'<self>; //cannot drop t (linear)
27               case t (
28                  start<t'> =>  case t' (
29                                  finish<t''> => wait t''; close self
30                              )
31               )
32         | not_authenticated<b'> => drop b'; close self
33       )
```

4   Runtime Interpreter

Typechecked programs are executed by an interpreter that leverages the concurrency features of the Go language. Every process is mapped to a goroutine that provides on a dedicated channel. This one-to-one mapping of concurrency units allows us to better assess the implementability of the proposed model.

Copy Semantics. Contractible processes (*i.e.,* in multicast or replicable mode) can be *assigned* multiple names using $\langle x,y \rangle \leftarrow \mathsf{split}\ u; P$. The following reduction rules achieve this in two steps (the suggestive name ι is used to denote the internal name of the channel provided, analogous to the keyword `self`).

$$(\text{SPL}) \qquad \mathsf{prc}(\{a\}; \iota; \langle x,y \rangle \leftarrow \mathsf{split}\ b; P) \longrightarrow \begin{array}{l} \mathsf{prc}(\{c,d\}; \iota; \mathsf{fwd}\ \iota\ b), \\ \mathsf{prc}(\{a\}; \iota; P[^{c,d}/_{x,y}]) \end{array}$$

$$(\text{FWD}) \qquad \mathsf{prc}(\{b\}; \iota; P),\ \mathsf{prc}(N; \iota; \mathsf{fwd}\ \iota\ b) \longrightarrow \mathsf{prc}(N; \iota; P)$$

Rule SPL generates two new names for the name being duplicated, connecting them via *forwarding*, which then reacts with the process providing on the name being duplicated to increase its set of external names, rule FWD.

$$(\textsc{Dup}) \quad \mathsf{prc}(\{a,b\}; \iota; P) \longrightarrow \begin{array}{c} \mathsf{prc}(a; \iota; P\sigma_1), \ \mathsf{prc}(b; \iota; P\sigma_2), \\ \big\{\mathsf{prc}(\{c\sigma_1, c\sigma_2\}; \iota; \mathsf{fwd}\ \iota\ c)\big\}_{c \in \mathbf{fn}(P)\backslash\{\iota\}} \end{array}$$

$$\text{where } P \neq \mathsf{fwd}\ \_\ \_ \text{ and } \mathbf{rename}(\mathbf{fn}(P)\backslash\{\iota\}) = \langle\sigma_1,\sigma_2\rangle$$

Processes with multiple names are given a *copy* semantics. Rule DUP generates a process copy P for each of the two name references a and b. By the hierarchical arrangement resulting from typechecking, process P is the root of a tree of processes that need to be duplicated as well. This is done in two steps. For every reference P has towards its clients (*i.e.,* immediate children), rule DUP generates two new (unique) names using $\mathbf{rename}(\mathbf{fn}(P)\backslash\{\iota\}) = \langle\sigma_1,\sigma_2\rangle$ (σ_1 and σ_2 are maps from names to names) and renames the two copies of P accordingly, *i.e.,* $P\sigma_1$ and $P\sigma_2$. Moreover, for *every* (externally) renamed name in P, $c \in \mathbf{fn}(P)\backslash\{\iota\}$, it creates a forwarding associating it to its renaming, $c\sigma_1$ and $c\sigma_2$. This results in a downwards chain of duplications to all child (provider) processes.
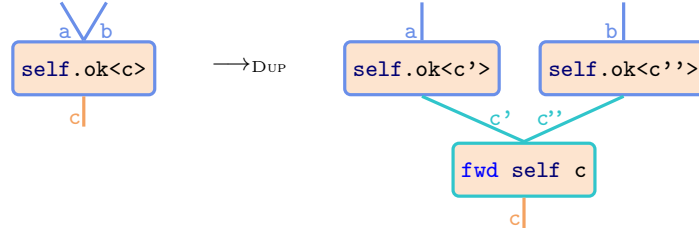

For example, a process `self.ok<c>` (depicted above), which is multiply referenced by the names a and b, is split into two copies where each copy renames the client reference name c to c' and c''. A corresponding forwarding process is also created to propagate the copying to the client process with name c via a combination of the rules FWD and DUP.

$$(\textsc{Grc}) \quad \mathsf{prc}(\emptyset; \iota; P) \longrightarrow \big\{\mathsf{prc}(\emptyset; \iota; \mathsf{fwd}\ \iota\ a)\big\}_{a \in \mathbf{fn}(P)\backslash\{\iota\}} \qquad \text{where } P \neq \mathsf{fwd}\ \_\ \_$$

Dually, unreferenced processes, *i.e.,* $N = \emptyset$, trigger a cascading garbage collection procedure to its clients via forwarding; rule GRC above. Corresponding this reduction discipline, processes executing non-structural commands (*e.g.* rules SND and RCV) only become active *when* they are referenced by a *single* name.

Synchronous and Asynchronous Implementations. In a hierarchically organised soup of processes that are typechecked according to a SAX specification, messages can flow in two directions: either from a provider (bottom process) to its client (top process) or vice versa. For example, in a RCV reduction (diagram below, left), messages flow *downwards* from a client, while in a SND reduction (below, right), messages flow *upwards* towards the client.[5]

---

[5] Other non-structural reductions (*e.g.* label branching and shifting) behave similarly.
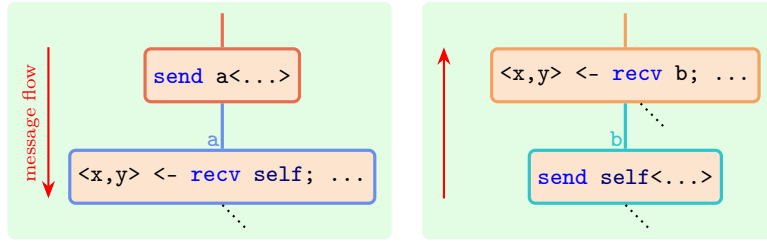
Although direct interactions between matching processes (*e.g.* send $u\langle v,w\rangle$ with $\langle x,y\rangle \leftarrow$ recv $u;P$ processes) are straightforward to implement, the possibility of having proxy processes mediating via forwarding (in order to alter the hierarchical structure) complicate the implementation of the communication protocol for the interpreter. Our interpreter offers two implementations to support our study: a synchronous setup, where Go channels are *unbuffered*, and an asynchronous setup with *buffered* channels.
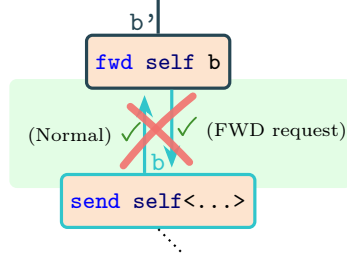
Synchronous. The synchronous setting employs *two* Go channels per provider. A *data channel* is used for non-structural interactions such as send, (label) select and cast (see rules SND and RCV above). In addition, a *control channel* is dedicated to structural interactions such as splitting and garbage-collection, which are all conducted via forwarding (see rules SPL and GRC above and tbl. 1). The implementation then makes use of the Go `select` construct to interact on either of these channels depending on the surrounding process context.
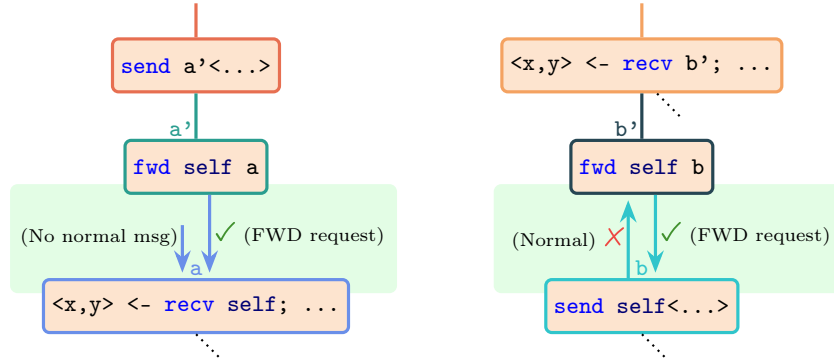
Consider a variant of the previous two send and recv examples with a forward process in between. In both the left and right cases, the fwd process is executed uniformly by the interpreter: it sends a channel name on the control channel indicating to the respective provider underneath it the name of the (new) data channel to listen on (instead of the existing one). In the left scenario, a client attempts to send a message downwards on channel a'; since the process providing on a' (the fwd process) is not ready to receive on the (synchronous) Go channel, the communication blocks. Conversely, the recv process providing on channel a waits for messages on *both* data and control channels. The fwd process does not communicate on the data channel, but instead sends on the control channel. This eventually succeeds, generating a provider recv process waiting on channel a', which can now react with the client sending on this channel.

In the right scenario, the provider on channel b is ready to send on the data channel while, simultaneously, waiting to receive on the control channel, resulting in a *mixed choice* [26]. This turns out *not* to be problematic in a synchronous setting. Concretely, since the forwarding process is not ready to receive on the data channel b, the message sending blocks. However, the forwarding process successfully sends a forwarding request on the control channel, as the send provider process on channel b is ready to accept it, completing a FWD reduction.

Asynchronous. In a setup with buffered channels (where sending does not require a *handshake* from the other channel endpoint), the execution strategy for the


right hand scenario discussed above *fails*. In an asynchronous setting (depicted on the side), data messages will be sent *upwards*, in the opposite direction of the forwarding requests (which are sent downwards). Since neither are blocking, both sending of messages will succeed in reaching the respective channel buffer. Nevertheless, neither message will eventually be read off this buffer, leading to two deadlocked processes.

One compositional solution to this problem is *not* to use a uniform forwarding behavior. In an asynchronous setting, our interpreter categorises the forward construct into two: an *active* (fwd^A) or *passive* forward (fwd^P), aligning with message flow direction. Apart from localising the change to the forwarding construct, this change allows us to collapse the data and control channels and just use one.

$$(\text{FWD}_\text{P}) \qquad \mathsf{prc}\big(b; \gamma; P^+\big),\ \mathsf{prc}\big(N; \iota; \mathsf{fwd}^\mathrm{P}\ \iota\ b\big),\ \longrightarrow\ \mathsf{prc}\big(N; \gamma; P^+\big)$$

$$(\text{FWD}_\text{A}) \qquad \mathsf{prc}\big(b; \gamma; P^-\big),\ \mathsf{prc}\big(N; \iota; \mathsf{fwd}^\mathrm{A}\ \iota\ b\big),\ \longrightarrow\ \mathsf{prc}\big(N; \gamma; P^-\big)$$

When messages flow upwards, *i.e.,* messages originate from a provider sending on the data channel (P^+ ranges over send $u\langle v,w\rangle$, $u.l\langle v\rangle$, close u and cast $u\langle v\rangle$), they may interact with a *passive* forwarding process. This forwarding process passively waits for incoming messages before reducing to a P^+ processes themselves (FWD$_\text{P}$). Conversely, when a synchronous process (P^- includes the remaining non-structural constructs) expects incoming messages from a client, it may interact with an *active* forwarding process. Similar to the synchronous case, *active* forwards initiate the interaction by sending a forwarding request message (FWD$_\text{A}$). Examples of passive and active forward processes are depicted below.
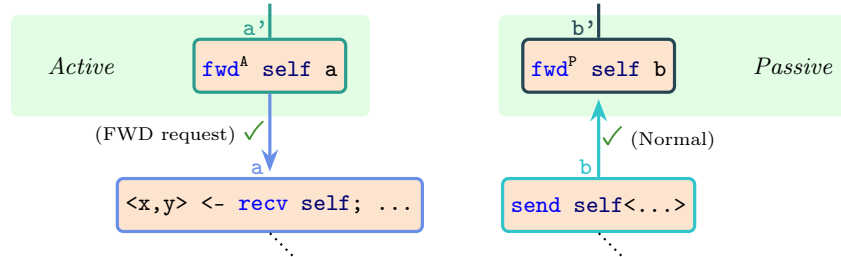
In [30, 29], different versions for the forward processes are also explored, depending on the *polarity* of the forwarded channels. From an implementation perspective this is similar to how we infer the direction of message flow, where messages flowing *upwards* use positive channels (and passive forwards), while messages flowing *downwards* use negative channels (and active forwards). The information the channel polarities is obtained from the associated types in sec. 2.

5   Evaluation

The objective of this study is to evaluate the implementability of the proposed asynchronous message-passing interpretation for Sax. Secs. 3 and 4 provide evidence that this can be accomplished using a concurrent implementation. This section assesses whether this implementation is satisfactory in terms of its ability to scale with the number of spawned processes, thereby utilising any underlying multicore architecture.

Setup. In order to measure the scalability of our implementation, we make use of Sax Tool [28], which is the only other existing implementation of a message-passing interpretation for Sax. The fact that we can express common programs in both Grits and Sax Tool allows us to use the latter as a baseline for comparing runtime performance. Although Sax Tool is a pedagogic tool with limited focus on performance, it was developed using Standard ML: programs are executed in a sequential setup which does not exploit any underlying parallel architecture. In contrast, the implementation discussed in sec. 4 maintained a one-to-one mapping to goroutines. Our evaluation utilises two inherently concurrent programs that can be parameterised to scale with the number of running processes. We compare the respective execution time executed over Grits (using synchronous and asynchronous semantics), against Sax Tool. All experiments were carried out on a Apple M2 Pro (10-core) CPU machine with 16GB of memory, running Go 1.21.6 on macOS 14. The respective readings are reported in the two graphs of fig. 2.

Natural Number Doubling. A program that can be interpreted both in Grits and Sax Tool is a number doubling procedure (adopted from [28]). It is defined as a process definition, `double`, that consumes a number and provides another number doubled in value. We reuse the unary natural number type `nat` from sec. 3, which allows us to represent natural numbers using a series of successor labels, *e.g.* `+{succ: +{succ: +{zero: 1}}}` represents the number 2. The process recursively constructs a new number by first deconstructing the number being received from a provider (`x`), and for every `succ` label obtained, two are sent instead (lines 5-6), recursing until the number is fully exhausted (line 3).

```
1   type nat = lin +{zero : 1, succ : nat}
2   let double(x : nat) : nat =
3       case x ( zero<x'> => self.zero<x'>
4               | succ<x'> => h <- new double(x');
5                             d : nat <- new d.succ<h>; // first succ
6                             self.succ<d>              // second succ
7       )
```
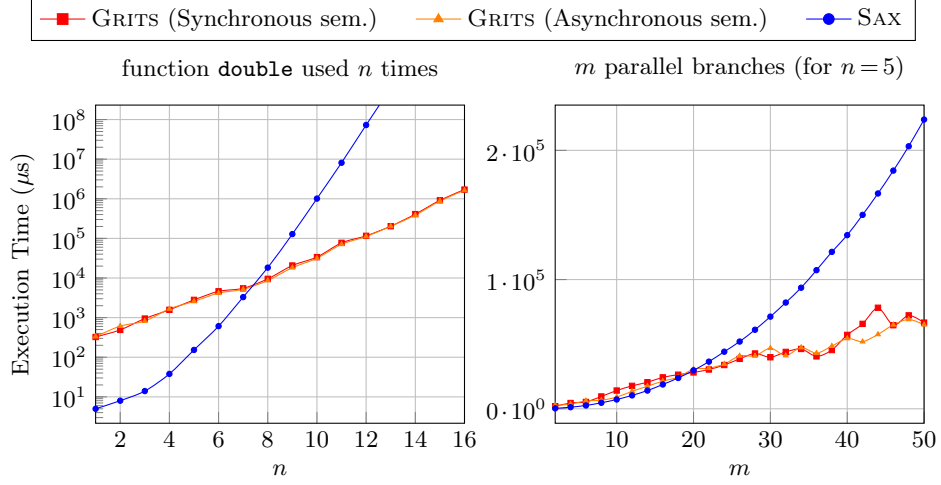
Fig. 2: Performance benchmarks comparing the different semantic implementations (from sec. 4) with Sax Tool. The *execution time* axis for the left graph is *logarithmic*, while the right one uses normal axis.

Sequential Doubling. For the first evaluation scenario, we invoke `double` multiple times in sequence, producing a natural number with an exponential size. *E.g.*, for the process providing on `n1` representing the number one (line 8) we double twice "in sequence" (lines 13 and 14) to obtain the final value of 4 $(= 1 \times 2^2)$.

```
8    prc[n1] : nat = // Produces the natural number 1, i.e. succ(zero)
9        t : 1 <- new close t;
10       z : nat <- new z.zero<t>;
11       self.succ<z>
12   prc[b] : nat =
13       d1 <- new double(n1);
14       d2 <- new double(d1); // double used twice
15       fwd self d2
```

The evaluation varies the number of times (n) the doubling function is repeated, to produce a number with exponential size $(= 2^n)$. The results are reported in fig. 2 (left), showing the time taken (μs) by the interpreter to finish executing. The Grits concurrent implementation initially performs less efficiently for smaller programs $(n \leq 7)$. This behavior is attributed to the overhead incurred when spawning new threads with a very short lifespan. However, as the program size increases, it exhibits better scalability, outperforming Sax Tool.

Concurrent Doubling. The second evaluation scenario induces more parallelism. It invokes the aforementioned sequential doubling procedure, fixed at $n = 5$, for m times *concurrently*, generating a forest-like structure with m *parallel* trees. The benchmark results (fig. 2, right) show a similar trend, with Sax Tool outperforming our implementation for smaller programs, but the situation reversing

for larger programs. We even observe that the execution of the Grits concurrent implementation appears to grow linearly for the readings taken.

Results. Despite limiting our experiments to the testing sizes of $1 \leq n \leq 16$ and $2 \leq m \leq 50$, the readings from both graphs in fig. 2 exhibit a clear trend. This allows us to extrapolate and conclude that the proposed model of Pruiksma et. al. [31] can be implemented adequately in concurrent fashion in order to be able to scale. Although the asynchronous implementation of sec. 4 suggests that the proposed model can also be implemented in a distributed setting where processes are dispersed across different locations, we cannot draw conclusive evidence as to how this performs in relation to the synchronous variant. This might stem from the fact that the Sax model uses short-lived processes which might perform very little work before terminating. For instance, instead of spawning a process just to send a single message, a more efficient way would be to send a message sequentially from an existing process.

Threats to Validity. The experiment setup could have suffered from limited *granularity control*, which is generally difficult to automate [39]. Our results rely on Go's handling of concurrency to maximise the underlying parallel hardware. We did not consider higher process numbers to avoid the risk of running into stack overflows once certain system limits are hit; this would have been caused by our implementation design, where each reduction is performed via a function call, and by Go's lack of support for tail-recursion optimisation [17]. Our choice of experiment programs could have also introduced biases. Similarly, our choice for a baseline, namely Sax Tool, could have also affected our scalability assessments.

6   Related Work

We mainly compare our work with other implementations based on the Curry-Howard interpretation [4] of intuitionistic linear logic. This induces a hierarchical structuring of concurrent processes, thereby avoiding the need to use type duality inherent in binary session type implementations [15, 20, 25, 34], or type projections, in the case of multiparty session types [12, 21, 22, 24].

To our knowledge, the only implementation based on the semi-axiomatic sequent calculus is Sax Tool [28], used to establish a baseline in sec. 5. At the time of writing, Sax Tool only support linear modes, and the mode-shifting programs in sec. 3 cannot be expressed. Internally, Sax Tool relies on a sequential implementation in Standard ML to simulate concurrency, while Grits uses native concurrency primitives to contend with the intricacies of the copy semantics discussed in sec. 4.

Das et. al. [9] introduce Rast, a language integrating session types with arithmetic refinements. It embeds assertions within the types to also account for a program execution's work and span. Similar to Sax Tool, Rast follows a sequential interpretation. Additionally, Das et. al. extended the work to obtain Nomos [8], which applies resource-aware session types to smart contracts.

Instead of developing bespoke languages, other projects integrate intuitionistic session types over existing languages. Ferrite [6] extends the SILL calculus [29, 37] with process sharing [3] mechanisms to introduce shared session types as a Rust library. CC0 [40, 35], a session-based extension of C0, adopts asynchronous message-passing semantics. It offers a Go back end using goroutines for processes and channels for message passing. For this variant, CC0 uses separate channels for bidirectional communication, which was not needed in our tool due to Sax's model. The Go runtime version was compared and shown to outperform a separate C-based implementation, where more heavyweight *pthreads* were used for concurrent processes.

Polite by Lakhani et al. [23] adopts a form of adjoint modalities to control type polarities, similar to SILL's approach to message flow reversal [29]. Since our channels are single-use, shifting allows us to seamlessly transition between modalities. Nomos [8] and Ferrite [6] use modes and shifts to obtain exclusive access to shared processes. Similar to the handling of forwarding in an asynchronous implementation, discussed in sec. 4, the work in [9, 8, 32, 29] adopt forwarding behavior based on polarity. Other frameworks, such as SILL and $SILL_S$ [3], utilise global substitutions for forwarding, a strategy unsuitable for our decentralised implementation. An adaptation of SILL is also used by Caires and Toninho [5] to study a fully sequential and deterministic evaluation strategy.

7   Conclusion

We investigate the implementability of the message-passing interpretation of Sax proposed in previous work [31]. This is conducted by building the tool Grits, the first type-checker and interpreter to *fully* handle message-passing programs satisfying Sax specifications. The interpreter executes programs in decentralised fashion, leveraging concurrency features from the Go language such as *goroutines*. Our empirical evaluation leads us to conclude that Sax's [31] proposed model is not based on any infeasible assumptions that might prevent it from being implemented in a concurrent fashion.

Future Work. We plan to expand on the Sax semantics by integrating notions of shared processes [3, 8, 6] that co-exists alongside replicated processes with a copy semantics. Another planned extension is to consider the actor model [19, 2] as an interpretation for Sax. Finally, we would like to investigate the use of Sax as a basis for the systematic instrumentation of detection monitors [13], partial-identity monitors [18] and enforcement monitors [1] for added assurances related to the temporal properties of program data; the Sax proof system can also be leveraged to enhance verdict explainability when property violations are detected [14].

References

[1] Aceto, L., Cassar, I., Francalanza, A., Ingólfsdóttir, A.: Bidirectional Runtime Enforcement of First-Order Branching-Time Properties. Log. Methods Comput. Sci. 19(1) (2023)

[2] Agha, G.A.: ACTORS - a model of concurrent computation in distributed systems. MIT Press series in artificial intelligence, MIT Press (1990)

[3] Balzer, S., Pfenning, F.: Manifest sharing with session types. Proc. ACM Program. Lang. 1(ICFP), 37:1–37:29 (2017)

[4] Caires, L., Pfenning, F.: Session types as intuitionistic linear propositions. In: CONCUR. Lecture Notes in Computer Science, vol. 6269, pp. 222–236. Springer (2010)

[5] Caires, L., Toninho, B.: The session abstract machine. In: Weirich, S. (ed.) Programming Languages and Systems - 33rd European Symposium on Programming, ESOP 2024, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2024, Luxembourg City, Luxembourg, April 6-11, 2024, Proceedings, Part I. Lecture Notes in Computer Science, vol. 14576, pp. 206–235. Springer (2024)

[6] Chen, R., Balzer, S., Toninho, B.: Ferrite: A judgmental embedding of session types in rust. In: Ali, K., Vitek, J. (eds.) 36th European Conference on Object-Oriented Programming, ECOOP 2022, June 6-10, 2022, Berlin, Germany. LIPIcs, vol. 222, pp. 22:1–22:28. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2022)

[7] Crary, K.: Higher-order representation of substructural logics. In: ICFP. pp. 131–142. ACM (2010)

[8] Das, A., Balzer, S., Hoffmann, J., Pfenning, F., Santurkar, I.: Resource-aware session types for digital contracts. In: 34th IEEE Computer Security Foundations Symposium, CSF 2021, Dubrovnik, Croatia, June 21-25, 2021. pp. 1–16. IEEE (2021)

[9] Das, A., Pfenning, F.: Rast: A language for resource-aware session types. Log. Methods Comput. Sci. 18(1) (2022)

[10] DeYoung, H., Pfenning, F., Pruiksma, K.: Semi-axiomatic sequent calculus. In: FSCD. LIPIcs, vol. 167, pp. 29:1–29:22. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2020)

[11] Effective Go: Effective Go - The Go Programming Language (nd), https://go.dev/doc/effective_go#sharing

[12] Fowler, S.: An Erlang implementation of multiparty session actors. In: Bartoletti, M., Henrio, L., Knight, S., Vieira, H.T. (eds.) Proceedings 9th Interaction and Concurrency Experience, ICE 2016, Heraklion, Greece, 8-9 June 2016. EPTCS, vol. 223, pp. 36–50 (2016)

[13] Francalanza, A.: A Theory of Monitors. Inf. Comput. 281, 104704 (2021)

[14] Francalanza, A., Cini, C.: Computer says no: Verdict explainability for runtime monitors using a local proof system. J. Log. Algebraic Methods Program. 119, 100636 (2021)

[15] Francalanza, A., Tabone, G.: Elixirst: A session-based type system for Elixir modules. J. Log. Algebraic Methods Program. 135, 100891 (2023)

[16] Gay, S.J., Hole, M.: Subtyping for session types in the pi calculus. Acta Informatica 42(2-3), 191–225 (2005)

[17] Golang GitHub: Golang GitHub issue #22624 (2017), https://github.com/golang/go/issues/22624

[18] Gommerstadt, H., Jia, L., Pfenning, F.: Session-typed concurrent contracts. J. Log. Algebraic Methods Program. 124, 100731 (2022)

[19] Hewitt, C., Bishop, P.B., Steiger, R.: A universal modular ACTOR formalism for artificial intelligence. In: Nilsson, N.J. (ed.) Proceedings of the 3rd International Joint Conference on Artificial Intelligence. Standford, CA, USA, August 20-23, 1973. pp. 235–245. William Kaufmann (1973)

[20] Honda, K.: Types for dyadic interaction. In: CONCUR. Lecture Notes in Computer Science, vol. 715, pp. 509–523. Springer (1993)

[21] Honda, K., Yoshida, N., Carbone, M.: Multiparty asynchronous session types. J. ACM 63(1), 9:1–9:67 (2016)

[22] Lagaillardie, N., Neykova, R., Yoshida, N.: Implementing multiparty session types in rust. In: Bliudze, S., Bocchi, L. (eds.) Coordination Models and Languages - 22nd IFIP WG 6.1 International Conference, COORDINATION 2020, Held as Part of the 15th International Federated Conference on Distributed Computing Techniques, DisCoTec 2020, Valletta, Malta, June 15-19, 2020, Proceedings. Lecture Notes in Computer Science, vol. 12134, pp. 127–136. Springer (2020)

[23] Lakhani, Z., Das, A., DeYoung, H., Mordido, A., Pfenning, F.: Polarized subtyping. In: Sergey, I. (ed.) Programming Languages and Systems - 31st European Symposium on Programming, ESOP 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings. Lecture Notes in Computer Science, vol. 13240, pp. 431–461. Springer (2022)

[24] Neykova, R., Yoshida, N.: Multiparty session actors. Log. Methods Comput. Sci. 13(1) (2017)

[25] Padovani, L.: A simple library implementation of binary sessions. J. Funct. Program. 27, e4 (2017)

[26] Palamidessi, C.: Comparing the expressive power of the synchronous and asynchronous pi-calculi. Math. Struct. Comput. Sci. 13(5), 685–719 (2003)

[27] Pfenning, F.: Lecture notes on Adjoint SAX (2023), Course notes for Substructural Logics (15-836)

[28] Pfenning, F.: Lecture notes on Semi-Axiomatic Sequent Calculus (2023), Course notes for Substructural Logics (15-836). Accompanying tool available from https://www.cs.cmu.edu/~fp/courses/15836-f23/resources.html

[29] Pfenning, F., Griffith, D.: Polarized substructural session types. In: Pitts, A.M. (ed.) Foundations of Software Science and Computation Structures - 18th International Conference, FoSSaCS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings. Lecture Notes in Computer Science, vol. 9034, pp. 3–22. Springer (2015)

[30] Pfenning, F., Pruiksma, K.: Relating message passing and shared memory, proof-theoretically. In: Jongmans, S., Lopes, A. (eds.) Coordination Models and Languages - 25th IFIP WG 6.1 International Conference, COORDINATION 2023, Held as Part of the 18th International Federated Conference on Distributed Computing Techniques, DisCoTec 2023, Lisbon, Portugal, June 19-23, 2023, Proceedings. Lecture Notes in Computer Science, vol. 13908, pp. 3–27. Springer (2023)

[31] Pruiksma, K., Pfenning, F.: A message-passing interpretation of adjoint logic. J. Log. Algebraic Methods Program. 120, 100637 (2021)

[32] Pruiksma, K., Pfenning, F.: Back to futures. J. Funct. Program. 32, e6 (2022)

[33] Sano, C., Kavanagh, R., Pientka, B.: Mechanizing session-types using a structural view: Enforcing linearity without linearity. Proc. ACM Program. Lang. 7(OOPSLA2), 374–399 (2023)

[34] Scalas, A., Yoshida, N.: Lightweight session programming in scala. In: Krishnamurthi, S., Lerner, B.S. (eds.) 30th European Conference on Object-Oriented Programming, ECOOP 2016, July 18-22, 2016, Rome, Italy. LIPIcs, vol. 56, pp. 21:1–21:28. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2016)

[35] Silva, M.E.P., Florido, M., Pfenning, F.: Non-blocking concurrent imperative programming with session types. In: Cervesato, I., Fernández, M. (eds.) Proceedings Fourth International Workshop on Linearity, LINEARITY 2016, Porto, Portugal, 25 June 2016. EPTCS, vol. 238, pp. 64–72 (2016)

[36] Tabone, G.: Grits: Implementing a Message-Passing Interpretation of the Semi-Axiomatic Sequent Calculus (Sax) (artefact for Coordination'24) (Mar 2024), https://doi.org/10.5281/zenodo.10837897, https://github.com/gertab/Grits

[37] Toninho, B., Caires, L., Pfenning, F.: Higher-order processes, functions, and sessions: A monadic integration. In: Felleisen, M., Gardner, P. (eds.) Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings. Lecture Notes in Computer Science, vol. 7792, pp. 350–369. Springer (2013)

[38] Wadler, P.: Propositions as sessions. In: Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming. p. 273–286. ICFP '12, Association for Computing Machinery, New York, NY, USA (2012)

[39] Westrick, S., Fluet, M., Rainey, M., Acar, U.A.: Automatic parallelism management. In: Proceedings of the ACM on Programming Languages. vol. 8, pp. 1118–1149 (Jan 2024)

[40] Willsey, M., Prabhu, R., Pfenning, F.: Design and implementation of Concurrent C0. In: Fourth International Workshop on Linearity. pp. 73–82. EPTCS 238 (Jun 2016)

A   Further Examples

We express further examples including manipulation of lists and trees from [28], along with an encoding of the examples from [31]. All examples shown are type-checked using GRITS, and are available with the tool.[6]

Natural Number List Example. The following is a simple list that stores the numbers one and zero. This was omitted from sec. 3.

```
1   // Provide a list containing cons(1, cons(0, nil))
2   let simpList() : listNat =
3       n1'' : lin 1 <- new close self;
4       n1'  : nat    <- new self.zero<n1''>;
5       n1 : nat      <- new self.succ<n1'>; // succ(zero)
6       n0' : lin 1   <- new close self;
7       n0 : nat      <- new self.zero<n0'>; // zero
8
9       lnil' : lin 1       <- new close self;
10      lnil : listNat      <- new self.nil<lnil'>;
11      l0' : nat * listNat <- new send self<n0, lnil>;
12      l0 : listNat        <- new self.cons<l0'>;
13      l1' : nat * listNat <- new send self<n1, l0>;
14      self.cons<l1'>
```

List Printing. In order to better understand how the data is being manipulated, we can utilise the printing construct. For instance we can preview the contents of a natural number list (line 15) by using the printListNat process definition (line 16) defined below (lines 17-41).

```
15  prc[a] : listNat = simpList()
16  prc[b] : lin 1 = printListNat(a)
```

The printListNat process declaration takes a list and consumes each element sequentially, printing its contents.

```
17  let printListNat(l : listNat) : lin 1 =
18      y <- new consumeListNat(l);
19      wait y;
20      close self
21
22  let consumeListNat(l : listNat) : lin 1 =
23      case l ( cons<c> => print _cons_;
24                          <element, remainingList> <- recv c;
25                          elementDone <- new consumeNat(element);
26                          wait elementDone;
27                          rightDone <- new consumeListNat(remainingList);
28                          wait rightDone;
29                          close self
30             | nil<c> =>  print _nil_;
31                          wait c; close self
32      )
33
34  let consumeNat(n : nat) : lin 1 =
35      case n ( zero<c> => print zero; wait c; close self
36             | succ<c> => print succ; consumeNat(c))
37
38  let printNat(n : nat) : lin 1 =
39      y <- new consumeNat(n);
40      wait y;
41      close self
```

---

[6] https://github.com/gertab/Grits/tree/main/examples (archived: 10.5281/zen-odo.10732024).

List Append. We illustrate the *append* function, which concatenates two lists of binary numbers. Binary numbers (`bin`) and lists (`listBin`) are represented using the following types:

```
1   type bin = +{b0 : bin, b1 : bin, e : 1}
2   type listBin = +{cons : bin * listBin, nil : 1}
```

For instance, the number 6 is represented as `e b1 b1 b0` (ignoring the `+{...}` symbols), and a list containing 6 and 1 is represented as `cons(e b1 b1 0, cons(e b1, nil))`. The `append` function is defined as follows.

```
3    let append(l1 : listBin, l2 : listBin) : listBin =
4        case l1 (
5                cons<c> =>
6                    <x, l1'> <- recv c;
7                    remainingL <- new append(l1', l2);
8                    reorderedL : bin * listBin <- new (send self<x, remainingL>);
9                    self.cons<reorderedL>
10              | nil<c> => wait c;
11                     fwd self l2
12       )
```

It works by deconstructing the first list (`l1`, line 4) and in the case where it is not empty, then the following binary number is appended to the other list `l2` (lines 7-8).

List Reverse. We can also reverse a list as shown in the process definition `reverse`.

```
13   let reverse(l : list) : list =
14       c : 1 <- new close self;
15       nilList : list <- new self.nil<c>;
16       reverse_inner(l, nilList)
17
18   let reverse_inner(l : listBin, accum : listBin) : listBin =
19       case l ( cons<p> =>
20                    <x, l'> <- recv p;
21                    t : bin * listBin <- new send self<x, accum>;
22                    accum2 : listBin <- new self.cons<t>;
23                    reverse_inner(l', accum2)
24              | nil<u> =>
25                    wait u;
26                    fwd self accum
27       )
```

To reverse a list `l`, we start by preparing another temporary empty list (line 15) and call the `reverse_inner` process definition (line 16) passing the two lists as parameters. Then, `reverse_inner` takes the first element from `l` (line 19), and appends it to the beginning of the other list (line 22). When the list being reverse is fully exhausted (line 24), then the temporary list is provided as a result (line 26).

MapReduce. We consider the MapReduce program described in [27] but not implemented by Sax Tool due to the use of modalities. In addition to the `nat` type, we use the `treeNat` type which defines a tree of natural numbers.

```
28   type treeNat = lin +{node : treeNat * treeNat, leaf : nat}
```

The function `mapreduce` takes parameters for reduction (`fs`), mapping (`hs`) and the original tree of natural numbers (`t`). Note that the names `fs` and `hs` have (an overall) modality of replicable since they will be duplicated and used multiple times to eventually produce the expected result. This contrasts with the inputted tree `t` which is linear – this restriction is enforces that the tree is traversed exactly once.

```
29    let mapreduce(fs : lin /\ rep ((nat * nat) -* nat),
30                  hs : lin /\ rep (treeNat -* nat),
31                  t : treeNat) : nat =
32        case t (
33              node<t'> => <l, r> <- recv t';
34                          <fs', fs''> <- split fs;        // Duplicate fs
35                          <fs'', fs'''> <- split fs'';
36                          <hs', hs''> <- split hs;        // Duplicate hs
37
38                          // Traverse the child nodes
39                          y1 <- new mapreduce(fs', hs', l);
40                          y2 <- new mapreduce(fs'', hs'', r);
41
42                          // Perform the reduction part
43                          p : nat * nat <- new send self<y1, y2>;
44                          fl : ((nat * nat) -* nat) <- new cast fs'''<self>;
45                          send fl<p, self>
46
47            | leaf<t'> => // Perform the mapping part
48                          hl : lin (nat -* nat) <- new cast hs<self>;
49                          drop fs;
50                          send hl<t', self>
51        )
```

Examples From [31]. Furthermore, GRITS can verify formal examples presented in [31], using the Curry-Howard correspondence [4, 38], where session types represent propositions and processes act as proofs. For instance, we mechanize a proof demonstrating the distribution of up shifts over implication, as discussed by Pruiksma and Pfenning [31, Ex. 5]. Formally,

$$f : \uparrow_l^m (A^l \multimap B^l) \vdash P :: (result : (\uparrow_l^m A^l) \multimap (\uparrow_l^m B^l))$$

where P is the program representing the proof required. We show that this holds by defining process definition `upDist`, which successfully typechecks for up shifting from linear to multicast (however this works for other modes as wells).

```
52    type A = lin 1        // A and B can be any linear type
53    type B = lin 1 * 1
54
55    type before = lin /\ mul (A -* B)
56    type after = (lin /\ mul A) -* (lin /\ mul B)
57
58    let upDist(f : before) : after =
59        <x, y> <- recv self;
60        y' <- shift self;
61        x' : A <- new cast x<x'>;
62        f' : A -* B <- new cast f<f'>;
63        send f'<x',self>
```

The remaining examples from [31] are listed below.

```
1   // Example 1
2   // A1 * B1 ⊢ B1 * A1 (showing types only)
3
4   type A1 = mul 1 * 1
5   type B1 = mul 1
6   let eg1(x : A1 * B1) : B1 * A1  =
7       <y, x'> <- recv x;
8       send self<x', y>
9
10  // Example 2
11  // +{left : A2, right : B2}, &{left : A2, right : B2} ⊢ A2 * B2
12
13  type A2 = lin 1
14  type B2 = lin 1
15  type lr = +{left : A2, right : B2}
16  type lr' = &{left : A2, right : B2}
17
18  let eg2(x : lr, y : lr') : A2 * B2  =
19      case x (
20          left<x'>  => y' : B2 <- new y.right<self>;
21                              send self<x', y'>
22          | right<x'> => y' : A2 <- new y.left<self>;
23                              send self<y', x'>
24      )
25
26  // Example 3
27  // &{left : A3, right : B3} ⊢ A3 * B3 (modes must admit contraction)
28
29  type A3 = rep 1
30  type B3 = rep 1
31
32  type C3 = &{left : A3, right : B3}
33
34  let eg3(p : C3) : A3 * B3 =
35      q : C3 <- new fwd self p;
36      <p1, p2> <- split q;
37      x : A3 <- new p1.left<self>;
38      y : B3 <- new p2.right<self>;
39      send self<x, y>
40
41  // Example 4
42  // A4 * B4 ⊢ &{left : A4, right : B4} (modes must admit weakening)
43
44  type A4 = aff 1
45  type B4 = aff 1
46
47  let eg4(x : aff A4 * B4) : aff &{left : A4, right : B4} =
48      case self (
49          left<p1>  => <y, z> <- recv x;
50                              drop z;
51                              fwd p1 y
52          | right<p2> => <y, z> <- recv x;
53                              drop y;
54                              fwd p2 z
55      )
56
57  // Example 5
58  // ↓ₖᵐ (A5 -* B5) ⊢ ↓ₖᵐ (A5) -* ↓ₖᵐ B5 (taking mode k as linear, and m as multicast)
59
60  type A5 = mul 1
61  type B5 = mul 1 * 1
62
63  let eg5(f : mul \/ lin (A5 -* B5)) : (mul \/ lin A5) -* (mul \/ lin B5) =
64      <x, y> <- recv self;
65      w <- shift f;
66      v <- shift x;
67      z : B5 <- new send w<v, self>;
68      cast y<z>
```

```
69
70  // Example 6
71  // +{left: A6, right: B6} -* C6 ⊢ &{left: A6 -* C6, right: B6 -* C6}
72
73  type A6 = 1
74  type B6 = 1
75  type C6 = 1
76
77  type xType = +{left : A6, right : B6} -* C6
78  type resType = &{left : A6 -* C6, right : B6 -* C6}
79  let eg6(x : xType) : resType =
80      case self (
81              left<ac>  => <a, c> <- recv self;
82                            ab : +{left : A6, right : B6} <- new self.left<a>;
83                            send x<ab, self>
84          | right<bc> => <b, c> <- recv self;
85                            ab : +{left : A6, right : B6} <- new self.right<b>;
86                            send x<ab, self>
87      )
88
89  // Example 6 (reverse direction)
90  // &{left: A6' -* C6', right: B6' -* C6'} ⊢ +{left: A6', right: B6'} -* C6'
91
92  type A6' = 1
93  type B6' = 1
94  type C6' = 1
95
96  type yType' = &{left : A6' -* C6', right : B6' -* C6'}
97  type resType' = +{left : A6', right : B6'} -* C6'
98  let eg6reverse(y : yType') : resType' =
99      <ab, c> <- recv self;
100     case ab (
101             left<a>  => ac : A6' -* C6' <- new y.left<self>;
102                           send ac<a, c>
103         | right<b> => bc : B6' -* C6' <- new y.right<self>;
104                           send bc<b, c>
105     )
106
107 // Example 7 and 8 revisit previous examples
108
109 // Example 9 (Circuits)
110 // bits, bits ⊢ bits
111
112 type bits = +{b0 : bits, b1 : bits}
113 let nor(x : bits, y : bits) : bits =
114     case x (
115             b0<x'> => case y (
116                             b0<y'> => z' <- new nor(x', y');
117                                       self.b1<z'>
118
119                         | b1<y'> => z' <- new nor(x', y');
120                                       self.b0<z'>
121                       )
122         | b1<x'> => case y (
123                             b0<y'> => z' <- new nor(x', y');
124                                       self.b0<z'>
125                         | b1<y'> => z' <- new nor(x', y');
126                                       self.b0<z'>
127                       )
128     )
129
130 let or(x : bits, y : bits) : bits =
131     w <- new nor(x, y);
132     <u, u'> <- split w;
133     nor(u, u')
134
135 // Example 10 refers to list mapping, discussed earlier
```

Extended Banking Example. The last example that we consider is a modified version of the banking example from sec. 3.

In this case, the services offered by a hypothetical bank are formalised by the *linear* type `bankType` (lines 1 to 2 below). Two choices are initially offered: a `secure` option to perform a transaction (details omitted) and another one for an `unsecure` option. An `unsecure` request forces the interaction to shift into affine mode ($\downarrow_l^a$ `gen_query`), offering a less restricted interaction mode, since this mode only replies to some general queries (`gen_query`).

```
1   type bankType    = lin &{ secure : ...,
2                              unsecure : aff \/ lin gen_query }
3   type gen_query   = aff +{some_query : 1}
```

The behaviour dictated by the new type `bankType` is implemented below. The `bank` process waits to receive either a `secure` or `unsecure` label (line 5). An unsecure request (line 8) is handled by shifting into affine mode (line 9), before handling the general query (line 8).

```
4   let bank() : bankType =
5       case self (
6           secure<s>     => ...
7         | unsecure<s>   => s' : aff 1 <- new close self;
8                            s'' : gen_query <- new self.some_query<s'>;
9                            cast self<s''>
10      )
```

The execution launched below models a user (`user`, line 11), who initiates a linear interaction but promptly cancels it (line 16); this is permitted by the `bankService`'s affine mode after performing the shift (line 15).

```
11  prc[bankService] : bankType = bank()
12  prc[user] : lin 1 =
13      print _unsecure_connection_drop_;
14      b : aff \/ lin gen_query <- new bankService.unsecure<self>;
15      b' <- shift b; // b' is now affine
16      drop b';
17      close self
```

B   Type System

Figs. 3 and 4 are the complete typing rules used by our type-checker, GRITS.

$$\frac{\Gamma \succeq m \succeq n \quad \Gamma \vdash P :: (x : A^m) \quad \Gamma', x : A^m \vdash Q :: (w : B^n)}{\Gamma, \Gamma' \vdash x \leftarrow \mathsf{new}\ P; Q :: (w : B^n)}\ \mathrm{CUT}$$

$$\frac{m \in \{\mathsf{a},\mathsf{r}\} \quad \Gamma \vdash P :: (w:B^n)}{\Gamma, u{:}A^m \vdash \mathsf{drop}\ u; P :: (w:B^n)}\ \mathrm{DRP} \qquad \frac{m \in \{\mathsf{m},\mathsf{r}\} \quad \Gamma, x{:}A^m, y{:}A^m \vdash P :: (w:B^n)}{\Gamma, u{:}A^m \vdash \langle x,y \rangle \leftarrow \mathsf{split}\ u; P :: (w:B^n)}\ \mathrm{SPL}$$

$$\frac{}{u : A^m \vdash \mathsf{fwd}\ w\ u :: (w : A^m)}\ \mathrm{ID} \qquad \frac{\Sigma(p) = \overline{y : A^m} \vdash P :: (x : B^n)}{u : A^m \vdash p(w, \overline{u}) :: (w : B^n)}\ \mathrm{CALL}$$

Fig. 3: Type Rules (for processes with a copy semantics)

$$\frac{}{\cdot \vdash \mathsf{close}\ w :: (w : \mathbf{1}^m)}\ 1\mathrm{R} \qquad \frac{\Gamma \vdash P :: (w : A^n)}{\Gamma, u : \mathbf{1}^m \vdash \mathsf{wait}\ u; P :: (w : A^n)}\ 1\mathrm{L}$$

$$\frac{}{u : A^m, v : B^m \vdash \mathsf{send}\ w\langle u,v \rangle :: (w : A^m \otimes B^m)}\ \otimes\mathrm{R} \qquad \frac{\Gamma, x : A^m, y : B^m \vdash P :: (w : C^n)}{\Gamma, u : A^m \otimes B^m \vdash \langle x,y \rangle \leftarrow \mathsf{recv}\ u; P :: (w : C^n)}\ \otimes\mathrm{L}$$

$$\frac{\Gamma, x : A^m \vdash P :: (y : B^m)}{\Gamma \vdash \langle x,y \rangle \leftarrow \mathsf{recv}\ w; P :: (w : A^m \multimap B^m)}\ \multimap\mathrm{R} \qquad \frac{}{u : A^m, w : A^m \multimap B^m \vdash \mathsf{send}\ w\langle u,v \rangle :: (v : B^m)}\ \multimap\mathrm{L}$$

$$\frac{l \in L}{u : A_l^m \vdash w.l\langle u \rangle :: (w : \oplus\{l : A_l\}_{l \in L}^m)}\ \oplus\mathrm{R} \qquad \frac{\Gamma, y_l : A_l^m \vdash P_l :: (w : B^n) \quad \text{for each } l \in L}{\Gamma, u : \oplus\{l : A_l\}_{l \in L}^m \vdash \mathsf{case}\ u\ (l\langle y_l \rangle \Rightarrow P_l)_{l \in L} :: (w : B^n)}\ \oplus\mathrm{L}$$

$$\frac{\Gamma \vdash P_l :: (y_l : A_l^m) \quad \text{for each } l \in L}{\Gamma \vdash \mathsf{case}\ w\ (l\langle y_l \rangle \Rightarrow P_l)_{l \in L} :: (w : \&\{l : A_l\}_{l \in L}^m)}\ \&\mathrm{R} \qquad \frac{l \in L}{u : \&\{l : A_l\}_{l \in L}^m \vdash u.l\langle w \rangle :: (w : A_l^m)}\ \&\mathrm{L}$$

$$\frac{\Gamma \vdash P :: (y : A^n)}{\Gamma \vdash y \leftarrow \mathsf{shift}\ w; P :: (w :\uparrow_n^m A^n)}\ \uparrow\mathrm{R} \qquad \frac{}{u :\uparrow_n^m A^n \vdash \mathsf{cast}\ u\langle w \rangle :: (w : A^n)}\ \uparrow\mathrm{L}$$

$$\frac{}{u : A^m \vdash \mathsf{cast}\ w\langle u \rangle :: (w :\downarrow_n^m A^m)}\ \downarrow\mathrm{R} \qquad \frac{\Gamma, x : A^m \vdash P :: (w : B^o)}{\Gamma, u :\downarrow_n^m A^m \vdash x \leftarrow \mathsf{shift}\ u; P :: (w : B^o)}\ \downarrow\mathrm{L}$$

Fig. 4: Type Rules

C   Dynamics

For completeness, we list all operational semantic rules. Fig. 5 supplements the dynamic rules introduced in sec. 2. Fig. 6 lists the forwarding reductions used in our asynchronous interpreter (sec. 4).

$$\text{Dup}\quad \mathsf{prc}(\{a,b\};\iota;P) \longrightarrow \begin{array}{l}\mathsf{prc}(\{a\};\iota;P\sigma_1),\ \mathsf{prc}(\{b\};\iota;P\sigma_2),\\ \big\{\mathsf{prc}(\{c\sigma_1,c\sigma_2\};\iota;\mathsf{fwd}\ \iota\ c)\big\}_{c\in\mathbf{fn}(P)\setminus\{\iota\}}\end{array}$$
$$\text{where } P\neq\mathsf{fwd}\ \_\ \_ \text{ and } \mathbf{rename}(\mathbf{fn}(P)\setminus\{\iota\})=\langle\sigma_1,\sigma_2\rangle$$

$$\text{Fwd}\quad \mathsf{prc}(\{b\};\gamma;P),\ \mathsf{prc}(N;\iota;\mathsf{fwd}\ \iota\ b),\ \longrightarrow\ \mathsf{prc}(N;\gamma;P)$$

$$\text{Cut}\quad \mathsf{prc}(\{a\};\iota;x\leftarrow\mathsf{new}\ P;Q)\ \longrightarrow\ \mathsf{prc}(\{b\};\gamma;P[\gamma/x]),\ \mathsf{prc}(\{a\};\iota;Q[b/x])$$

$$\text{Drp}\quad \mathsf{prc}(\{a\};\gamma;\mathsf{drop}\ b;Q)\ \longrightarrow\ \mathsf{prc}(\emptyset;\iota;\mathsf{fwd}\ \iota\ b),\ \mathsf{prc}(\{a\};\gamma;Q)$$

$$\text{GrC}\quad \mathsf{prc}(\emptyset;\gamma;P),\ \longrightarrow\ \mathsf{prc}(\emptyset;\gamma;\mathsf{fwd}\ \gamma\ a)_{a\in\mathbf{fn}(P)\setminus\{\gamma\}}\qquad \text{where } P\neq\mathsf{fwd}\ \_\ \_$$

$$\text{Spl}\quad \mathsf{prc}(\{a\};\iota;\langle x,y\rangle\leftarrow\mathsf{split}\ b;Q)\longrightarrow\ \mathsf{prc}(\{c,d\};\gamma;\mathsf{fwd}\ \gamma\ b),\ \mathsf{prc}(\{a\};\iota;Q[c,d/x,y])$$

$$\text{Call}\quad \mathsf{prc}(N;\iota;p(\overline{a},\iota))\ \longrightarrow\ \mathsf{prc}(N;\iota;P[\iota,\overline{a}/x,\overline{y}])\qquad \text{where } \Sigma(p)=\overline{y:A^m}\vdash P::(x:B^n)$$

$$\text{Snd}\quad \mathsf{prc}(\{b\};\iota;\mathsf{send}\ \iota\langle c,d\rangle),\ \mathsf{prc}(\{a\};\gamma;\langle x,y\rangle\leftarrow\mathsf{recv}\ b;P)\ \longrightarrow\ \mathsf{prc}(\{a\};\gamma;P[c,d/x,y])$$

$$\text{Rcv}\quad \mathsf{prc}(\{b\};\iota;\langle x,y\rangle\leftarrow\mathsf{recv}\ \iota;P),\ \mathsf{prc}(\{a\};\gamma;\mathsf{send}\ b\langle c,\gamma\rangle)\ \longrightarrow\ \mathsf{prc}(\{a\};\gamma;P[c,\gamma/x,y])$$

$$\text{Sel}\quad \mathsf{prc}(\{b\};\iota;\iota.k\langle c\rangle),\ \mathsf{prc}(\{a\};\gamma;\mathsf{case}\,b(l\langle y_l\rangle\Rightarrow P_l)_{l\in L})\ \longrightarrow\ \mathsf{prc}(\{a\};\gamma;P_k[c/y_k])\quad \text{where } k\in L$$

$$\text{Bra}\quad \mathsf{prc}(\{b\};\iota;\mathsf{case}\,\iota(l\langle y_l\rangle\Rightarrow P_l)_{l\in L}),\ \mathsf{prc}(\{a\};\gamma;b.k\langle\gamma\rangle)\ \longrightarrow\ \mathsf{prc}(\{a\};\gamma;P_k[\gamma/y_k])\quad \text{where } k\in L$$

$$\text{Cst}\quad \mathsf{prc}(\{b\};\iota;\mathsf{cast}\ \iota\langle c\rangle),\ \mathsf{prc}(\{a\};\gamma;y\leftarrow\mathsf{shift}\ b;P)\ \longrightarrow\ \mathsf{prc}(\{a\};\gamma;P[c/y])$$

$$\text{Shf}\quad \mathsf{prc}(\{b\};\iota;y\leftarrow\mathsf{shift}\ \iota;P),\ \mathsf{prc}(\{a\};\gamma;\mathsf{cast}\ b\langle\gamma\rangle)\ \longrightarrow\ \mathsf{prc}(\{a\};\gamma;P[\gamma/y])$$

$$\text{Cls}\quad \mathsf{prc}(\{b\};\iota;\mathsf{close}\ \iota),\ \mathsf{prc}(\{a\};\gamma;\mathsf{wait}\ b;P)\ \longrightarrow\ \mathsf{prc}(\{a\};\gamma;P)$$

Fig. 5: Operational semantic rules

$$\text{Fwd}_\text{P}\qquad \mathsf{prc}\big(b;\gamma;P^+\big),\ \mathsf{prc}\big(N;\iota;\mathsf{fwd}^\text{P}\ \iota\ b\big),\ \longrightarrow\ \mathsf{prc}\big(N;\gamma;P^+\big)$$

$$\text{Fwd}_\text{A}\qquad \mathsf{prc}\big(b;\gamma;P^-\big),\ \mathsf{prc}\big(N;\iota;\mathsf{fwd}^\text{A}\ \iota\ b\big),\ \longrightarrow\ \mathsf{prc}\big(N;\gamma;P^-\big)$$

Fig. 6: Rules Fwd$_\text{A/P}$ replace Fwd from fig. 5

D   Language Grammar of Programs Accepted by GRITS

```
<prog> ::= <statement>*

<statement> ::= type <label> = <type>                    // labelled session type
              | let <label> ( [<param>] ) : <type> = <term>
                                                         // function declaration
              | assuming <param>                         // add name type assumptions
              | prc '[' <name> ']' : <type> = <term>     // create processes
              | exec <label> ()                          // execute function

<param> ::= <name> : <type> [ , <param> ]                // typed variable names

<type> ::= [<modality>] <type_i>                         // session type with
                                                         //   optional modality

<type_i> ::= <label>                                     // session type label
           | 1                                           // unit type
           | + { <branch_type> }                         // internal choice
           | & { <branch_type> }                         // external choice
           | <type_i> * <type_i>                         // send
           | <type_i> -* <type_i>                        // receive
           | <modality> /\ <modality> <type_i>           // upshift
           | <modality> \/ <modality> <type_i>           // downshift
           | ( <type_i> )

<branch_type> ::= <label> : <type_i> [ , <branch_type> ]
                                                         // labelled branches

<modality> ::= r | rep | replicable                      // replicable mode
             | m | mul | multicast                       // multicast mode
             | a | aff | affine                          // affine mode
             | l | lin | linear                          // linear mode

<term> ::= send <name> '<' <name> , <name> '>'           // send names
         | '<' <name> , <name> '>' <- recv <name> ; <term>
                                                         // receive names
         | <name> . <label> '<' <name> '>'               // send label
         | case <name> ( <branches> )                    // receive label
         | <name> [ : <type> ] <- new <term>; <term>     // spawn new process
         | <label> ( [<names>] )                         // function call
         | fwd <name> <name>                             // forward name
         | '<' <name> , <name> '>' <- split <name> ; <term> // split name
         | close <name>                                  // close name
         | wait <name> ; term                            // wait for name to close
         | cast <name> '<' <name> '>'                    // send shift
         | <name> <- shift <name> ; <term>               // receive shift
         | ( <term> )

<branches> ::= <label> '<' <name> '>' => <term> [ '|' <branches> ]
                                                         // term branches

<names> ::= <name> [ ',' <names> ]                       // list of names

<name> ::= 'self'                                        // provider channel[s]
         | <channel_name>                                // channel name
         | <polarity> <channel_name>                     // channel with explicit polarity

<polarity> ::= +                                         // positive polarity
             | -                                         // negative polarity

Others: - Whitespace is ignored
        - <label> is an alpha-numeric combination, typically used to represent
          a choice option
        - // Single line comments
        - /* and multi line comments */
```