

On Separation, Session Types and Algebra

(Draft Paper. January 13, 2011)

Akbar Hussain, Peter W. O’Hearn, and Rasmus L. Petersen

Queen Mary University of London

Abstract. This paper explores the relation between two formalisms and one algebraic framework for concurrency. Session Types and Concurrent Separation Logic are formalisms that support independent reasoning about concurrent processes. We first translate a small language we call Baby Session Types (BST), into a ‘basic’ version of Concurrent Separation Logic (BCSL), and we show that the translation is sound and complete. We then describe a model for BCSL (hence, BST), which exhibits some of the structure of a Concurrent Kleene Algebra, an algebra where operators for parallel and sequential composition are linked by an exchange law. The model construction is general, starting from any ordered commutative monoid of propositions. However, an instantiation where Session Type contexts are the propositions provides a natural model of and concrete meaning to what have recently been termed ‘Plotkin triples’ in the algebra models, corresponding to the intuition of a typing (assertion) as providing an over-approximation of the future.

1 Introduction

This paper explores the relation between several ideas that have been studied of late in the theory of concurrency.

Session Types (ST) is a formalism (or collection of formalisms) for statically typing properties of concurrent processes that communicate via message passing [9, 21, 18]. Concurrent Separation Logic (CSL) is a formalism for reasoning about concurrent processes that interact via shared memory [13, 3]. Many researchers have noticed an intuitive similarity between the formalisms: Each achieves independent reasoning about processes by controlling the usage of resources (be they message channels or heap memory).

A number of works advance ideas related to both CSL and ST in modular reasoning about message-passing programs [19, 12, 8, 4]. We mention also a significant line of work on tpestate checking; see, e.g., [1] and its references. But as far as we know no formal linkage between CSL and ST has previously been established. A difficulty in doing so is that CSL and ST are far apart in the languages they typically use to advance their ideas. To ease comparison, we use a stripped-down version of Session Types which we call Baby Session Types (BST). BST retains the ability to pass endpoints of channels in messages and it uses the resource-oriented typing characteristic of Session Types, but it removes some of the pi-calculus surroundings of Session Types

formalisms (e.g., replication). We also utilize a simultaneously restricted and generalized form of Concurrent Separation Logic, which we call Basic Concurrent Separation Logic (BCSL). BCSL retains the concurrency proof rule of CSL while avoiding such constructs as critical regions or semaphores, and it treats preconditions and postconditions in a general way which does not require that they be interpreted semantically as in typical models of Separation Logic (e.g., as elements of a boolean residuated commutative monoid).

We are then able to form an instantiation of BCSL in which the preconditions and postconditions are typing judgements from BST. This then makes it possible to relate the proof-theoretic power of the two formalisms, and we do so by translating BST into BCSL. The translation is sound and complete, in that typings are sent to Hoare triples in a way that preserves and reflects provability. The translation in this paper goes from BST to BCSL, and not the other way round, for the simple reason that Session Types do not contain postconditions: they are in a sense a formalism of continuations (hence, precondition-only, as in the ‘Hoare doubles’ used in logic for continuations [17]). If we were to retain expressive-enough structure in BST, then we might be able to effect a translation in the opposite direction. This is a question we leave unanswered.

Next we consider the algebraic structure underlying BST and BCSL, taking our lead from the recent Concurrent Kleene Algebra (CKA, [7]). The authors there show that the rules of Basic CSL can be derived from the structure of a CKA. We want to show here something of a converse: instead of starting from a CKA and deriving the CSL rules, we start from CSL and derive part of the CKA structure. That we only get part, and not the whole, structure will, we hope, help to inform ongoing investigations we are undertaking with the authors of [7] on algebraic models for concurrency.

The way we obtain our model is using predicate transformers. We define a general model of BCSL, starting with only an ordered commutative monoid of propositions, where the order models entailment. The predicates are obtained from the model-theoretic structure of (intuitionistic) Bunched Logic [14, 16]. There is also a completeness result linking the semantics back to the proof theory. By composition, this also gives us a model of BST.

The predicate transformer model works for arbitrary ordered commutative monoids, and thus is not limited to Session Types. But, the specific work on BST pays off in giving us a concrete model. In particular, it provides a natural way of thinking about pre/post specs in terms of a relationship $P \sqsupseteq C; Q$ in the algebra, where we think of P as describing an over-approximation of the future.

2 Baby Session Types (BST)

Programs. A BST program is a parallel composition of processes that communicate by exchanging messages.

$$P ::= k?j.P \mid k!j.P \mid P \parallel P \mid \text{inact}$$

The form $k?j.P$ binds variable j in the continuation P , while $k!j.P$ performs no binding. Following standard practice, we will work with terms up to alpha-conversion of bound variables. As a consequence, we tacitly assume that bound variables are fresh when stating proof rules.

Types. Metavariables α and β range over types, given by the following grammar.

$$\alpha, \beta ::= ![\alpha]; \beta \mid ?[\alpha]; \beta \mid \text{end}$$

Here, $![\alpha]; \beta$ describes a channel on which you can send an α , after which the channel behaves as described by β . Similarly, $?[\alpha]; \beta$ means ‘receive α , then β ’.

The co-type $\bar{\alpha}$ is obtained by switching $!$ and $?$ at the outermost level (not within $[\alpha]$), and by setting $\overline{\text{end}} = \text{end}$.

$$\overline{![\alpha]; \beta} = ?[\alpha]; \bar{\beta} \quad \overline{?[\alpha]; \beta} = ![\alpha]; \bar{\beta} \quad \overline{\text{end}} = \text{end}$$

Typing Contexts. The metavariable Δ ranges over finite multisets of variable/type pairs. We say that a context Δ is *consistent* if any channel occurs at most twice, and when it occurs twice the occurrences must be co-types of one another. The consistent contexts are those that require channel-ends to be used in a consistent manner, as two end-points in a point-to-point communication without races. A typing context is called *completed* if end is the only type that appears within it. The metavariable Φ is often used to range over completed contexts. $\Delta_1 \circ \Delta_2$ is multiset union. We write $\Delta \asymp \Delta'$ to mean that $\Delta_1 \circ \Delta_2$ is consistent.

We will use the following notion of entailment between contexts:

$$\Delta_1 \vdash \Delta_2 \text{ iff } \Delta_1 \text{ is inconsistent or } \exists \Phi. \Delta_1 = \Delta_2 \circ \Phi.$$

This entailment encapsulates the idea that an inconsistency implies anything, and that we can forget about completed contexts (or we can add them, as preconditions). It is used in the Consequence rule below.

Typing Rules. The main sequent of the typing system has the form:

$$P \triangleright \Delta$$

which says that process P has typing Δ .

RULES FOR BABY SESSION TYPES

$$\begin{array}{c} \text{[Consequence]} \frac{\Delta_1 \vdash \Delta_2 \quad P \triangleright \Delta_2}{P \triangleright \Delta_1} \\ \\ \text{[Inact]} \frac{}{\text{inact} \triangleright \emptyset} \qquad \text{[Par]} \frac{P_1 \triangleright \Delta_1 \quad P_2 \triangleright \Delta_2}{P_1 \parallel P_2 \triangleright \Delta_1 \circ \Delta_2} \\ \\ \text{[Receive]} \frac{P \triangleright \Delta \circ k: \beta \circ j: \alpha}{k?j.P \triangleright \Delta \circ k: ?[\alpha]; \bar{\beta}} \qquad \text{[Send]} \frac{P \triangleright \Delta \circ k: \beta}{k!j.P \triangleright \Delta \circ k: ![\alpha]; \bar{\beta} \circ j: \alpha} \end{array}$$

We are taking a minimalist approach in this paper, in which we strip as much surrounding detail from each of ST and CSL as possible, leaving simple (even simplistic) basic notions that can be compared technically. We want to focus on the resource control exemplified by the concurrency rules above, and not other language features (e.g., allocation, procedures, locks, pattern matching...) that might be important for practical

language expressiveness but that are relatively independent of our desire to understand the core similarities and differences.

The formulation of BST above takes its lead from the recent [18], but what is left out of the language is almost as notable as what is included. We have avoided recursion, thread and session creation, communication of non-channel data values (e.g., integers), and the variants of record matching and selection often used in Session Type systems. Also, because there are no ‘non-linear’ channels, we cannot typecheck a program in which two or more processes compete for a service (unless the typing context is inconsistent). In short, we have left out many of the things that might be useful in a useful language. What is left over includes mobility of channel ends, ownership transfer, race avoidance, and independent reasoning about processes, some of the more original (and challenging to model) aspects of Session Types, and what we want to concentrate on.

A few further contextual remarks on different formulations of Session Types are in order. In some works, the composition $(k : \alpha) \circ (k : \bar{\alpha})$ is defined to be $k : \perp$, where \perp is a special additional type that represents hidden communication. The treatment in [18] instead unions the typings together, choosing to do hiding in the types in a way that matches binding-time in the programs rather than matching parallel composition. We follow [18] in using unioning and avoiding the \perp type, but we do not have an outermost binding construct (written $\nu xy.P$ in [18]) which establishes two ends of a channel, so we simply allow a channel variable to appear twice. We make these remarks just to be clear in that BST is not intended to be in any way original, and is just following ideas in some of the formulations of Session Types.

As a very basic example of race avoidance, notice that it is not possible to find any *consistent* typing *raceType* such that

$$(k!x.inact) \parallel (k!y.inact) \triangleright \text{raceType}$$

is a derivable typing. The reason is that, to type this, in the Par rule we would need to assign the ability to send on k to both processes, and this would make $\Delta_1 \circ \Delta_2$ inconsistent in the rule.

On the other hand, the ability to send or receive on a channel is not stationary, once and for all lying within a particular process. Here is a simple example where processes exchange these abilities.

$$\begin{array}{c} k!z.k?y.inact \quad \parallel \quad k?x.k!x.inact \\ \triangleright \\ z : \text{end} \quad , \quad k : ![end]; ?[end]; \text{end} \quad , \quad k : ?[end]; ![end]; \text{end} \end{array}$$

Here, the two occurrences of k in the typing context are given types dual to one another.

A slightly more intricate example shows channel permissions flowing amongst processes. We depict the information used in typing the processes as intermediate assertions in the code. This is intended to be suggestive of how Hoare logic proof outlines are often given, and of the way that proofs of concurrent processes have been depicted in CSL where sequential proofs are done independently for the processes.

$$\text{Let } H = ![\alpha]; \text{end}$$

$$\begin{array}{c}
\{h : H, h' : \overline{H}, \\
k : ![H]; \text{end}, j : ![\overline{H}]; \text{end}\} \\
k!h \\
\{h' : \overline{H}, k : \text{end}, \\
j : ![\overline{H}]; \text{end}\} \\
j!h' \\
\{k : \text{end}, j : \text{end}\} \\
P1
\end{array}
\parallel\parallel
\begin{array}{c}
\{k : ?[H]; \text{end}, w : \text{end}\} \\
k?(x) \\
\{x : H, k : \text{end}, w : \text{end}\} \\
x!w \\
\{k : \text{end}, x : \text{end}\} \\
P2
\end{array}
\parallel\parallel
\begin{array}{c}
\{j : ?[\overline{H}]; \text{end}\} \\
j?(y) \\
\{y : \overline{H}, j : \text{end}\} \\
y?z \\
\{j : \text{end}, y : \text{end}, \\
z : \text{end}\} \\
P3
\end{array}$$

You can think of the intermediate typing contexts in the code above as describing the permissions (i.e the channels) that each process holds or owns at any point of execution. The k 's in $P1$ and $P2$ have cotype of each other, indicating that the two ends of channels can communicate privately without external interference. The same goes for the j s in $P1$ and $P3$.

$P1$ completes before $P2$ and $P3$ because the later two are waiting to acquire the relevant permissions. One can see how the ownership of permissions is transferred using the $k!j$ and $k?j$ commands, where each time a $k!$ is executed, the channel that is passed is removed from the post-condition. The opposite is true when $k?$ is executed, where the received channel is then added to the post-condition. Once $P1$ completes both $P2$ and $P3$ have completed their first step of receiving a channel from $P1$ and are then allowed to transfer channels between each other.

Aside: On Inconsistent Contexts. The inclusion of inconsistent as well as consistent typing contexts might seem at first strange. In an early version of this work, we attempted a formulation of BST using consistent contexts only, but a technical difficulty (pointed out by Vasco Vasconcelos) arose. Consider the judgement

$$k!j.\text{inact} \parallel k?x.x!z.j?m.\text{inact} \triangleright k:![\alpha]; \text{end}, k:?\alpha; \text{end}, j:\alpha, j:\overline{\alpha}, z:\text{end}$$

where $\alpha = ![end]; \text{end}$.

This judgement is typable in BST using a derivation that uses consistent contexts at every step. However, if we take one step of execution, we obtain the contractum

$$j!z.j?m.\text{inact}$$

and subject reduction would dictate that it should be typable as

$$j!z.j?m.\text{inact} \triangleright j:\alpha, j:\overline{\alpha}, z:\text{end}$$

which it is, and this context is consistent, but the derivation uses an inconsistent context at the step

$$j?m.\text{inact} \triangleright j:\text{end}, j:\overline{\alpha}, z:\text{end}.$$

We will study subject reduction formally in the next section but this illustrates that if we leave the inconsistent contexts out, we will obtain undesirable properties.

It is because we have included inconsistent contexts that we included the Consequence rule. Often in session type formalisms, one uses a rule

$$[\text{Inact}] \frac{\Phi \text{ is completed}}{\text{inact} \triangleright \Phi}$$

and has that Weakening of completed contexts Φ

$$\frac{P \triangleright \Delta}{P \triangleright \Delta \circ \Phi}$$

is an admissible rule. We want Weakening to take into account inconsistency as well as completedness, and it was simpler to just include the Consequence rule explicitly to do this.

This discussion of inconsistent contexts is one instance of the many subtleties in formalisms for session types, as discussed in particular in [21].

3 Operational Semantics and Subject Reduction

In light of the subtleties noted in the last section we include here an account of subject reduction. This section is basic sanity-checking material, and can be omitted without loss of continuity.

Structural congruence on programs is the transitive, symmetric relation generated by the following three rules:

$$P \parallel \text{inact} \cong P \quad P \parallel Q \cong Q \parallel P \quad (P \parallel Q) \parallel R \cong P \parallel (Q \parallel R)$$

and reduction is the relation generated by the following three rules:

$$\begin{array}{l} k!j.P \parallel k?j'.Q \xrightarrow{k} P \parallel Q[j/j'] \quad [Com(k)] \\ P \xrightarrow{k} P' \implies P \parallel Q \xrightarrow{k} P' \parallel Q \quad [Par] \\ Q \cong Q' \wedge Q' \xrightarrow{k} P' \wedge P' \cong P \implies Q \xrightarrow{k} P \quad [Str] \end{array}$$

Here, $Q[j/j']$ denotes the result of substituting j for j' in Q . In the substitution lemma below we use a similar notation $\Delta[j/j']$ for substitution into a typing context.

Our treatment of subject reduction will involve changing the types as we change the program.

Definition 1 *If Δ is consistent and $\Delta = \Delta' \circ k : ![\alpha]; \beta \circ k : ?[\alpha]; \bar{\beta}$ then we define*

$$\Delta/k = \Delta' \circ k : \beta \circ k : \bar{\beta}$$

otherwise the symbol is undefined.

We have lemmas which account for the structure of (consistent) typings for sending and receiving.

Lemma 2 *If $k!j.P \triangleright \Delta$, with Δ consistent, then there exists α , β and Δ' such that $\Delta = \Delta' \circ k : ![\alpha]; \beta \circ j : \alpha$ and $P \triangleright \Delta' \circ k : \beta$.*

Proof: Obviously, the equality is modulo reordering of Δ . The proof is by induction of the typing derivation, only two cases are relevant:

Case [Consequence]: As Δ is consistent then $\Delta = \Delta_2 \circ \Phi$ and so Δ_2 is also consistent. Then, by induction, there is α, β and Δ'' such that $\Delta_2 = \Delta'' \circ k : ![\alpha]; \beta \circ j : \alpha$ and $P \triangleright \Delta'' \circ k : \beta$. Now, by the rule of consequence, α, β and $\Delta' = \Delta'' \circ \Phi$ have the required properties.

Case [Send]: Follows directly from the rule.

■

Corollary 3 *If $k!j.P \triangleright \Delta$, with Δ consistent, then there exists α, β and Δ' such that $\Delta = \Delta' \circ k : ![\alpha]; \beta$.*

Lemma 4 *If $k?j.P \triangleright \Delta$, with Δ consistent, then there exists α, β and Δ' with j not free in Δ' , such that $\Delta = \Delta' \circ k : ?[\alpha]; \beta$ and $P \triangleright \Delta' \circ k : \beta \circ j : \alpha$.*

Proof: Obviously, the equality is modulo reordering of Δ . The proof is by induction of the typing derivation, only two cases are relevant:

Case [Consequence]: As Δ is consistent then $\Delta = \Delta_2 \circ \Phi$ and so Δ_2 is also consistent. Then, by induction, there is α, β and Δ'' with j not free in Δ'' , such that $\Delta_2 = \Delta'' \circ k : ![\alpha]; \beta$ and $P \triangleright \Delta'' \circ k : \beta \circ j : \alpha$. Now, by the rule of consequence, α, β and $\Delta' = \Delta'' \circ \Phi$ have the required properties. (If j is free in Φ , then we alpha-rename $k?j.P$ to $k?l.P$ for some fresh l which is then free neither in Δ'' nor in Φ .)

Case [Receive]: Follows directly from the rule.

■

Lemma 5 *If $k!j.P \parallel k?j.Q \triangleright \Delta$, with Δ consistent, then there exists α, β and Δ' such that $\Delta = \Delta' \circ k : ![\alpha]; \beta \circ k : ?[\alpha]; \bar{\beta}$.*

Proof: Obviously, the equality is modulo reordering of Δ . The proof is by induction of the typing derivation, only two cases are relevant:

Case [Consequence]: As Δ is consistent then $\Delta = \Delta_2 \circ \Phi$ and so Δ_2 is also consistent. Then, by induction, there is α, β and Δ'' such that $\Delta_2 = \Delta'' \circ k : ![\alpha]; \beta$. Now α, β and $\Delta' = \Delta'' \circ \Phi$ have the required properties.

Case [Par]: So $\Delta = \Delta_1 \circ \Delta_2$ and if Δ is consistent then so are both Δ_1 and Δ_2 . Thus

- By Corollary 3 there is α_1, β_1 and Δ'_1 such that $\Delta_1 = \Delta'_1 \circ k : ![\alpha_1]; \beta_1$.
- By Lemma 4 there is α_2, β_2 and Δ'_2 such that $\Delta_2 = \Delta'_2 \circ k : ?[\alpha_2]; \beta_2$.

Since $\Delta_1 \times \Delta_2$ we conclude that $\alpha_2 = \alpha_1$ and that $\beta_2 = \bar{\beta}_1$. Then $\alpha = \alpha_1, \beta = \beta_1$ and $\Delta' = \Delta'_1 \circ \Delta'_2$ have the desired properties.

■

Lemma 6 (Substitution) *If $P \triangleright \Delta$ then $P[j/j'] \triangleright \Delta[j/j']$.*

¹ A formulation of the lemma that takes alpha renaming into account would be: If $k?j'.P \triangleright \Delta$, with Δ consistent, then there exists α, β, Δ' and j with j not free in Δ' , such that $\Delta = \Delta' \circ k : ?[\alpha]; \beta$ and $P[j/j'] \triangleright \Delta' \circ k : \beta \circ j : \alpha$.

Proof: Proof is by induction on the derivation of $P \triangleright \Delta \circ j' : \alpha$.

For the rule of consequence we have to note that if Δ is inconsistent then so is $\Delta[j/j']$.

For the receive rule we note that if j is the channel received then it is bound and the substitution has no effect ($?$ is a binder and the substitution is capture avoiding). ■

Lemma 7 (SubjectReduction) *If $P \triangleright \Delta$ with Δ consistent, and $P \xrightarrow{k} P'$ then Δ/k is defined and consistent and $P' \triangleright \Delta/k$.*

Proof: Proof is by case on the rule used to prove $P \xrightarrow{k} P'$, then, if necessary, by induction on the typing derivation for $P \triangleright \Delta$.

Case [Com (k)]: By Lemma 5, Δ/k is defined and since Δ is consistent then so is Δ/k . That $P' \triangleright \Delta/k$ is shown by induction on the derivation for $P \triangleright \Delta$; since $P = k!j.P_1 \parallel k?j'.P_2$, only two cases apply:

Case [Consequence]: If Δ is consistent then $\Delta = \Delta_2 \circ \Phi$ and so Δ_2 is also consistent. Then, by induction, $P' \triangleright \Delta_2/k$. Now, since Δ_2/k is defined and $\Delta_2 \asymp \Phi$, Φ cannot mention k . Thus $\Delta/k = (\Delta_2 \circ \Phi)/k$ is defined and equal to $\Delta_2/k \circ \Phi$. This means that $\Delta/k \vdash \Delta_2/k$ and so, by the rule of consequence, $P' \triangleright \Delta/k$.

Case [Par]: So $\Delta = \Delta_1 \circ \Delta_2$ and if Δ is consistent then so are both Δ_1 and Δ_2 . Thus

- by Lemma 2 there is α_1, β_1 and Δ'_1 such that $\Delta_1 = \Delta'_1 \circ k : ![\alpha_1]; \beta_1 \circ j : \alpha$ and $P_1 \triangleright \Delta'_1 \circ k : \beta_1$.

- by Lemma 4 there is α_2, β_2 and Δ'_2 with j' not free in Δ'_2 , such that $\Delta_2 = \Delta'_2 \circ k : ?[\alpha_2]; \beta_2$ and $P_2 \triangleright \Delta'_2 \circ k : \beta_2 \circ j' : \alpha$.

By Lemma 4, $(\Delta'_2 \circ k : \beta_2 \circ j' : \alpha)[j/j'] = \Delta'_2 \circ k : \beta_2 \circ j : \alpha$ and by Lemma 6, $P_2[j/j'] \triangleright \Delta'_2 \circ k : \beta_2 \circ j : \alpha$ and so the parallel rule gives $P_1 \parallel P_2[j/j'] \triangleright (\Delta_1 \circ \Delta_2)/k$.

Case [Par]: Follows from the fact that $(\Delta_1 \circ \Delta_2)/k = \Delta_1/k \circ \Delta_2$ when Δ_1/k is defined and $\Delta_1 \asymp \Delta_2$.

Case [Str]: Structural congruence preserves typing.

■

4 Basic Concurrent Separation Logic (BCSL)

Where BST is defined by reference to particular language constructs, in formulating BCSL we follow the lead of Abstract Separation Logic [5] and define a system which abstracts away from the exact details of primitive commands or pre/post specs, and later we will instantiate it to be more like BST.

Basic CSL. We presume we are given

- A preordered commutative monoid of propositions $(Props, \vdash, *, emp)$;
- A set Com equipped with total binary operations $c_1 \parallel c_2$ and $c_1; c_2$, and with a distinguished element $skip \in Com$.

In presentations of Separation Logic, propositions are usually taken to have stronger structure. For example, propositions are usually assumed to carry a Boolean algebra structure, and that is not required of BCSL. This extra generality will ease the comparison to Session Types.

PROOF RULES FOR BCSL

$$\begin{array}{c}
\text{[Skip]} \frac{}{\{X\} \text{ skip } \{X\}} \qquad \text{[Frame]} \frac{\{X\} c \{Y\}}{\{X * F\} c \{Y * F\}} \\
\text{[Seq]} \frac{\{X\} c_1 \{Y\} \quad \{Y\} c_2 \{Z\}}{\{X\} c_1; c_2 \{Z\}} \quad \text{[Par]} \frac{\{X_1\} c_1 \{Y_1\} \quad \{X_2\} c_2 \{Y_2\}}{\{X_1 * X_2\} c_1 \parallel c_2 \{Y_1 * Y_2\}} \\
\text{[Consequence]} \frac{X' \vdash X \quad \{X\} c \{Y\} \quad Y \vdash Y'}{\{X'\} c \{Y'\}}
\end{array}$$

We intend that in a specialization of BCSL there may be additional rules and axioms: Those just given form the core, that may be added to.

Heap Model Instantiation. We briefly recap how CSL works with a heap model. The structure of propositions is obtained by setting

$$(Props, \vdash, *, emp) = (P(Heaps), \subseteq, *, \{u\})$$

where the material to the right is defined as follows. *Heaps* is the set $\mathbb{N} \rightarrow_f \mathbb{N}$ of finite partial functions from natural numbers to natural numbers, and $P(Heaps)$ is the powerset. u is the empty partial function, and $X * Y$ is defined as follows. First, let $h \bullet h'$ denote the union of heaps with disjoint domain, which is undefined when the domains overlap. Then, $X * Y = \{h_X \bullet h_Y \mid h_X \in X \wedge h_Y \in Y \wedge h_X \bullet h_Y \downarrow\}$.

A typical predicate is the points-to fact $n \mapsto m$, where n and m are natural numbers. It is the singleton set $\{h\}$ where h is the heap that maps n to m and which is undefined on all numbers other than n . Another predicate is $n \mapsto -$, which is $\bigvee_m n \mapsto m$.

A typical command is the mutation statement $[n] := m$ where n and m are natural numbers. Associated with it is the following Hoare triple axiom.

$$\overline{\{n \mapsto -\}[n] := m \{n \mapsto m\}}$$

Brookes has proven a fundamental result that CSL for heaps cannot prove any racy programs [3]; or more precisely, there can never be a race in a proven program, starting from a state satisfying the precondition. As an example, for the program

$$[10] := 23 \parallel [10] := 44$$

we cannot find any consistent precondition. The reason being that each process needs a precondition $10 \mapsto -$ according to the axiom for $[n] := m$ given above, and $10 \mapsto - * 10 \mapsto -$ is *false*.

The heap model of CSL also allows for the transfer of ownership of heap buffers between processes.

The first example used to show this was a ‘pointer transferring buffer’, where a cell is allocated in one process, placed in a buffer and read out by a second process, and then safely disposed in that second process. In CSL the proof goes like this

$$\begin{array}{c}
\{emp\} \\
\{emp * emp\} \\
\{emp\} \qquad \{emp\} \\
x := \mathbf{cons}(a, b) \parallel \mathit{get}(y) \\
\{x \mapsto -, -\} \qquad \{y \mapsto -, -\} \\
\mathit{put}(x); \qquad \mathit{use}(y); \\
\{emp\} \qquad \{y \mapsto -, -\} \\
\qquad \mathit{dispose}(y); \\
\qquad \{emp\} \\
\{emp * emp\} \\
\{emp\}
\end{array}$$

where we would have to separately verify the pre/post specs for the *put* and *get* operations. Here, the knowledge that x is allocated disappears from the left process in the *put* operation, to be materialized after the *get* in the second process. This is, evidently, somewhat analogous to the example of ownership transfer in the previous section.

We have treated this last example rather informally, but we hope the reader can see from the discussion in this section and the previous one some of the apparent similarities between Session Types and Separation Logic, which led us to be curious as to the actual relationship between them.

5 Session Instantiation of BCSL: BCSL/ST

To instantiate BCSL what we need to do is set down the structure of propositions and the structure of commands. In fact, we have formulated BCSL in a general way so as to make this instantiation immediate.

Propositions. To instantiate BCSL we must first define a preordered commutative monoid $(Props, \vdash, *, emp)$. We simply take $Prop$ to be the set of session typing contexts Δ , and $\Delta * \Delta'$ to be $\Delta \circ \Delta'$. emp is the empty context \emptyset . $X \vdash Y$ is the entailment notion between typing contexts already defined.

Commands. We extend the commands of BCSL with primitives for receiving and sending channels. The former uses a continuation, to account for the binding aspect, but the latter does not use a continuation as we can use sequencing instead.

$$C ::= k?j.C \mid k!j \mid C \parallel C \mid C; C \mid \mathbf{skip}$$

There is an asymmetry here, in that the receiving form $k?j.C$ uses an explicit continuation where the sending form $k!j$ does not. There is no deep reason. We would prefer to do away with all explicit continuations, letting $;$ do the job. But, if we were not to use a binding form, then to treat receiving we should then either include imperative references to store the result or a functional language notation that includes return values.

Either of these would be a fine choice in a language, but to keep our formalism as small as possible we made the compromise to include this binding form (so as to avoid return values or references).

Proof Rules. The proof rules are those of BCSL together with

SPECIALIZED RULES FOR SESSION INSTANTIATION.

$$\begin{array}{c} \text{[Send]} \frac{}{\{k: ![\alpha]; \beta * j: \alpha\} k!j \{k: \beta\}} \\ \text{[Receive]} \frac{\{A * k: \beta * j: \alpha\} P \{B\}}{\{A * k: ?[\alpha]; \beta\} k?j.P \{B\}} \end{array}$$

Note that our tacit assumption that the bound variable j is fresh in the Receive rule means that it is not free in A or B and not equal to k .

In the proof of completeness we will need the following 2 lemmas, the later of which encapsulates the triples that can be proven about the $k!$ construct.

Lemma 8 *If Δ is inconsistent then $P \triangleright \Delta$ holds for all programs P .*

Proof: Assume Δ is inconsistent. Suppose $P \triangleright \Delta'$ and since Δ is inconsistent we know that $\Delta \vdash \Delta'$ then we can apply the *Consequence* rule to get $P \triangleright \Delta$. Thus all we need to prove is that

$$\forall P. \exists \Delta'. P \triangleright \Delta'$$

We show this by induction on the structure of P :

Case $P = k!?.P'$: By induction, there is Δ'' such that $P' \triangleright \Delta''$. By the rule of consequence $P' \triangleright \Delta'' \circ k : \text{end} \circ j : \text{end}$ (even though this context might be inconsistent).

By the rule for receive we then have $k?j.P' \triangleright \Delta'' \circ k : ?[\text{end}]; \text{end}$.

Case $P = k!j.P'$: By induction, there is Δ'' such that $P' \triangleright \Delta''$. By the rule of consequence $P' \triangleright \Delta'' \circ k : \text{end}$ (even though this context might be inconsistent). By the rule for send we then have $k!j.P' \triangleright \Delta'' \circ k : ![\text{end}]; \text{end} \circ j : \text{end}$.

Case $P = P_1 \parallel P_2$: Follows from the rule and induction hypothesis, with $\Delta' = \Delta'_1 \circ \Delta'_2$.

Case $P = \text{inact}$: Follows from the rule with $\Delta' = \emptyset$.

■

Lemma 9 (Send Characterization Lemma) *Assume $\{A\} k!j \{B\}$ is derivable, where A is consistent. Then there is a Δ such that $A = \Delta * k: ![\alpha]; \beta * j: \alpha$ and if Δ does not contain $k?[\alpha]; \bar{\beta}$ then*

- B is consistent
- there is Δ' and Δ_k such that $B = \Delta' * \Delta_k$
- $\beta \neq \text{end} \Rightarrow \Delta_k = k: \beta$ and if $\beta = \text{end}$ then Δ_k is either $k: \beta$ or *emp*
- $\Delta * k: \beta \vdash \Delta' * k: \beta$

Proof: *Case, last step is Send*

$$\overline{\{k: ![\alpha]; \beta * j: \alpha\} k!j \{k: \beta\}}$$

This case is true where $\Delta, \Delta' = emp$.

Case, last step is Frame

$$\frac{\{X\} k!j \{Y\}}{\{X * F\} k!j \{Y * F\}}$$

Since $X * F$ is consistent we know that X is consistent. Then, by induction, we know that X has the form $\Delta_X * k: ![\alpha]; \beta * j: \alpha$, and so, with $\Delta = \Delta_X * F$, we see that $X * F$ does as well.

Now, if $X * F$ does not contain $k?[\alpha]; \bar{\beta}$ then neither does X and so, by induction, we have that

- Y is consistent
- there is Δ'_Y and Δ_k such that $Y = \Delta'_Y * \Delta_k$
- $\beta \neq end \Rightarrow \Delta_k = k: \beta$ and if $\beta = end$ then Δ_k is either $k: \beta$ or emp
- $\Delta_X * k: \beta \vdash \Delta'_Y * k: \beta$

while we need to show that

- $Y * F$ is consistent
- there is Δ' such that $Y * F = \Delta' * \Delta_k$
- $\Delta * k: \beta \vdash \Delta' * k: \beta$

First we argue that $Y * F$ is consistent: Since $X * F = \Delta_X * k: ![\alpha]; \beta * j: \alpha * F$ is consistent and does not contain $k?[\alpha]; \bar{\beta}$ we conclude that $\Delta_X * k: \beta * F$ is consistent.

Since $\Delta_X * k: \beta \vdash \Delta'_Y * k: \beta$, we have $\Delta_X * k: \beta * F \vdash \Delta'_Y * k: \beta * F$ and so $\Delta'_Y * k: \beta * F$ is consistent. This makes $Y * F = \Delta'_Y * \Delta_k * F$ consistent.

Now, with $\Delta' = \Delta'_Y * F$ the two other points are true as well.

Case, last step is Consequence

$$\frac{X' \vdash X \quad \{X\} k!j \{Y\} \quad Y \vdash Y'}{\{X'\} k!j \{Y'\}}$$

Left

$$\frac{X' \vdash X \quad \{X\} k!j \{Y\}}{\{X'\} k!j \{Y\}}$$

Since X' is consistent and we know that $X' \vdash X$ then we know that X is consistent as adding on completed contexts does not make a context inconsistent. Then by induction we know that X has the form $\Delta * k: ![\alpha]; \beta * j: \alpha$. Then we have our desired result by the rule of Consequence and the fact that $X' \vdash \Delta * k: ![\alpha]; \beta * j: \alpha$. Furthermore if X' does not contain $k: [\alpha]; \bar{\beta}$ then nor does X and by induction we know that

- Y is consistent
- there is Δ' and Δ_k such that $Y = \Delta' * \Delta_k$
- $\beta \neq end \Rightarrow \Delta_k = k: \beta$ and if $\beta = end$ then Δ_k is either $k: \beta$ or emp

– $\Delta * k: \beta \vdash \Delta' * k: \beta$

Right

$$\frac{\{X\} k!j \{Y\} \quad Y \vdash Y'}{\{X\} k!j \{Y'\}}$$

Since X is consistent then by induction we know that X has the form $\Delta * k: ![\alpha]; \beta * j: \alpha$. Now if X does not contain $k?[\alpha]; \bar{\beta}$ then we know that

- Y is consistent
- there is Δ' and Δ_k such that $Y = \Delta' * \Delta_k$
- $\beta \neq \text{end} \Rightarrow \Delta_k = k: \beta$ and if $\beta = \text{end}$ then Δ_k is either $k: \beta$ or emp
- $\Delta * k: \beta \vdash \Delta' * k: \beta$

Since $Y \vdash Y'$ and Y is consistent we know that

- Y' is consistent
- there is Δ'_Y and Δ'_k such that $Y' = \Delta'_Y * \Delta'_k$
- $\beta \neq \text{end} \Rightarrow \Delta'_k = k: \beta$ and if $\beta = \text{end}$ then Δ'_k is either $k: \beta$ or emp
- $\Delta * k: \beta \vdash \Delta'_Y * k: \beta$

where the last item follows from the fact that $\Delta' * k: \beta \vdash \Delta'_Y * k: \beta$.

■

Translation. The translation $\langle\langle \cdot \rangle\rangle$ is a straightforward mapping from untyped BST to BCSL programs:

$$\begin{aligned} \langle\langle \text{inact} \rangle\rangle &= \text{skip} \\ \langle\langle P \parallel Q \rangle\rangle &= \langle\langle P \rangle\rangle \parallel \langle\langle Q \rangle\rangle \\ \langle\langle k?j.P \rangle\rangle &= k?j.\langle\langle P \rangle\rangle \\ \langle\langle k!j.P \rangle\rangle &= (k!j); \langle\langle P \rangle\rangle \end{aligned}$$

Theorem 10 (Soundness and Completeness of translation) $P \triangleright \Delta$ in **BST** if and only if $\{\Delta\} \langle\langle P \rangle\rangle \{\text{emp}\}$ in **BCSL/ST**

Proof: FOR THE ‘IF DIRECTION’ (COMPLETENESS), $\{\Delta\} \langle\langle P \rangle\rangle \{\text{emp}\} \Rightarrow P \triangleright \Delta$, it is convenient to prove the equivalent

$$(\exists \Phi \text{ completed. } \{\Delta\} \langle\langle P \rangle\rangle \{\Phi\}) \Rightarrow P \triangleright \Delta.$$

That this is equivalent follows at once from the entailment $\Phi \vdash \text{emp}$ and the Rule of Consequence. The proof requires a careful treatment of the use of $k!$ in Sequencing, but otherwise proceeds by straightforward induction on the derivation of $\{\Delta\} \langle\langle P \rangle\rangle \{\Phi\}$. We give a few cases.

Case, last step is Frame:

$$\frac{\{\Delta\} P \{\Delta'\}}{\{\Delta * F\} P \{\Delta' * F\}}$$

Since we know that $\Delta' * F$ is completed, we can conclude that both Δ' and F are. Since Δ' is completed, the induction hypothesis gives us that $P \triangleright \Delta$. Since F is completed, we know that $\Delta * F \vdash \Delta$, so we can use the Consequence rule to derive the desired $P \triangleright \Delta * F$.

Case, last step is `Seq`: All uses of `;` in the translation $\langle\langle \cdot \rangle\rangle$ have a `Send k!` to the immediate left. So we know the last rule is of the form

$$\frac{\{\Delta_0\} k!j \{\Delta_1\} \quad \{\Delta_1\} \langle\langle P \rangle\rangle \{\Phi\}}{\{\Delta_0\} (k!j); \langle\langle P \rangle\rangle \{\Phi\}}$$

If Δ_0 is inconsistent, we immediately have $(k!j); \langle\langle P \rangle\rangle \triangleright \Delta_0$ by Lemma 8, so we assume Δ_0 consistent. We can then apply Send Characterization Lemma (Lemma 9) to obtain two cases:

Case Δ_0 does not contain $k : ?[\alpha]; \bar{\beta}$: We know that we can take this instance of the rule to be of the form

$$\frac{\{\Delta * k : ![\alpha]; \beta * j : \alpha\} k!j \{\Delta' * \Delta_k\} \quad \{\Delta' * \Delta_k\} \langle\langle P \rangle\rangle \{\Phi\}}{\{\Delta * k : ![\alpha]; \beta * j : \alpha\} (k!j); \langle\langle P \rangle\rangle \{\Phi\}}$$

where Δ_k is either $k : \beta$ or, if $\beta = \text{end}$, possibly emp .

We need to show:

$$k!j . P \triangleright \Delta * k : ![\alpha]; \beta \circ j : \alpha$$

We can show this using the rule

$$\frac{P \triangleright \Delta \circ k : \beta}{k!j . P \triangleright \Delta \circ k : ![\alpha]; \beta \circ j : \alpha}$$

if we have the premise, so this case is reduced to showing $P \triangleright \Delta \circ k : \beta$. The induction hypothesis gives us $P \triangleright \Delta' \circ \Delta_k$. If $\beta = \text{end}$ and $\Delta_k = \text{emp}$, we can use the rule of consequence to obtain $P \triangleright \Delta' \circ k : \beta$ and if not, we have that already.

The Send Characterization Lemma (Lemma 9) tells us that $\Delta * k : \beta \vdash \Delta' * k : \beta$, and so by the rule of consequence we obtain the desired result.

Case Δ_0 does contain $k : ?[\alpha]; \bar{\beta}$: If we write Δ_0 as $\Delta_A * k : ?[\alpha]; \bar{\beta} * k : ![\alpha]; \beta * j : \alpha$, we can use the send rule and then the frame rule to prove

$$\{\Delta_A * k : ?[\alpha]; \bar{\beta} * k : ![\alpha]; \beta * j : \alpha\} k!j \{\Delta_A * k : ?[\alpha]; \bar{\beta} * k : \beta\}$$

Since $\Delta_A * k : ?[\alpha]; \bar{\beta} * k : \beta$ is inconsistent, we have that $P \triangleright \Delta_A \circ k : ?[\alpha]; \bar{\beta} \circ k : \beta$. We can then use the typing rule

$$\frac{P \triangleright \Delta_A \circ k : ?[\alpha]; \bar{\beta} \circ k : \beta}{k!j . P \triangleright \Delta_A \circ k : ?[\alpha]; \bar{\beta} \circ k : ![\alpha]; \beta \circ j : \alpha}$$

to conclude that $P \triangleright \Delta_0$.

Case, last step is `skip`:

$$\overline{\{\Delta\} \text{skip} \{\Delta\}}$$

Since $\Delta = \Phi$ is completed, we have the desired result from the axiom $\text{inact} \triangleright \emptyset$ of BST.

Case, last step is *Par*;

$$\frac{\{\Delta_1\} \langle\langle P_1 \rangle\rangle \{\Phi_1\} \quad \{\Delta_2\} \langle\langle P_2 \rangle\rangle \{\Phi_2\}}{\{\Delta_1 * \Delta_2\} c_1 \parallel c_2 \{\Phi_1 * \Phi_2\}}$$

The desired result follows from induction hypothesis and the rule

$$\frac{P_1 \triangleright \Delta_1 \quad P_2 \triangleright \Delta_2}{P_1 \parallel P_2 \triangleright \Delta_1 \circ \Delta_2}$$

Case, last step is *Rule of Consequence*: We treat the left and right separately. The instance

$$\frac{\Delta' \vdash \Delta \quad \{\Delta\} \langle\langle P \rangle\rangle \{\Phi\}}{\{\Delta'\} \langle\langle P \rangle\rangle \{\Phi\}}$$

The desired result follows from induction hypothesis and the *Consequence* rule. In the other instance

$$\frac{\{\Delta\} \langle\langle P \rangle\rangle \{\Delta'\} \quad \Delta' \vdash \Phi}{\{\Delta\} \langle\langle P \rangle\rangle \{\Phi\}}$$

Δ' must be completed for $\Delta' \vdash \Phi$ to hold, by the def of \vdash . The desired result then follows from induction hypothesis.

Case, last step is *Receive*:

$$\frac{\{\Delta * k: \beta * j: \alpha\} P \{\Delta'\}}{\{\Delta * k: ?[\alpha]; \beta\} k?j.P \{\Delta'\}}$$

We need to show:

$$k?j.P \triangleright \Delta \circ k: ?[\alpha]; \beta$$

By induction hypothesis the premise in the *BST* rule

$$\frac{P \triangleright \Delta \circ k: \beta \circ j: \alpha}{k?j.P \triangleright \Delta \circ k: ?[\alpha]; \beta}$$

holds, and then the result follows from the *Receive* rule.

Case, last step is *Send*: *Send* never applies since the *Send* rule is an axiom which does not mention the translation of any program, since all programs in **BST** have continuations.

THE ‘ONLY IF DIRECTION’ (SOUNDNESS), $P \triangleright \Delta \Rightarrow \{\Delta\} \langle\langle P \rangle\rangle \{emp\}$ follows by showing that all four session type rules are admissible in the translation: the proof is by induction on proofs with a case analysis on the last rule. All cases are straightforward.

Case *inact*:

$$\overline{\text{inact} \triangleright \emptyset}$$

We are required to show $\{\emptyset\} \text{skip} \{emp\}$. Since $\emptyset = emp$ we can use the axiom for *skip* to get the desired result.

Case Par:

$$\frac{P_1 \triangleright \Delta_1 \quad P_2 \triangleright \Delta_2}{P_1 \parallel P_2 \triangleright \Delta_1 \circ \Delta_2}$$

We are required to show $\{\Delta_1 * \Delta_2\} P_1 \parallel P_2 \{emp\}$ This follows from the induction hypothesis and the concurrency rule

$$\frac{\{\Delta_1\} P_1 \{emp\} \quad \{\Delta_2\} P_2 \{emp\}}{\{\Delta_1 * \Delta_2\} P_1 \parallel P_2 \{emp * emp\}}$$

Case Receive:

$$\frac{P \triangleright \Delta \circ k : \beta \circ j : \alpha}{k?j.P \triangleright \Delta \circ k : ?[\alpha]; \beta}$$

We are required to show

$$\{\Delta * k : ?[\alpha]; \beta\} k?j.\langle\langle P \rangle\rangle \{emp\}$$

By induction we get $\{\Delta * k : \beta * j : \alpha\} \langle\langle P \rangle\rangle \{emp\}$ which together with the rule for Receive gives the desired result.

Case Send:

$$\frac{P \triangleright \Delta \circ k : \beta}{k!j.P \triangleright \Delta \circ k : ![\alpha]; \beta \circ j : \alpha}$$

We are required to show

$$\{\Delta * k : ![\alpha]; \beta * j : \alpha\} (k!j); \langle\langle P \rangle\rangle \{emp\}$$

By induction we get $\{\Delta * k : \beta\} \langle\langle P \rangle\rangle \{emp\}$ and the following instance of the rule for sequencing and the Send rule gives us the desired result.

$$\frac{\frac{\{k : ![\alpha]; \beta * j : \alpha\} k!j \{k : \beta\}}{\{\Delta * k : ![\alpha]; \beta * j : \alpha\} k!j \{\Delta * k : \beta\}} \quad \{\Delta * k : \beta\} \langle\langle P \rangle\rangle \{emp\}}{\{\Delta * k : ![\alpha]; \beta * j : \alpha\} (k!j); \langle\langle P \rangle\rangle \{emp\}}$$

Case Consequence:

$$\frac{\Delta_1 \vdash \Delta_2 \quad P \triangleright \Delta_2}{P \triangleright \Delta_1}$$

We are required to show

$$\{\Delta_1\} \langle\langle P \rangle\rangle \{emp\}$$

By induction we get $\{\Delta_2\} \langle\langle P \rangle\rangle \{emp\}$. Since $\Delta_1 \vdash \Delta_2$ we can apply the rule of consequence to get $\{\Delta_1\} \langle\langle P \rangle\rangle \{emp\}$.

■

6 Building a Model

6.1 Preliminary Considerations

In this section we study a model built from BCSL which, hence, gives a model of BST as well. The recent Concurrent Kleene Algebra (CKA) of [7] has influenced the way we have set up this model. We do not need all of CKA, but will utilize the following key ideas.

The first idea is that of two ordered semi-groups, $;$ and \parallel , on the same carrier poset, linked by the exchange law:

$$(F_1 \parallel F_2); (G_1 \parallel G_2) \sqsubseteq (F_1; G_1) \parallel (F_2; G_2)$$

This pleasant law – which is a weakened version of the familiar law from 2-categories or bicategories (weakened by using an ordering rather than an equality) – holds in many situations, such as when \parallel is interleaving and $;$ is concatenation of sequences, in some partial order models, as well as in situations related to separation logic (e.g., [2, 4, 7]).

The second idea is to interpret Hoare triples in terms of special relationships in the algebra. The suggestion in [7] is to interpret the triple $\{F\} C \{G\}$ as $F; C \sqsubseteq G$. Notice that the pre and post F and G are taken from the same algebra as the program C : they are viewed as the same sorts of things, where usually in Hoare logic the pres and posts are different sorts of things, assertions. Ordinary Hoare logic can be recovered, though, by injecting assertions into program meanings. Remarkably, all of the rules of BCSL can be derived from this interpretation of triples in a CKA (we won't list all the axioms of CKA here, but make some comments at the end of this section).

We follow this idea concerning triples as well, with a small twist. Rather than using $F; C \sqsubseteq G$ to interpret a Hoare triple, we will use $F \sqsupseteq C; G$. This fits very well with intuitions concerning session types. In a triple $\{\Delta\} P \{\Delta'\}$, we think of

$$\llbracket \Delta \rrbracket \sqsupseteq \llbracket P \rrbracket; \llbracket \Delta' \rrbracket$$

as saying that Δ overapproximates what P followed by Δ' might do in the future.

The alternate interpretation $F \sqsupseteq C; G$ of triples has been dubbed ‘Plotkin triple’ by Hoare, because he has shown (in as yet unpublished notes) that this interpretation of triples can be used to derive rules in a form of structural operational semantics. The only point we wish to make is that the alternate triple makes good sense when thinking temporally, about overapproximating the future, as is the case with Session Types. Indeed, the Plotkin triple satisfies all of the rules of Hoare logic, and even BCSL with a further stipulation concerning locality; see Definition 14 and the discussion thereafter.

6.2 Predicate Transformer Model

We now move on to the technical details.

Propositions, or Worlds. We suppose first that we are given a total ordered commutative monoid $(Props, \vdash, *, emp)$ in which the order has a least element \perp . As the order is a preorder, there can be many elements equivalent to \perp . We define

$$\text{false} = \{p \mid p \vdash \perp\}.$$

In connecting to BCSL the elements of $Props$ will be the propositions, but in the model construction they will use the kind of structure found in possible worlds semantics of bunched logic [14, 16].

Predicates. The model is built from predicate transformers on subsets of $Props$. We consider only the down-closed subsets, which corresponds to the idea that preconditions are closed under the rule of consequence on the left in Hoare logic. That is, if $F(post) = pre$ then we think of pre as a disjunction, and it does no harm to throw in all the elements that imply anything in pre . We define the downwards closure of a proposition $p \in Props$ as:

$$q \Downarrow = \{p \mid p \vdash q\}$$

This notation extends to sets of propositions in the evident way: $A \Downarrow = \bigcup \{q \Downarrow \mid q \in A\}$.

In understanding our use of down-closed sets it can be helpful to consider that, for $A, B \subseteq Props$, then

$$A \Downarrow \subseteq B \Downarrow \iff \forall a \in A. \exists b \in B. a \vdash b$$

where $(\cdot) \Downarrow$ is the downwards closure. Downwards closure allows us to model a natural entailment as subset inclusion.

The downwards-closed sets have logical structure familiar from Kripke's semantics of intuitionistic logic: they form a complete Heyting algebra. In particular, they are closed under arbitrary unions, and this will be used below in the semantics of \parallel . The downwards-closed sets also simultaneously have a monoidal structure as used in (intuitionistic) Bunched Logic, using a construction dating at least back to [11].

Definition 11 *Let $Preds$ be the set of non-empty down-closed subsets of $Props$. That is, a predicate X is a non-empty subset of $Props$ such that $P \in X$ and $Q \vdash P$ implies $Q \in X$. $Preds$, ordered by subset inclusion, has a total commutative monoid structure \otimes, I , where*

$$\begin{aligned} X \otimes Y &= \{p \mid p \vdash x * y \wedge x \in X \wedge y \in Y\} \\ I &= \{p \mid p \vdash emp\} \end{aligned}$$

Notice that, when instantiated to session types, the unit I consists of exactly the completed typing contexts plus the inconsistent ones. We are requiring predicates to be non-empty because of the presence of the least element \perp amongst the propositions, internalizing falsity as the set of inconsistent propositions. This is just a representation point. It would be isomorphic to say that a predicate is a (possibly empty) set of consistent propositions, down-closed amongst the consistent ones.

Predicate Transformers. We are going to define several operations on the monotone function space $Preds \rightarrow Preds$. In each case the input parameter X is an element of $Preds$, i.e., a non-empty down-closed subset of $Props$. For the function $do\text{-}after[Y]$ we also have $Y \in Preds$.

$$\begin{aligned}
(F_1 \parallel F_2)X &= \bigcup \{FX_1 \otimes GX_2 \mid X_1 \otimes X_2 \subseteq X\} \\
\text{nothing}X &= \text{if } X \supseteq I \text{ then } I \text{ else false} \\
(F_1; F_2)X &= F_1(F_2(X)) \\
\text{skip}X &= X \\
do\text{-}after[Y]X &= \text{if } X = true(= Props) \text{ then } Y \text{ else false}
\end{aligned}$$

The ordering \sqsubseteq we use on predicate transformers as the inverse pointwise order:

$$F \sqsubseteq G \iff \forall X. FX \supseteq GX.$$

Defining the order of the predicate transformers in this way allows us to characterize the order in terms of fault avoiding triples for partial correctness in the standard way, where if $\{P\}C\{Q\}$ and $C' \sqsubseteq C$ then $\{P\}C'\{Q\}$. That is if we interpret a triple as $P \subseteq wlp(C, Q)$, where $wlp(C, \cdot)$ is a weakest liberal precondition predicate transformer, with the additional expectation that the precondition P ensures avoidance of memory faults. If the order was subset inclusion then we would get the opposite effect which is not in line with partial correctness. The top element of this predicate transformer lattice is the function $\lambda X. \text{false}$ which validates the triple $\{\text{false}\}C\{Q\}$ and the bottom element of the lattice is the function $\lambda X. Props$ which validates all Hoare triples. We think of the top and bottom elements as universal faulting and diverging programs, respectively.

Locality and Exchange. With this definition of the predicate transformers and the order, we are now in a position to verify the exchange law mentioned earlier.

Lemma 12 (Exchange Law) *The predicate transformers satisfy*

$$(F_1 \parallel F_2); (G_1 \parallel G_2) \sqsubseteq (F_1; G_1) \parallel (F_2; G_2)$$

Proof:

$$\begin{aligned}
& ((F_1 \parallel F_2); (G_1 \parallel G_2))X \\
&= \bigcup \{F_1Y_1 \otimes F_2Y_2 \mid Y_1 \otimes Y_2 \subseteq (G_1 \parallel G_2)X\} \\
&= \bigcup \{F_1Y_1 \otimes F_2Y_2 \mid Y_1 \otimes Y_2 \subseteq \bigcup \{G_1X_1 \otimes G_2X_2 \mid X_1 \otimes X_2 \subseteq X\}\} \\
&\supseteq \bigcup \{F_1(G_1X_1) \otimes F_2(G_2X_2) \mid X_1 \otimes X_2 \subseteq X\} \quad \text{where } Y_1 = G_1X_1 \text{ and } Y_2 = G_2X_2 \\
&= \bigcup \{(F_1; G_1)X_1 \otimes (F_2; G_2)X_2 \mid X_1 \otimes X_2 \subseteq X\} \\
&= ((F_1; G_1) \parallel (F_2; G_2))X
\end{aligned}$$

We are allowed to make the \supseteq step because $X_1 \otimes X_2 \subseteq X \Rightarrow G_1X_1 \otimes G_2X_2 \subseteq \bigcup \{G_1X_1 \otimes G_2X_2 \mid X_1 \otimes X_2 \subseteq X\}$ ■

As an example of the exchange law for heap programs, consider

$$\begin{aligned} & ([10] := 5 \parallel [20] := 7); ([20] := 3 \parallel [10] := 4) \\ & \quad \sqsubseteq \\ & ([10] := 5; [20] := 3) \parallel ([20] := 7; [10] := 4). \end{aligned}$$

On the left-hand side of the law we get a program with no races, whereas on the right-hand side we get a faulting program due to the race between address [10] and [20]. The right-hand side corresponds to \top in our semantics. The reason is that, given a postcondition, there is no way to split the precondition in a way that sends address 10 to both processes (each needs 10 to execute safely, and so we end up with $\{\perp\}$ as the precondition. (Our explanation of this example is intuitive rather than formal, but will be made formal in Section 6.3 below.)

It is evident that $(; , \text{skip})$ forms a monoid, and that \parallel is commutative and associative. Interestingly, skip is not the unit of \parallel for all monotone predicate transformers: rather, the unit is nothing .

Proposition 13 *nothing is the unit of \parallel for monotone predicate transformers.*

Proof: We simply calculate:

$$\begin{aligned} (F \parallel \text{nothing})X &= \bigcup \{FX_1 \otimes \text{nothing}X_2 \mid X_1 \otimes X_2 \subseteq X\} \\ &= \bigcup \{FX_1 \otimes I \mid X_1 \otimes X_2 \subseteq X \wedge X_2 \supseteq I\} \\ &= \bigcup \{FX_1 \mid X_1 \otimes X_2 \subseteq X \wedge X_2 \supseteq I\} \\ &= \bigcup \{FX_1 \mid X_1 \subseteq X\} \\ &= FX \end{aligned}$$

The second last step follows from the fact that $X_1 \supseteq X_1$ and by the monotonicity of \otimes we know that $X_1 \otimes X_2 \supseteq X_1 \otimes I = X_1$ since $X_2 \supseteq I$. Since $X_1 \otimes X_2 \subseteq X$ we know that $X_1 \subseteq X_1 \otimes X_2 \subseteq X$. Conversely, if $X_1 \subseteq X$, we can always pick $X_2 = I$ to obtain $X_1 \otimes X_2 \subseteq X \wedge X_2 \supseteq I$.

For the last step we know from the monotonicity of F that $\forall X_1 \subseteq X. FX_1 \subseteq FX$ hence $\bigcup \{FX_1 \mid X_1 \subseteq X\} \subseteq FX$ furthermore we know that $FX \in \bigcup \{FX_1 \mid X_1 \subseteq X\}$. ■

While skip is not the unit of \parallel in general, those elements that do have skip as a unit are special.

Definition 14 *A predicate transformer F is **local** if $F = F \parallel \text{skip}$.*

The idea to define locality in this way comes from our companion work [10]. Previously, before the algebraic structure was considered, locality had been defined in a more elaborate manner [5].

We will not restrict attention to local transformers only. In particular, the transformer $\text{do-after}[Y]$ used to interpret pre and postconditions below is not local. It is not local since $(\text{do-after}[Y] \parallel \text{skip})\text{true}$ will be $\bigcup \{Y \otimes X_2 \mid \text{true} \otimes X_2 \subseteq \text{true}\}$, whereas if it was local we would require the result to be Y and it evidently is not.

However, while our model contains non-local elements, it will be important that all programs are local, and for this the following lemma is essential. From this lemma we can conclude that if a program is built using $;$ and \parallel from primitive commands that are monotone and local, then the program itself is monotone and local as well.

- Lemma 15 (Preservation Lemma)** 1. *skip* is local, and if F and G are local and monotone then $F; G$ and $F \parallel G$ are local.
2. *skip* and *nothing* are monotone, and if F and G are monotone (wrt \sqsubseteq) then so are $F; G$ and $F \parallel G$. (That is, if $X \sqsubseteq Y$ then $(F; G)X \sqsubseteq (F; G)Y$ and $(F \parallel G)X \sqsubseteq (F \parallel G)Y$)
3. \parallel and $;$ are monotone wrt \sqsubseteq : if $F_1 \sqsubseteq G_1$ and $F_2 \sqsubseteq G_2$ then $F_1 \parallel F_2 \sqsubseteq G_1 \parallel G_2$ and $F_1; F_2 \sqsubseteq G_1; G_2$.

Proof: PROOF OF 15.1

case: *skip* is local:

$$\begin{aligned} (\text{skip} \parallel \text{skip})X &= \bigcup \{ \text{skip}X_1 \otimes \text{skip}X_2 \mid X_1 \otimes X_2 \subseteq X \} \\ &= \bigcup \{ X_1 \otimes X_2 \mid X_1 \otimes X_2 \subseteq X \} \end{aligned}$$

Since X is an upper bound of $\{X_1 \otimes X_2 \mid X_1 \otimes X_2 \subseteq X\}$ then by definition $\bigcup \{X_1 \otimes X_2 \mid X_1 \otimes X_2 \subseteq X\} \subseteq X$. For the other direction we pick $X_1 = X$ and $X_2 = I$. Then we get that $X \otimes I \in \{X_1 \otimes X_2 \mid X_1 \otimes X_2 \subseteq X\}$, hence $X \subseteq \bigcup \{X_1 \otimes X_2 \mid X_1 \otimes X_2 \subseteq X\}$. So we conclude that $(\text{skip} \parallel \text{skip})X = X = \text{skip}X$.

case: \parallel is local:

$$\begin{aligned} (F \parallel G) &= (F \parallel \text{skip}) \parallel (G \parallel \text{skip}) \\ &= ((F \parallel \text{skip}) \parallel G) \parallel \text{skip} && \text{associativity of } \parallel \\ &= (F \parallel (\text{skip} \parallel G)) \parallel \text{skip} && \text{associativity of } \parallel \\ &= (F \parallel (G \parallel \text{skip})) \parallel \text{skip} && \text{commutativity of } \parallel \\ &= ((F \parallel G) \parallel \text{skip}) \parallel \text{skip} && \text{associativity of } \parallel \\ &= (F \parallel G) \parallel (\text{skip} \parallel \text{skip}) && \text{associativity of } \parallel \\ &= (F \parallel G) \parallel \text{skip} && \text{locality of skip} \end{aligned}$$

case: $;$ is local: We first show $(F; G) \parallel \text{skip} \sqsubseteq F; G$ as follows.

$$\begin{aligned} ((F; G) \parallel \text{skip})X &= \bigcup \{ (F; G)X_1 \otimes X_2 \mid X_1 \otimes X_2 \subseteq X \} \\ &\supseteq \bigcup \{ (F; G)X \otimes I \mid X \otimes I \subseteq X \} \\ &= (F; G)X \end{aligned}$$

In this proof, we pick $X_1 = X$ and $X_2 = I$ at the appropriate point to give us $\{(F; G)X \otimes I \mid X \otimes I \subseteq X\} \subseteq \{(F; G)X_1 \otimes X_2 \mid X_1 \otimes X_2 \subseteq X\}$, and so $\bigcup \{(F; G)X \otimes I \mid X \otimes I \subseteq X\} \subseteq \bigcup \{(F; G)X_1 \otimes X_2 \mid X_1 \otimes X_2 \subseteq X\}$. Since $(F; G)X \otimes I = (F; G)X$ we know that $(F; G)X = \bigcup \{(F; G)X \otimes I \mid X \otimes I \subseteq X\}$.

Conversely, we show $(F; G) \parallel skip \sqsupseteq F; G$, with the following calculation.

$$\begin{aligned} (F; G) \parallel skip &= (F; G) \parallel (skip; skip) && \text{skip unit of ;} \\ &\sqsupseteq (F \parallel skip); (G \parallel skip) && \text{exchange law} \\ &= F; G && \text{locality of } F, G \end{aligned}$$

PROOF OF 15.2

Suppose $X \sqsubseteq Y$.

case: *skip* is monotone: By definition of *skip*, $skip X = X \sqsubseteq Y = skip Y$.

case: *nothing* is monotone: If $nothing X = \{\perp\}$ then we know $\{\perp\} \sqsubseteq nothing Y$.
If $nothing X = I$ then we know that $X \sqsupseteq I$ and since $X \sqsubseteq Y$ we know that $Y \sqsupseteq I$ and therefore by the definition of *nothing* we know that $nothing Y = I$.

case: \parallel preserves monotonicity:

$$\begin{aligned} (F \parallel G)X &= \bigcup \{FX_1 \otimes GX_2 \mid X_1 \otimes X_2 \subseteq X\} \\ &\subseteq \bigcup \{FY_1 \otimes GY_2 \mid Y_1 \otimes Y_2 \subseteq Y\} \\ &= (F \parallel G)Y \end{aligned}$$

For any X_1, X_2 , if $X_1 \otimes X_2 \subseteq X$ then $X_1 \otimes X_2 \subseteq Y$ because $X \sqsubseteq Y$ and because \subseteq is transitive. It follows that $\{FX_1 \otimes GX_2 \mid X_1 \otimes X_2 \subseteq X\} \subseteq \{FX_1 \otimes GX_2 \mid X_1 \otimes X_2 \subseteq Y\}$ and so by definition $\bigcup \{FX_1 \otimes GX_2 \mid X_1 \otimes X_2 \subseteq X\} \subseteq \bigcup \{FX_1 \otimes GX_2 \mid X_1 \otimes X_2 \subseteq Y\}$.

case: $;$ preserves monotonicity:

$$(F; G)X = F(GX)$$

Since G is monotone and $X \sqsubseteq Y$ then we know that $GX \sqsubseteq GY$. Since F is monotone then we know that $F(GX) \subseteq F(GY)$, thus we get our desired result $(F; G)X \subseteq (F; G)Y$.

PROOF OF 15.3

Suppose $F_1 \sqsubseteq G_1$ and $F_2 \sqsubseteq G_2$

case: \parallel is monotone:

$$\begin{aligned} (F_1 \parallel F_2)X &= \bigcup \{F_1X_1 \otimes F_2X_2 \mid X_1 \otimes X_2 \subseteq X\} \\ (G_1 \parallel G_2)X &= \bigcup \{G_1X_1 \otimes G_2X_2 \mid X_1 \otimes X_2 \subseteq X\} \end{aligned}$$

By the monotonicity of \otimes and the assumptions, we know that $F_1X_1 \otimes F_2X_2 \sqsubseteq G_1X_1 \otimes G_2X_2$. By definition we know that $\bigcup \{F_1X_1 \otimes F_2X_2 \mid \} \subseteq \bigcup \{G_1X_1 \otimes G_2X_2 \mid \}$

case: ; is monotone:

$$\begin{aligned}(F_1; F_2)X &= F_1(F_2X) \\ (G_1; G_2)X &= G_1(G_2X)\end{aligned}$$

Since $F_2 \sqsubseteq G_2$ we now that $F_2X \supseteq G_2X$ and since $F_1 \sqsubseteq G_1$ we know that $F_1(F_2X) \supseteq G_1(G_2X)$ which by definition is $(F_1; F_2)X \sqsubseteq (G_1; G_2)X$.

■

If one has the Exchange Law and F is local and monotone, then the frame rule

$$\frac{P \sqsupseteq F; Q}{(P \parallel R) \sqsupseteq F; (Q \parallel R)}$$

holds for all P, Q and R . Note that the frame rule needs only F to be local, not P, Q or R .

Connecting various triples. This brings us to the next point, concerning Plotkin triples. The standard predicate transformer view of triples is that $\{Y\} F \{Z\}$ means $Y \sqsubseteq FZ$ (let us call the latter the ‘Dijkstra triple’). We can connect this to Plotkin triples, providing a measure of justification for the latter, by using the construct *do-after*[.] which converts a predicate into a predicate transformer. *do-after*[Y] is the greatest predicate transformer corresponding to the pre/post spec $\{Y\} - \{true\}$, and the Plotkin triple

$$do\text{-after}[X] \sqsupseteq F; do\text{-after}[Y]$$

is intended to provide a model of the pre/post spec $\{X\} F \{Y\}$ which says that the future of X over-approximates F followed by the future described by the postcondition.

Theorem 16 (Connecting Dijkstra and Plotkin Triples) *For all $Y, Z \in Preds$ and monotone $F : Preds \rightarrow Preds$,*

$$Y \sqsubseteq FZ \text{ iff } do\text{-after}[Y] \sqsupseteq F; do\text{-after}[Z]$$

Proof: Only if direction: $Y \sqsubseteq FZ \Rightarrow do\text{-after}[Y] \sqsupseteq F; do\text{-after}[Z]$

Suppose $X \neq true$. We must show $do\text{-after}[Y]X \sqsubseteq (F; do\text{-after}[Z])X$. By the definition of *do-after* we know that $do\text{-after}[Y]X = \{\perp\} \sqsubseteq (F; do\text{-after}[Z])X$.
Suppose $X = true$. We must show $do\text{-after}[Y]X \sqsubseteq (F; do\text{-after}[Z])X$. By the definition of *do-after* we know that $do\text{-after}[Y]X = Y$ and $do\text{-after}[Z]X = Z$.
By the definition of ; we know that $(F; do\text{-after}[Z])X = F(Z)$. We have our desired result since $Y \sqsubseteq FZ$.

If direction: $do\text{-after}[Y] \sqsupseteq F; do\text{-after}[Z] \Rightarrow Y \sqsubseteq FZ$

We take X to be *true*, then we show $Y \sqsubseteq FZ$. By the definition of *do-after* and ; we know that $Y \sqsubseteq F(Z)$ therefore we have our desired result.

■

For this result we did not mention locality at all. Suppose we were interested in an algebra containing only local transformers, including for the pre/post specs. Could we similarly characterize pre/post specs using Plotkin triples?

We can almost get there using the following calculation, using $do\text{-}after[-] \parallel skip$ to create a local predicate transformer from a predicate.

$$\begin{aligned}
& (do\text{-}after[Y] \parallel skip) \sqsupseteq F; (do\text{-}after[Z] \parallel skip) \\
\text{iff } & (do\text{-}after[Y] \parallel skip) \sqsupseteq (F * skip); (do\text{-}after[Z] \parallel skip) && \text{by Locality} \\
\text{iff } & (do\text{-}after[Y] \parallel skip) \sqsupseteq (F; do\text{-}after[Z]) \parallel (skip; skip) && \text{by Exchange} \\
\text{iff } & (do\text{-}after[Y] \parallel skip) \sqsupseteq (F; do\text{-}after[Z]) \parallel skip && \text{by Unity}
\end{aligned}$$

Then, if $Y \subseteq GZ$ we obtain $(do\text{-}after[Y] \parallel skip) \sqsupseteq F; (do\text{-}after[Z] \parallel skip)$ from the monotonicity of $(\cdot) \parallel skip$ and Theorem 16.

For the reverse direction, though, we cannot use this same sort of reasoning, because while $(\cdot) \parallel skip$ preserves order, it does not reflect it. That is, $F \parallel skip \sqsupseteq G \parallel skip$ does not imply $F \sqsupseteq G$ (for at least one of F, G nonlocal). Currently, we are unsure whether we can characterize ‘Dijkstra triples’ $Y \subseteq FZ$ in terms of Plotkin triples, when all transformers are required to be local. But, a characterization is easily at hand if we drop the assumption of locality (for the preconditions and postconditions, at any rate).

Thus, our use of the non-local $do\text{-}after[Y]$ in the characterization of Dijkstra in terms of Plotkin might be read as a (weak) suggestion that locality $F = F \parallel skip$ should perhaps not be taken as an absolute requirement in the development of algebraic structure linking \parallel and $;$.

The previous result, Theorem 16, on Plotkin and Dijkstra triples provides a kind of sanity check on the former, showing that they connect to the standard interpretation of pre/post specs for predicate transformers. Our final result completes the picture, by linking the truth of the Dijkstra (hence, Plotkin) triple back to provability in BCSL.

Theorem 17 (Soundness and Completeness of Model) *Let $\llbracket c \rrbracket$ be the predicate transformer defined by c according to the above. For primitive commands c_{prim} we assume that the local predicate transformer $\llbracket c_{prim} \rrbracket$ is given, and that the property “ $\exists q \in X. \{p\} c_{prim} \{q\}$ is derivable in BCSL iff $p \in \llbracket c_{prim} \rrbracket X$ ” holds for these primitive statements. Given these presumptions:*

- (Soundness) $p \in \llbracket c \rrbracket X \Rightarrow \exists q \in X. \{p\} c \{q\}$.
- (Completeness) $(\exists q \in X. \{p\} c \{q\}) \Rightarrow p \in \llbracket c \rrbracket X$

Proof: Proof of Soundness. The proof is by induction on the structure of c , using the clauses in the definition of $\llbracket c \rrbracket$.

Case \parallel

$$p \in \llbracket c_1 \parallel c_2 \rrbracket X$$

From the definition of \parallel we know that $\exists X_1, X_2. X_1 \otimes X_2 \subseteq X \wedge p \in \llbracket c_1 \rrbracket X_1 \otimes \llbracket c_2 \rrbracket X_2$. From this we know that there are p_1, p_2 where $p \vdash p_1 * p_2$ where $p_1 \in \llbracket c_1 \rrbracket X_1$ and $p_2 \in \llbracket c_2 \rrbracket X_2$. By induction we get $\exists q_1 \in X_1. \{p_1\} c_1 \{q_1\}$ and $\exists q_2 \in X_2. \{p_2\} c_2 \{q_2\}$ and we know that $q_1 * q_2 \in X_1 \otimes X_2$. By the *Par*

rule we get $\{p_1 * p_2\} c_1 \parallel c_2 \{q_1 * q_2\}$ and then by the rule of consequence $\{p\} c_1 \parallel c_2 \{q_1 * q_2\}$. Now, since $X_1 \otimes X_2 \subseteq X$ we know that $q_1 * q_2 \in X$.

Case ;

$$p \in \llbracket c_1; c_2 \rrbracket X$$

From the definition of ; we know that $p \in \llbracket c_1 \rrbracket (\llbracket c_2 \rrbracket X)$. From induction we get that $\exists r \in \llbracket c_2 \rrbracket X$. $\{p\} c_1 \{r\}$ and, fixing such an r , that $\exists q \in X$. $\{r\} c_2 \{q\}$. We then apply the *Seq* rule to get our desired result choosing r as the linking assertion between c_1 and c_2 .

Case skip

$$p \in \llbracket skip \rrbracket X$$

By definition of *skip* we know that $p \in X$ and the result follows from the skip rule in BSCL $\{p\} skip \{p\}$ where $q = p$.

Case c_{prim}

$$p \in \llbracket c_{prim} \rrbracket X$$

This holds by the presumptions above.

Proof of Completeness. The proof is by induction on the derivation of $\{p\} c \{q\}$, and goes by a case analysis according to the last rule in the derivation.

Case skip

$$\overline{\{p\} skip \{q\}}$$

Where $q \in X$

because the skip rule has both pre and post equal, we know that $q = p$ and $p \in X$. The desired result follows since $\llbracket skip \rrbracket X = X$.

Case Par

$$\frac{\{p_1\} c_1 \{q_1\} \quad \{p_2\} c_2 \{q_2\}}{\{p_1 * p_2\} c_1 \parallel c_2 \{q_1 * q_2\}}$$

where $q_1 * q_2 \in X$

Let $X_1 = q_1 \Downarrow$, $X_2 = q_2 \Downarrow$. We aim first to show $X_1 \otimes X_2 \subseteq X$.

Suppose $t \in X_1 \otimes X_2$; then by definition of \otimes we know that $t \vdash x_1 * x_2$ for some $x_1 \in X_1$ and $x_2 \in X_2$. We also know that $x_1 \vdash q_1$ and $x_2 \vdash q_2$. By the monotonicity of $*$ it follows that $x_1 * x_2 \vdash q_1 * q_2$, and since $q_1 * q_2 \in X$ we know that $x_1 * x_2 \in X$ because X is downwards closed, hence we also know that $t \in X$; thus we conclude that $X_1 \otimes X_2 \subseteq X$.

By induction we know that $p_1 \in \llbracket c_1 \rrbracket X_1$ and $p_2 \in \llbracket c_2 \rrbracket X_2$. Since $p_1 * p_2 \vdash p_1 * p_2$ we know that $p_1 * p_2 \in \{p \mid p \vdash p_1 * p_2 \wedge p_1 \in \llbracket c_1 \rrbracket X_1 \wedge p_2 \in \llbracket c_2 \rrbracket X_2\}$ where we take $p = p_1 * p_2$. Therefore we can conclude that $p_1 * p_2 \in \llbracket c_1 \rrbracket X_1 \otimes \llbracket c_2 \rrbracket X_2$.

From the definition of $\llbracket \cdot \rrbracket$ we know that $\llbracket c_1 \parallel c_2 \rrbracket X = \bigcup \{ \llbracket c_1 \rrbracket Y_1 \otimes \llbracket c_2 \rrbracket Y_2 \mid Y_1 \otimes Y_2 \subseteq X \}$. For the particular X_1 and X_2 defined above we have already shown that $X_1 \otimes X_2 \subseteq X$, and it is therefore evident that $\llbracket c_1 \rrbracket X_1 \otimes \llbracket c_2 \rrbracket X_2 \subseteq \bigcup \{ \llbracket c_1 \rrbracket Y_1 \otimes \llbracket c_2 \rrbracket Y_2 \mid Y_1 \otimes Y_2 \subseteq X \}$. Therefore, $\llbracket c_1 \rrbracket X_1 \otimes \llbracket c_2 \rrbracket X_2 \subseteq \llbracket c_1 \parallel c_2 \rrbracket X$, and since we have already remarked $p_1 * p_2 \in \llbracket c_1 \rrbracket X_1 \otimes \llbracket c_2 \rrbracket X_2$, this proves $p_1 * p_2 \in \llbracket c_1 \parallel c_2 \rrbracket X$ as required.

Case Seq

$$\frac{\{p\} c_1 \{r\} \quad \{r\} c_2 \{q\}}{\{p\} c_1; c_2 \{q\}}$$

where $q \in X$

Let $X_r = r \Downarrow$. From induction we know that $p \in \llbracket c_1 \rrbracket X_r$ and $r \in \llbracket c_2 \rrbracket X$. By downwards closure we know that $r \Downarrow \subseteq \llbracket c_2 \rrbracket X$, therefore $p \in \llbracket c_1 \rrbracket (\llbracket c_2 \rrbracket X)$ by monotonicity of $\llbracket c_1 \rrbracket$ and the result follows from the definition $\llbracket c_1; c_2 \rrbracket X = \llbracket c_1 \rrbracket (\llbracket c_2 \rrbracket X)$.

Case Frame

$$\frac{\{p\} c \{q'\}}{\{p * r\} c \{q' * r\}}$$

where $q = q' * r \in X$. We are required to show that $p * r \in \llbracket c \rrbracket X$.

Let $X_1 = q' \Downarrow$, $X_2 = r \Downarrow$.

We aim first to show $X_1 \otimes X_2 \subseteq X$. Suppose $t \in X_1 \otimes X_2$; then by definition of \otimes we know that $t \vdash x_1 * x_2$ for some $x_1 \in X_1$ and $x_2 \in X_2$. We also know that $x_1 \vdash q'$ and $x_2 \vdash r$. By the monotonicity of $*$ it follows that $x_1 * x_2 \vdash q' * r$, and since $q' * r \in X$ we know that $x_1 * x_2 \in X$ because X is downwards closed, hence we also know that $t \in X$; thus we conclude that $X_1 \otimes X_2 \subseteq X$.

Now, to prove the case, by induction we know that $p \in \llbracket c \rrbracket X_1$. Since $p * r \vdash p * r$ we know that $p * r \in \{p' \mid p' \vdash p_1 * p_2 \wedge p_1 \in \llbracket c \rrbracket X_1 \wedge p_2 \in X_2\}$ where we take $p' = p * r$. Therefore we can conclude that $p * r \in \llbracket c \rrbracket X_1 \otimes X_2$. We can assume that $\llbracket c \rrbracket$ is *local* because the theorem assumes $\llbracket c_{prim} \rrbracket$ local, and Lemma 15 shows that locality is preserved by parallel and sequential composition. By locality, we know that $\llbracket c \rrbracket (X) = \llbracket c \parallel skip \rrbracket (X) = \bigcup \{ \llbracket c \rrbracket Y_1 \otimes skip Y_2 \mid Y_1 \otimes Y_2 \subseteq X \} = \bigcup \{ (\llbracket c \rrbracket Y_1) \otimes Y_2 \mid Y_1 \otimes Y_2 \subseteq X \}$. For the particular X_1 and X_2 defined above we have already shown that $X_1 \otimes X_2 \subseteq X$, and it is therefore evident that $(\llbracket c \rrbracket X_1) \otimes X_2 \subseteq \bigcup \{ (\llbracket c \rrbracket Y_1) \otimes Y_2 \mid Y_1 \otimes Y_2 \subseteq X \}$. Therefore, $(\llbracket c \rrbracket X_1) \otimes X_2 \subseteq \llbracket c \parallel skip \rrbracket (X) = \llbracket c \rrbracket (X)$, and since we have already remarked $p * r \in (\llbracket c \rrbracket X_1) \otimes X_2$, this proves $p * r \in \llbracket c \rrbracket X$ as required.

Case Cons

$$\frac{p' \vdash p \quad \{p\} c \{q\} \quad q \vdash q'}{\{p'\} c \{q'\}}$$

where $q' \in X$

Since X is downwards closed and we know that $q' \in X$ with the fact that $q \vdash q'$ then we know that $q \in X$ therefore $q \Downarrow \subseteq X$. Now we claim that $p' \in \llbracket c \rrbracket(q \Downarrow) \Rightarrow p' \in \llbracket c \rrbracket X$; this is true since $\llbracket c \rrbracket(q \Downarrow) \subseteq \llbracket c \rrbracket X$ because $q \Downarrow \subseteq X$ and $\llbracket c \rrbracket$ is monotone. We know by induction that $p \in \llbracket c \rrbracket(q \Downarrow)$ and since $p' \vdash p$ we know that $p' \in \llbracket c \rrbracket(q \Downarrow)$ by downwards closure of $\llbracket c \rrbracket(q \Downarrow)$.

■

The net effect of Theorems 16 and 17 is that, starting from BCSL, we have constructed an algebra in which the relationship $do\text{-}after[p \Downarrow] \sqsupseteq \llbracket c \rrbracket; do\text{-}after[q \Downarrow]$ is equivalent to provability of the Hoare triple $\{p\} c \{q\}$.

6.3 On the Session Types Instantiation

The predicate transformers $\llbracket k!j \rrbracket$ and $\llbracket k?j.P \rrbracket$ are defined by reference to provability in BCSL/ST.

$$\begin{aligned} \llbracket k!j \rrbracket X &= \{p \mid \exists q \in X. \{p\} k!j \{q\} \text{ is provable}\} \\ \llbracket k?j.P \rrbracket X &= \{p \mid \exists q \in X. \{p\} k?j.P \{q\} \text{ is provable}\} \end{aligned}$$

That each of these predicate transformers is local follows from the frame rule in BCSL/ST. That each is monotone follows from the rule of consequence in BCSL/ST.

We remark that the technical results on completeness, etc, earlier in this section do not literally apply to the binding form $k?j.P$, because the results above refer to programs built from primitive commands using \parallel and $;$, and $k?j.P$ is not of this form. However, the extension of these results is straightforward, and omitted.

A session types version of the earlier instance of the exchange law is

$$\begin{aligned} (k_1!a \parallel k_2!b); (k_2!c \parallel k_1!d) \\ \sqsubseteq \\ (k_1!a; k_2!c) \parallel (k_2!b; k_1!d). \end{aligned}$$

The first command satisfies the triple

$$\{k_1 : ![\alpha]; ![\alpha]; \text{end}, k_2 : ![\alpha]; ![\alpha]; \text{end}, a, b, c, d : \alpha\} (k_1!a \parallel k_2!b); (k_2!c \parallel k_1!d) \{\text{end}\}$$

The only preconditions that lead to a provable triple for the second program are inconsistent ones. As a consequence, the second program denotes \top in the predicate transformer model. (The reader might enjoy working through the calculation that this program denotes \top .) The second program can be rendered in BST, and we can ask a typing question as follows:

$$(k_1!a.k_2!c.\text{inact}) \parallel (k_2!b.k_1!d.\text{inact}) \triangleright \Delta \quad ??$$

In fact, there is no consistent typing context Δ that can be found as a Baby Session Types program making this judgement true, because it has channel races. This is the case for any session types program that (after translation to BCSL) denotes \top in the predicate transformer model.

The first program, $(k_1!a \parallel k_2!b); (k_2!c \parallel k_1!d)$, is not in the range of the translation from BST to BCSL, because it uses a compound command before a $;$ and BST does not have full $;$. It is worth discussing why we are able to prove a Hoare triple of this command in BCSL. The reason is that we have used a non-trivial postcondition in

$$\overline{\{k: ![\alpha]; \beta * j: \Delta\} k!j \{k: \beta\}}$$

where k need not have type end.

This raises a question. If we attempt to extend the local reasoning idea from [15] to session types, or to Hoare logic with session types, we might say that a spec should be able to concentrate only on the channels that a program uses, and *for only the time that the channel is used*. In this case, the presence of β rather than end in the typing rule for $k!j$ is making a prediction about the continuation, talking about the time after the communication happens: it is a little bit non-local, in a temporal sense. Why do we have to specify what happens outside the ‘temporal footprint’ of a program?

It seems that what is happening is that there is a missing inference rule, which we might call the ‘futuristic frame rule’

$$\frac{\{X\} c \{Y\}}{\{X; F\} c \{Y; F\}}$$

The idea is that if we extend our precondition into the future, then the postcondition can similarly be extended.

The futuristic rule makes immediate semantic sense in terms of Plotkin triples. The validity of

$$\frac{P \sqsupseteq C; Q}{P; F \sqsupseteq C; Q; F}$$

follows at once from associativity and monotonicity (on the left) of $;$. We should also note that the ‘archaic frame rule’

$$\frac{\{X\} c \{Y\}}{\{F; X\} c \{F; Y\}}$$

is not valid in terms of Plotkin triples, and we would not expect it to be when thinking about intuitions coming from Session Types.

Including the futuristic rule would perhaps allow us to reason about a system with sequential (temporal) compositionality. Making good on such speculation must, alas, be left to other work. We remark in particular that, syntactically, the exact correct way to formulate this rule in session types languages, and an understanding of its typechecking properties, would require detailed study.

6.4 Summary of the Algebraic Structure

The model we have found exhibits the following structure.

1. An ordered bisemigroup $(;, \parallel)$ in which \parallel is commutative, satisfying the exchange law. Here, a bisemigroup is a poset equipped with two associative and monotone operators.

2. A unit $skip$ of $;$ which is idempotent wrt $*$ (i.e., $skip * skip = skip$).

This is the structure found as well in our companion paper [10]. It allows a definition of the local elements as those elements satisfying $f \parallel skip = f$, and $skip$ then becomes a unit of \parallel as well as $;$ for the local elements. It is variations on and extensions of this sort of structure that are currently being investigated in work with Hoare, Möller and others.

Incidentally, the work in [10], which has similar but not identical structure, could not have been used for our purposes here. It uses an unordered rather than ordered monoid, and it takes that choice because the monoid elements there are thought of as concrete states, where here they are propositions. And we think of a Session Typing context as the analogue of a proposition rather than a state.

7 Conclusion

In this paper we have investigated links between three formalisms for concurrency: Session Types, Concurrent Separation Logic, and algebraic models (exemplified by Concurrent Kleene Algebra).

The model we gave in Section 6 is compositional but partly syntactic in nature. It has some of the character of a term model in logic, built from proof theory, though the use of mathematical functions (predicate transformers) to interpret some aspects of the language is not exclusively syntactic. In the Session instantiation, the model is built from Session Types themselves, and thus cannot be taken as providing an independent meaning of the type system. Our original aim in this work was to provide a denotational, fully syntax free interpretation of Session Types using a model that simultaneously captures traces and ownership, as in [8]. However, we were unsuccessful in this attempt, and providing a convincing denotational model of Session Types, particularly a compositional interpretation of the contexts Δ , is a problem we do not see how to solve.

Rather than providing justification for proof rules, the value of the model lies in making links between and describing underlying principles of different formalisms. In particular, it interprets a Session Typing judgement $P \triangleright \Delta$ as a relationship $\llbracket P \rrbracket \sqsubseteq \llbracket \Delta \rrbracket; \llbracket emp \rrbracket$, corresponding to the idea that Δ provides an over-approximation (in the sense used in abstract interpretation) of what P will do. When applied to Concurrent Separation Logic based on Session Types, it interprets a triple $\{\Delta\} P \{\Delta'\}$ as $\llbracket \Delta \rrbracket \supseteq \llbracket P \rrbracket; \llbracket \Delta' \rrbracket$, thus providing a concrete version of pre/post specs in which we think of the precondition as providing an overapproximation of what a program and its continuation might do in the future. As well as containing some detailed information concerning the relation between Concurrent Separation Logic and Session Types, we hope that this paper is in some way a contribution to or illustration of ideas in a developing line of work on algebraic models of concurrency, stemming from Abstract Separation Logic and Concurrent Kleene Algebra [5, 7, 20, 6, 10]. In particular, we have shown how, starting from a notion of Concurrent Separation Logic, one may derive some of the structure found in Concurrent Kleene Algebra, complementing the observations on the inverse direction in [7].

ACKNOWLEDGEMENTS. We thank Philipa Gardner for encouraging us to make the link to Session Types, Hongseok Yang for nudging us towards predicate transformers, Vasco Vasconcelos for advice on Session Types, and Tony Hoare and Bernhard Möller for discussions on algebra.

References

1. K. Bierhoff and J. Aldrich. Modular typestate checking of aliased objects. In *22nd OOPSLA*, pages 301–320, 2007.
2. Stephen L. Bloom and Zoltán Ésik. Free shuffle algebras in language varieties. *Theor. Comput. Sci.*, 163(1&2):55–98, 1996.
3. S. D. Brookes. A semantics of concurrent separation logic. *Theoretical Computer Science*, 375(1-3):227–270, 2007. (Preliminary version appeared in CONCUR’04, LNCS 3170, pp16-34).
4. L. Caires. Spatial-behavioral types for concurrency and resource control in distributed systems. *Theor. Comput. Sci.*, 402(2-3):120–141, 2008.
5. C. Calcagno, P.W. O’Hearn, and H. Yang. Local action and abstract separation logic. In *LICS*, pages 366–378. IEEE Computer Society, 2007.
6. H.-H. Dang, P. Höfner, and B. Möller. Towards algebraic separation logic. In *RelMiCS, Springer LNCS 5827*, pages 59–72, 2009.
7. C. A. R. Hoare, B. Möller, G. Struth, and I. Wehrman. Concurrent Kleene algebra. In *20th CONCUR, Springer LNCS 5710*, pages 399–414, 2009.
8. T. Hoare and P.W. O’Hearn. Separation logic semantics for communicating processes. *Proceedings of 1st FICS Conference, Electr. Notes Theor. Comput. Sci.*, 212:3–25, 2008.
9. K. Honda, V. T. Vasconcelos, and M. Kubo. Language primitives and type discipline for structured communication-based programming. In *7th ESOP, Springer LNCS 1381*, pages 122–138, 1998.
10. A. Hussain, P.W. O’Hearn, and R.L. Petersen. The algebraic structure of local state transformers. *Manuscript, January*, 2011.
11. J. Lambek. Deductive systems and categories I: Syntactic calculus and residuated categories. *Mathematical Systems Theory*, 2, 1968.
12. K. R. M. Leino, P. Müller, and J. Smans. Verification of concurrent programs with chalice. In *FOSAD, Springer LNCS 5705*, pages 195–222, 2009.
13. P. W. O’Hearn. Resources, concurrency and local reasoning. *Theoretical Computer Science*, 375(1-3):271–307, May 2007. Preliminary version appeared in CONCUR’04, LNCS 3170, 49–67.
14. P. W. O’Hearn and D. J. Pym. The logic of bunched implications. *Bulletin of Symbolic Logic*, 5(2):215–244, 99.
15. P.W. O’Hearn, J.C. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. *15th CSL, LNCS 2142, pp1-19*, 2001.
16. D. Pym, P. O’Hearn, and H. Yang. Possible worlds and resources: the semantics of BI. *Theoretical Computer Science*, 315(1):257–305, 2004.
17. R.D. Tennent. *Semantics of Programming Languages*. Prentice-Hall International, 1991.
18. V.T. Vasconcelos. Fundamentals of session types. In *9th SFM, Springer LNCS 5569*, pages 158–186, 2009.
19. J. Villard, É. Lozes, and C. Calcagno. Proving copyless message passing. In *7th APLAS, Springer LNCS 5904*, pages 194–209, 2009.
20. I. Wehrman, C. A. R. Hoare, and P. W. O’Hearn. Graphical models of separation logic. *Inf. Process. Lett.*, 109(17):1001–1004, 2009.

21. N. Yoshida and V. T. Vasconcelos. Language primitives and type discipline for structured communication-based programming revisited: Two systems for higher-order session communication. *Electron. Notes Theor. Comput. Sci.*, 171(4):73–93, 2007.