Linearity, Control Effects, and Behavioral Types*

Luís Caires[1] and Jorge A. Pérez[1,2]

[1] NOVA LINCS and Departamento de Informática, FCT, Universidade Nova de Lisboa
[2] University of Groningen & CWI, Amsterdam

Abstract. Mainstream programming idioms intensively rely on state mutation, sharing, and concurrency. Designing type systems for handling and disciplining such idioms is challenging, due to long known conflicts between internal non-determinism, linearity, and control effects such as exceptions. In this paper, we present the first type system that accommodates non-deterministic and abortable behaviors in the setting of session-based concurrent programs. Remarkably, our type system builds on a Curry-Howard correspondence with (classical) linear logic conservatively extended with two dual modalities capturing an additive (co)monad, and provides a first example of a Curry-Howard interpretation of a realistic programming language with built-in internal non-determinism. Thanks to its deep logical foundations, our system elegantly addresses several well-known tensions between control, linearity, and non-determinism: globally, it enforces progress and fidelity; locally, it allows the specification of non-deterministic and abortable computations. The expressivity of our system is illustrated by several examples, including a typed encoding of a higher-order functional language with threads, session channels, non-determinism, and exceptions.

1 Introduction

In this paper, we study a principled, typeful foundation to represent a relevant class of control effects within a behavioral type system for stateful concurrent programs. Sophisticated structural type systems have shaped mainstream static type checking for a long time now, and are fairly complete tools to discipline and effectively check programs that manipulate pure values. Unfortunately, the same cannot be said for most mainstream programming idioms, which intensively rely on state mutation, sharing, and often concurrency, about which "standard" type systems are quite silent.

Interactive concurrent systems need to manipulate stateful resources, ranging from basic memory references and passive objects (such as files, locks, and communication channels) to dynamic entities (such as threads or web references) typically subject to linearity constraints. To extend type-based verification techniques to this challenging setting, substructural type systems, based on various forms of linearity and affinity, have been increasingly investigated [31,45,47,46,35], and start to make their way towards practical adoption. Recent examples include Mozilla's Rust, but also embeddings of session types in target languages without linear types [32,41]. Some approaches use types to model states (cf. assertions); examples include typestate and several affine, linear, and stateful type systems, see, e.g., [20,31,33]. In other works, types are used to model behaviors (cf. processes); examples in this line include session types and usage types such as, e.g., [27,28,34], often referred to as *behavioral types* [29].

* Extended version, with appendices (January 21, 2017).

```
LET a = Ref() IN LET b = Ref() IN          LET a = Ref() IN LET b = Ref() IN
LET x1 = f() IN                            LET x1 = f() IN
LET a′ = write a x1 IN                     LET x2 = g() IN
LET x2 = g() IN                            LET a′ = write a x1 IN
LET b′ = write b x2 IN (free a′; free b′)  LET b′ = write b x2 IN (free a′; free b′)
```

Fig. 1. Two code snippets.

Linear types are in general very expressive, and the fine-grained specifications of usage types that they typically support may simultaneously bring a benefit and a curse. In particular, a still open issue is how to seamlessly combine linearity with many other useful programming mechanisms—a prominent example being the interaction of linearity with non-determinism and control effects, such as exceptions and (linear) continuations. The general issue is that by their very essence control effects (and associated programming language constructs) conflict with the linear, stateful usage discipline of values manipulated by programs, which makes it difficult to statically check programs. For example, when a communication channel is aborted, any linear values held by a channel client continuation must be aborted as well (which may not be always possible), or passed away to some candidate consumer code in scope. Likewise, after an exception is raised, it is not always clear how to safely discard a continuation holding linear values, nor how to proceed after the exception is caught. This situation is already present in non-deterministic programs where subexpressions may return more than one result, or even no result at all (e.g., “fail”), as in the non-deterministic monad.

These challenges and conflicts are illustrated by the examples in Fig. 1, adapted from [46], which we express in an idealized linear functional language. The Ref function is assumed to return a stateful fresh value r that may either be initially discarded, or subject to a strictly linear protocol consisting of a $write\ r\ x$ operation followed by a $free\ r$ operation. We use a common idiom when programming with usage types, in which an operation f acting on a linear value a that needs to be used according to a stateful protocol would be called as $a' = (fa)$, where a' refers to the new state of a (which gets “consumed” in the call $(f\ a)$). So, in our examples, failure to call $free\ r'$ after $r' = write\ r\ x$ may result in, say, a memory leak. Now, in Fig. 1 (left), if for some reason the call $f()$ raises an exception, the continuation will be safely aborted, but if $f()$ succeeds and the call $g()$ raises an exception instead, the resulting behavior would be ill-defined, as the required execution of $free\ a'$ will be discarded. On the other hand, consider the slightly different code snippet in Fig. 1 (right): even assuming that $f()$ or $g()$ may raise an exception, there will be no value usage violations, since both a and b will still be in their initially discardable state at such stage. A suitable typing discipline should deem the left snippet unsafe but the right snippet safe, taking into account the interference between control effects and the linear usage behavior of values in scope.

For a further example, consider the slightly more involved scenario given in Fig. 2, which involves concurrent communication and explicit exception handling. Our idealized linear functional language is now assumed to include the ability to fork threads, and manipulate session typed communication channels. In the first line of Fig. 2, a thread

```
LET log = FORK l. RECV(l, m_1); RECV(l, m_2); CLOSE ? l IN
LET res = FORK f. RECV(f, code); SEND(f, qs(code)); RECV(f, bk); CLOSE ? f IN
        SEND(res, QU2112); RECV(res, qss);
            TRY
                LET log = Check log qss IN
                    SEND(res, bkok); SEND(log, “ok”); CLOSE ! log; CLOSE ! res
            CATCH logc. SEND(logc, “fail”); CLOSE ! logc
```

Fig. 2. Code snippet for the server example.

representing a logging server is forked: the fork primitive FORK $l.e$ spawns a thread with body e accessing one endpoint of session channel l, and returns the other endpoint (bound to log) to the caller. The logging server receives precisely two messages, whose payload is a string, and closes the connection. Then another concurrent thread is spawned, mocking a resource allocation service: it receives a resource code, returns the quality of service constraints, and receives some reservation information bk.

The server at channel res is used by client code that first sends a resource code QU2112, receives a quality of service qss spec, and then calls a conformance checking operation $Check$ which, crucially, may raise an exception:

$$\texttt{LET}\ Check = \lambda l.\lambda x.\texttt{SEND}(l, \text{“}checkin\text{”}); \mathsf{if}\ \ Valid(x)\ \mathsf{then}\ l\ \mathsf{else}\ \texttt{THROW}\,(l)\ \texttt{IN}\ldots$$

All interactions between client and server are meant to be logged, so channel log is also passed to the $Check$ function together with the quality of service specification.

The overall expected behavior would be as follows. If $Valid()$ returns true, then $Check$ succeeds and the client will proceed booking the resource; however, if $Check$ raises an exception, the continuation will be discarded and the exception handler invoked. In this case, the continuation of the resource allocation thread at the other endpoint will also need to be aborted. If the overall ongoing (linear) session between client and server could be safely discarded at that particular stage of the protocol, then the overall behavior may be deemed safe, even if the log could not be safely aborted, since the linear outstanding interaction on the log endpoint will be performed anyway by the exception handler. Indeed, notice that the log is linearly passed to the exception handler as $logc$. Again, a suitable typing discipline combining effects and linearity should be able to express the assumptions underlying the reasoning above, and deem the code snippet safe, under the stated assumptions, but unsafe if the client server session is not safely abortable exactly before the RECV(f, bk) interaction (cf. Line 2 in Fig. 2). Moreover, any such typing discipline should also be compatible with internal non-determinism, in the sense that if the result of $Valid()$ is non-deterministic, then the resulting computation must be soundly typed for any alternative result, including the degenerate situation in which no value at all is returned, and the rest of the computation needs to safely abort, including, in that extreme case, the exception handler code itself!

The main goal of the paper is to investigate a principled foundation to express, reason and type-check a wide class of control effects in the context of a linear behavioral type system. Crucially, our approach builds on prior work on Curry-Howard correspondences between session types and various fragments of linear logic; our type system is a conservative extension of a standard system of classical linear logic. By approaching a Curry-Howard correspondence from the programming language perspective (in the spirit of, e.g., [18,6,5]), we introduce two new dual logical modalities—monadic $\&A$ and co-monadic $\oplus A$—with associated programming constructs and proof reductions. As is often the case for type systems motivated by Curry-Howard correspondences, our system ensures global progress and usage / session protocol fidelity. Moreover, it is intrinsically compatible with all other logically motivated constructs and methods introduced in prior/related work, such as behavioral polymorphism [10,48], logical relations [37], dependent types [42], higher-order code mobility [44], and multiparty protocols [9,16].

It turns out that our new modalities $\&A$ and $\oplus A$ suffice to express general forms of internal non-determinism, and, importantly, include failure—an explicitly typed form of affinity—as a special case. These two modalities can be seen as an additional pair of linear logic exponentials, and as such obey the basic monadic / co-monadic laws. However, while the standard linear logic modalities $!A$ and $?A$ encapsulate contraction and weakening, $\&A$ and $\oplus A$ encapsulate non-determinism and failure, in a sense to be made precise below. Although related to the non-deterministic monad and to well-known powerdomain models of non-determinism [39], a key novelty of our work is the perfect Curry-Howard match between proof reductions associated to the $\&A$ and $\oplus A$ modalities and sensible operational rules. This correspondence allows us to state cut-elimination, and to naturally derive key properties with practical impact (e.g., lock freedom, fidelity, and strong normalization), while supporting natural effectful programming idioms and powerful reasoning techniques (such as logical relations).

We will illustrate through examples how expressive linear usage protocols involving effects may be compiled down to the basic linear logic system extended with these two primitives and associated programming constructs. We will present our basic results and examples for a canonical session-based π-calculus model realizing a Curry-Howard interpretation of session types as linear logic propositions. As is well-known, the π-calculus is a complete foundational model, able to represent, e.g., general concurrent computation, higher-order data, and object-oriented features [40]. Hence, our development is carried out in the setting of most higher-level programming languages. In particular, we will use this process model as target language in a typed encoding of an effectful linear higher-order functional language with threads, session-typed channels, non-determinism, and exceptions, allowing us to show typings for the above examples.

Structure of the paper. Next, § 2 presents our Curry-Howard interpretation of session types for concurrent processes via examples. § 3 establishes meta-theoretical results for typed processes: cut elimination (Theorem 3.1), type preservation (Theorem 3.2), progress (Theorem 3.3). Also, a postponing result (Theorem 3.5) connects our process model with the (non confluent) non-determinism typical of process calculi. § 4 encodes λ^{exc} (a linear, higher-order functional language with concurrency and exceptions) into session-typed processes. λ^{exc} is the reference language for the motivating examples above. Theorem 4.2 ensures that our encoding preserves typing; therefore, all results in § 3 will carry over to λ^{exc}. In § 5 we discuss related works, and § 6 concludes.

2 The Core Language and its Type System

We base our development on a standard session-typed π-calculus, a core language in which general higher-order concurrent programs may be modeled and analyzed [40]. The (binary) session discipline [27,28] applies to pairs of name passing processes that communicate through point-to-point channels. In this setting, interaction between processes always occur in matching pairs: when one partner sends, the other receives; when one partner offers a selection, the other chooses; when a partner closes the session, the other must acknowledge—no further interactions may occur on the same channel. Sessions are initiated when a participant invokes a server, which acts as a shared provider, with the capability of unboundedly spawning fresh sessions between the invoking client and the newly created service instance process. A service name may be publicly shared by any clients in the environment. A session-based system exhibits concurrency and parallelism because many sessions may be executing simultaneously and independently. No races in communications within a session (or even between different sessions) can occur. Both session and server names may be passed around in communications. Session channels are subject to a linear usage discipline, conforming to a specific state dependent protocol, while server channels can be freely shared, and used by an arbitrary number of concurrent clients that can call on them for spawning new session instances.

Next we gradually introduce the ingredients of our typed process model (syntax, semantics, session types and their linear logic interpretation), which are summarized in Figure 3 (Page 14). This presentation allows us to better motivate and describe the key novelty in this paper—the(dual) types for non-deterministic behaviors in sessions.

Caires and Pfenning [11] introduced a type system for π-calculus processes that corresponds to a linear logic proof system, revealing the first Curry-Howard interpretation of session types as linear logic propositions. Unlike traditional session type systems, Curry-Howard interpretations of behavioral types ensure global progress (i.e., well-typed processes never get stuck), livelock-freedom, and confluence (up to $\equiv$), and may be developed within intuitionistic [11,13] or classical linear logic [13,48], with certain subtle differences in expressiveness. Our system extends the presentation Σ_2 of classical linear logic [1] with mix principles and, crucially, with new exponential modalities $\oplus A$ and $\& A$, which will be interpreted as (dual) types for non-deterministic sessions.

Definition 2.1 (Types). *Types (A, B, C) are given by*

$$A, B ::= \bot \mid \mathbf{1} \mid !A \mid ?A \mid A \otimes B \mid A \parr B \mid A \oplus B \mid A \mathbin{\&} B \mid \oplus A \mid \& A$$

In examples we will also assume given some basic (data) types (e.g., naturals, strings, etc), but will not elaborate the nature of such basic types (see, e.g., [42]). Despite notational similarity, there is no ambiguity between our new (unary) modalities $\oplus A$ and $\& A$ and standard linear logic (binary) operators for additive disjunction and conjunction.

For any type A, we define its dual $\overline{A}$, where $\overline{(\cdot)}$ corresponds to linear logic negation $(\cdot)^{\bot}$, following standard de Morgan-like laws. Intuitively, the type of a session endpoint is the dual of the type of the opposite endpoint.

Definition 2.2 (Duality). *The duality relation on types is given by:*

$$\begin{array}{lllll}
\overline{\mathbf{1}} = \bot & \overline{!A} = ?\overline{A} & \overline{A \otimes B} = \overline{A} \parr \overline{B} & \overline{A \oplus B} = \overline{A} \mathbin{\&} \overline{B} & \overline{\oplus A} = \& \overline{A} \\
\overline{\bot} = \mathbf{1} & \overline{?A} = !\overline{A} & \overline{A \parr B} = \overline{A} \otimes \overline{B} & \overline{A \mathbin{\&} B} = \overline{A} \oplus \overline{B} & \overline{\& A} = \oplus \overline{A}
\end{array}$$

Typing judgments have the form $P \vdash \Delta; \Theta$, where P is a program term, Δ is the linear context and Θ is the unrestricted context, along the lines of DILL [4] and Σ_2 [1]. Both contexts are assignments of types to (channel) names $x, y, z, \ldots$. We write '$\cdot$' to denote empty typing environments. After erasing the term P, our judgment corresponds exactly to a logical sequent in the classical linear logic Σ_2 of [1]. Remarkably, this formulation naturally supports a Curry-Howard interpretation for the exponentials $!A$ and $?A$ in terms of standard (π-calculus) semantics for lazy replication [11,13].

2.1 Reduction Semantics

The operational semantics of our session calculus is defined by a relation of reduction (denoted $P \to Q$) that expresses dynamic evolution, and a relation of structural congruence (denoted $P \equiv Q$), which equates processes with the same spatial (or static) structure. This semantics exhibits a precise correspondence with cut elimination at the logic level. While most cut-reduction steps directly correspond to process reductions, other cut-reduction steps are better expressed in the process world as structural congruence principles or as behavioral equivalences; this applies similarly to the so-called commuting conversions, which are known to capture typed behavioral equivalences [37].

To describe reductions and conversions on proof trees (which correspond to typing derivations), we introduce a simple algebraic notation. For each typing rule (T*) with k premises $d_1, \ldots, d_k$ we denote by T*$(p_1, p_2, \ldots)$ the derivation obtained by applying rule (T*) to the derivations $p_1, \ldots, p_k$. If the proof rule binds names $\tilde{x}$ in the conclusion (as in, e.g., cut), we would then write T*$(\tilde{x})(p_1, p_2, \ldots)$ to make this binding explicit.

2.2 Basic Typing Rules, Congruence Rules, and Reduction Rules

The parallel composition of processes is typed in our system by rules corresponding to the cut and mix principles (dependent and independent composition, respectively).

$$\frac{P \vdash \Delta; \Theta \quad Q \vdash \Delta'; \Theta}{P \mid Q \vdash \Delta, \Delta'; \Theta}\,(\mathsf{T} \mid\,) \quad \frac{P \vdash \Delta, x{:}\overline{A}; \Theta \quad Q \vdash \Delta', x{:}A; \Theta}{(\nu x)(P \mid Q) \vdash \Delta, \Delta'; \Theta}\,(\mathsf{Tcut}) \quad \frac{}{\mathbf{0} \vdash ; \Theta}\,(\mathsf{T}\cdot)$$

The mix rule (T |) types the composition of two processes that do not share linear names; P and Q run in parallel but do not interact. The cut rule (Tcut) types the composition of P and Q while establishing a binary session between them using a single linear channel x; each process holds one of the two (dual) endpoints x of a session of type A and $\overline{A}$. This channel is kept private to the composition by the restriction operator $(\nu x)(\ldots)$ so that the newly established session will not be affected by interferences. Rule (T$\cdot$) allows the inactive process **0** to be introduced. Neutrality of **0** is expressed by the conversion $\mathrm{T}|(\mathrm{T}\cdot, D) \cong D$ at the level of proofs, which corresponds exactly to the usual structural congruence principle $\mathbf{0} \mid P \equiv P$ (we consider here a conversion, not a computational reduction, since it does not involve any process interaction). We take process terms up to basic structural congruence principles, namely we assume that $- \mid -$ is commutative and associative with unit **0**, etc. This way, e.g., $P \mid Q$ and $Q \mid P$ denote the same process, i.e., the (unique) parallel composition of P and Q. Thus, Rule (Tcut) is symmetric w.r.t. its premises: if $\mathrm{Tcut}(D_1, D_2)$ is a derivation then $\mathrm{Tcut}(D_2, D_1)$ is the same derivation; we then also consider the conversion $\mathrm{Tcut}(D_1, D_2) \cong \mathrm{Tcut}(D_2, D_1)$.

Session Send and Receive Session-typed processes communicate by sending and receiving messages according to some session discipline. The message payload can be a value of some primitive data type or a session channel; we focus here on the general case of session passing (delegation). Type $A \otimes B$ is the type of a session that first sends a session of type A and then continues as a session of type B. As such, it corresponds to the session type $!A.B$ of [27]. Dually, $A \parr B$ is the type of a session that first receives a session of type A and then continues as a session of type B; it thus corresponds to the session type $?\overline{A}.B$. Hence, the session type $?A.B$ corresponds to the linear type $A \multimap B$. We have the following typing rules for send $A \otimes B$ and receive $A \parr B$.

$$\frac{P \vdash \Delta, y{:}A; \Theta \quad Q \vdash \Delta', x{:}B; \Theta}{\overline{x}(y).(P \mid Q) \vdash \Delta, \Delta', x{:}A \otimes B; \Theta}\ (\mathrm{T}\otimes) \qquad \frac{R \vdash \Gamma, y{:}C, x{:}D; \Theta}{x(y).R \vdash \Gamma, x{:}C \parr D; \Theta}\ (\mathrm{T}\parr)$$

An output process is then of the form $\overline{x}(y).M$, where y is a freshly created name. The behavior of such an output process is to send session y on x and then proceed as defined by M. In our typed language, the output continuation M has the form $P \mid Q$, where P defines the behavior of the session y being sent and Q the behavior of the continuation session on x. An input process is of the form $x(y).R$, a process that receives on session x a session n, passed in parameter y, and then proceeds as specified by R. The continuation R will use the received session but also any other open sessions (including x). Notice that y is bound both in $\overline{x}(y).M$ and in $x(y).R$, and so only fresh names can be sent in output processes; this corresponds to the *internal mobility* discipline [7], without loss of expressiveness. The associated principal cut reduction corresponds to process communication, where $C = \overline{A}$, $D = \overline{B}$, expressed by

$$\mathtt{Tcut}(x)(\mathrm{T}\otimes(y)(D_1, D_2), \mathrm{T}\parr(y)(D_3)) \to \mathtt{Tcut}(x)(\mathtt{Tcut}(y)(D_1, D_3), D_2)$$

This reduction exactly captures (bound output) communication in the π-calculus

$$(\nu x)(\overline{x}(y).M \mid x(y).R) \to (\nu x)(\nu y)(M \mid R)$$

where we write $M \equiv P \mid Q$. Although $- \mid -$ is commutative there is no ambiguity in Rule (T$\otimes$): P and Q are the split of M typed by $P \vdash \Delta, y{:}A; \Theta$ and $Q \vdash \Delta', x{:}B; \Theta$, respectively, and Δ and Δ' are the split of the linear context in the conclusion. The multiplicative units $\bot$ and $\mathbf{1}$ type session termination actions as seen from each endpoint; no partner can further use a closed session.

$$\frac{}{x.\overline{\mathtt{close}} \vdash x{:}\mathbf{1}; \Theta}\ (\mathrm{T}\mathbf{1}) \qquad \frac{P \vdash \Delta; \Theta}{x.\mathtt{close}; P \vdash x{:}\bot, \Delta; \Theta}\ (\mathrm{T}\bot)$$

The associated principal cut reduction corresponds to session termination, which we define at the level of processes and proof trees respectively by the rules

$$(\nu x)(x.\overline{\mathtt{close}} \mid x.\mathtt{close}; P) \to P \qquad \mathtt{Tcut}(x)(\mathrm{T}\mathbf{1}, \mathrm{T}\bot(D)) \to D$$

Types $\mathbf{1}$ and $\bot$ correspond to the single type $\mathtt{end}$ in usual session types, and usually have a silent interpretation. In the presence of mix principles, as we consider here, propositions $\bot \multimap \mathbf{1}$ and $\mathbf{1} \multimap \bot$ are valid. Considering $\bot = \mathbf{1}$, we could define a single type '$\bullet$' as standing for "both" $\mathbf{1}$ or $\bot$, where $\overline{\bullet} = \bullet$. (Recall that $A \multimap B \triangleq \overline{A} \parr B$.)

Session Offer and Choice The linear type $A \oplus B$ types a session that first *chooses* (from the dual partner menu) either “left” or “right”, and then continues as a session of type A or B, depending on the choice. This type is the binary version of the session type $\oplus_{i \in I}\{l_i{:}A_i\}$ (labeled internal choice). The linear type $A \& B$ types a session that first *offers* both “left” or “right” menu options and then continues as a session of type A or B, depending on the choice made by the partner. Thus, $A \& B$ is the binary version of the session type $\&_{i \in I}\{l_i{:}A_i\}$ (labeled external choice). Offers and choices are typed by the additive linear conjunction and disjunction $\&$ and $\oplus$, as defined by the rules:

$$\frac{R \vdash \Delta, x{:}A; \Theta}{x.\mathtt{inl}; R \vdash \Delta, x{:}A \oplus B; \Theta}\ (\mathrm{T}\oplus_1) \qquad \frac{R \vdash \Delta, x{:}B; \Theta}{x.\mathtt{inr}; R \vdash \Delta, x{:}A \oplus B; \Theta}\ (\mathrm{T}\oplus_2)$$

$$\frac{P \vdash \Delta, x{:}A; \Theta \quad Q \vdash \Delta, x{:}B; \Theta}{x.\mathtt{case}(P, Q) \vdash \Delta, x{:}A \,\&\, B; \Theta}\ (\mathrm{T}\&)$$

The associated principal cut reductions correspond to the process and proof reductions

$$\begin{array}{l}(\nu x)(x.\mathtt{case}(P, Q) \mid x.\mathtt{inl}; R) \to (\nu x)(P \mid R)\\ (\nu x)(x.\mathtt{case}(P, Q) \mid x.\mathtt{inr}; R) \to (\nu x)(Q \mid R)\end{array}$$

$$\begin{array}{l}\mathtt{Tcut}(x)(\mathrm{T}\&(D_1, D_2), \mathrm{T}\oplus_1(D_3)) \to \mathtt{Tcut}(x)(D_1, D_3)\\ \mathtt{Tcut}(x)(\mathrm{T}\&(D_1, D_2), \mathrm{T}\oplus_2(D_3)) \to \mathtt{Tcut}(x)(D_2, D_3)\end{array}$$

In examples we may consider n-are labeled sums, close to usual session types constructs:

$$\frac{R \vdash \Delta, x{:}A; \Theta}{x.\mathtt{l}_i; R \vdash \Delta, x{:} \oplus_{i \in I} \{\mathtt{l}_i : A_i\}; \Theta} \qquad \frac{P_i \vdash \Delta, x{:}A_i; \Theta \quad (\text{all } i \in I)}{x.\mathtt{case}_{i \in I}(\mathtt{l}_i.P_i) \vdash \Delta, x{:} \,\&_{i \in I} \{\mathtt{l}_i : A_i\}; \Theta}$$

with associated principal cut reduction expressed by

$$(\nu x)(x.\mathtt{case}_{i \in I}(\mathtt{l}_i.P_i) \mid x.\mathtt{l}_i; R) \to (\nu x)(P_i \mid R)$$

Example 2.3 (Movie Server (1)). Consider a toy scenario involving a movie server and some clients. We first model a single session (on channel s) established between client $\mathit{Alice}(s)$ and server instance $\mathit{SBody}(s)$. The server session offers two options: “buy movie” (`inl`), and “preview trailer” (`inr`). Alice selects the “preview” option from the server menu, and plays the corresponding protocol. Consider now the following terms:

$$\begin{array}{rl}\mathit{SBody}(s) &\triangleq s.\mathtt{case}(s(\mathit{title}).s(\mathit{card}).s\langle \mathit{movie}\rangle.s.\overline{\mathtt{close}}, s(\mathit{title}).s\langle \mathit{trailer}\rangle.s.\overline{\mathtt{close}})\\ \mathit{Alice}(s) &\triangleq s.\mathtt{inr}; s\langle \text{“}\mathit{mullholanddrive}\text{”}\rangle.s(\mathit{preview}).s.\mathtt{close}; \mathbf{0}\\ \mathit{System}_1 &\triangleq (\nu s)(\mathit{SBody}(s) \mid \mathit{Alice}(s))\end{array}$$

Assume some given basic types for movie titles (T), credit card data (C) and movie files M, which are self-dual (since they do not type communication capabilities but values of basic types). We can then provide the following types and derivable type assignments for the various system components as follows:

$$\begin{array}{l}\mathit{SBT} \triangleq (T \multimap C \multimap M \otimes \mathbf{1}) \ \& \ (T \multimap M \otimes \mathbf{1})\\ \mathit{SBody}(s) \vdash s : \mathit{SBT}\ ; \cdot \qquad \mathit{Alice}(s) \vdash s : \overline{\mathit{SBT}}\ ; \cdot\end{array}$$

We would then have $\mathit{System}_1 \vdash \cdot\, ; \cdot$. While the type of the server endpoint is SBT, the type of a client endpoint would be $\overline{\mathit{SBT}} = (T \otimes C \otimes M \multimap \bot) \oplus (T \otimes M \multimap \bot)$. □

Shared Service Definition and Invocation Shared service definition and invocation are typed by the linear logic exponentials ! and ?. Type $!A$ types a shared channel that persistently offers a replicated service which whenever invoked spawns a fresh session of type A (from the server's perspective). Dually, type $?A$ types a shared channel on which requests to a persistently replicated service of type A can be unboundedly issued (from the client's perspective). We consider the following typing rules:

$$\frac{P \vdash \Delta; x{:}A, \Theta}{P \vdash \Delta, x{:}?A; \Theta}\ (\mathrm{T?}) \qquad \frac{Q \vdash y{:}A; \Theta}{!x(y).Q \vdash x{:}!A; \Theta}\ (\mathrm{T!}) \qquad \frac{P \vdash \Delta, y{:}A; x{:}A, \Theta}{\overline{x}?(y).P \vdash \Delta; x{:}A, \Theta}\ (\mathsf{Tcopy})$$

The associated principal cut reduction corresponds to shared service invocation

$$(\nu x)(!x(y).Q \mid \overline{x}?(y).P) \to (\nu x)(!x(y).Q \mid (\nu y)(P \mid Q))$$

This operational interpretation of the rules for $!A$ and $?A$ (cf. [1,38,4], implementing "lazy" contraction) exactly coincides with the usual interpretation of lazy replication. Notice that Rule (T?) is silent on the term assignment: it implements a bookkeeping device to move the typed channel $x{:}?A$ to the unrestricted context, and does not induce a computational effect (e.g., as exchange is also implicitly handled).

As our typing judgments have two different regions, linear and exponential, two cut rules are required [4], one for cutting a linear (session) channel in the linear context (Rule (Tcut), already presented in § 2.2), and the following rule, for cutting an unrestricted (shared) channel in the exponential context [38,4]:

$$\frac{P \vdash y{:}\overline{A}; \Theta \quad Q \vdash \Delta; x{:}A, \Theta}{(\nu x)(!x(y).P \mid Q) \vdash \Delta; \Theta}\ (\mathsf{Tcut}^?)$$

For typing "source programs" only the linear Rule (Tcut) is required, but Rule (Tcut$^?$) is required for cut-elimination; hence, Rule (Tcut$^?$) is a "runtime" typing rule. The principal reduction above is expressed at the level of proofs by

$$\mathsf{Tcut}(x)(\mathrm{T!}(y)(D_1), \mathsf{Tcopy}(y)(D_2)) \to \mathsf{Tcut}_?(xy)(D_1, \mathsf{Tcut}(y)(D_1, D_2))$$

Example 2.4 (Movie Server (2)). We illustrate the usage of $!A$ and $?A$ types using a shared movie server, which may answer requests from an unbounded number of clients; here we use just two concurrent clients, $SAlice$ and $SBob$. Alice still selects the "preview trailer" option as in Example 2.3, but Bob selects the "buy movie" option. Recall the definitions of processes $SBody(s)$ and $Alice(s)$ and type SBT from Example 2.3.

$$MOVIES(srv) \triangleq\ !srv(s).SBody(s)$$

$$Bob(s) \triangleq s.\mathtt{inl}; \overline{s}(\text{``}inception\text{''}).\overline{s}(bobscard).s(mpeg).s.\mathtt{close}; \mathbf{0}$$

$$SAlice(srv) \triangleq \overline{srv}(s).Alice(s) \qquad SBob(srv) \triangleq \overline{srv}(s).Bob(s)$$

$$System_2 \triangleq (\nu srv)(MOVIES(srv) \mid SAlice(srv) \mid SBob(srv))$$

The following typing judgments are derivable:

$$MOVIES(srv) \vdash srv : !SBT\ ;\ \cdot \qquad SAlice(srv) \vdash \cdot\ ; srv : \overline{SBT}$$
$$SBob(srv) \vdash \cdot\ ; srv : \overline{SBT} \qquad Alice(srv) \mid Bob(srv) \vdash srv : ?\overline{SBT};\ \cdot$$

We can obtain $System_2 \vdash \cdot; \cdot$ as follows: we first use the (mix) Rule (T |) to compose the two clients; then, Rule (T?) is used to merge the shared endpoints under the explicit type $?\overline{SBT}$; finally, the clients are composed with the server using Rule (Tcut). □

Identity We interpret the identity axiom by the *forwarder process* $[x \leftrightarrow y]$ [12,48], which denotes a bidirectional (linear) link between sessions x and y, giving a logical justification to a known concept in π-calculi (cf. [24]). The forwarder at type A is typed

$$[x \leftrightarrow y] \vdash x{:}A, y{:}\overline{A}; \Theta \ (\mathsf{Tid})$$

The associated cut reduction $(\nu x)(P \mid [x \leftrightarrow y]) \to P\{y/x\}$ (where y is not free in P) is akin to the application of an explicit substitution. It is known since [30] that linear forwarders can simulate substitution in the sense of the above reduction rule. We also introduce $[x \leftrightarrow y] \equiv [y \leftrightarrow x]$ as a structural congruence axiom, as a direct consequence of (implicit) exchange in the typing context. While a well-typed copycat process F_A without forwarder links can be easily constructed for any concrete type A by η-expansion (see [13]) the primitive forwarder is important when considering polymorphism [10]. It also allows us to represent the "free" output construct $x\langle y\rangle.P$ (where y is a free channel name in scope) by $\overline{x}(z).([y \leftrightarrow z] \mid P)$ (cf. [7]).

2.3 Non-Determinism and Failure

The developments of this paper focus on the challenge of expressing fundamental primitives for non-deterministic behavior—including the special important case of abortable behavior—in the setting of our Curry-Howard correspondence for session types.

It is often believed that a Curry-Howard interpretation of a programming language is hard to reconcile with true (so-called internal) non-determinism in computation, since reduction steps should express at most behavioral equivalences on processes, via proof identities, which are inherently confluent from an operational viewpoint. However, it is clear, at least from work on denotational semantics and functional programming, that non-determinism can be handled equationally by working on the powerdomain of computation results. In the logical setting, developments on differential linear logic [22] also require the interpretation domain for proofs to be closed under a (formal) notion of "sum", which could be interpreted as non-deterministic choice. Although partially inspired by such approaches, our proposal picks a fairly different road, which turns out to lead to the first example of a Curry-Howard interpretation of a realistic programming language with built-in internal non-determinism.

It is well-known after Girard that the linear logic exponential modalities $!A$ and $?A$, which have been used above to model the type of shared channel names, are not uniquely defined by their standard proof rules: not surprisingly, if one adds additional operators defined by the same rules, we obtain independent monad / comonad pairs. We exploit this fact to our advantage, noting that it allows us to modularly add new "exponential" modalities to the base logical system, defined by identical proof rules (in Girard's original formulation), without semantically interfering with the existing ones. Any such pair of connectives (say, $\&A$ and $\oplus A$) will yield a dual monad / comonad pair defined by the fundamental principles (in a simplified form):

$$\frac{\vdash \Delta, A}{\vdash \Delta, \& A} \quad \frac{\vdash \& \Delta, A}{\vdash \& \Delta, \oplus A}$$

For the usual modalities $!A$ and $?A$, additional specific rules for $?A$ define the intended semantics of the linear logic exponentials, which encapsulate the structural principles of weakening and contraction:

$$\frac{\vdash \Delta}{\vdash \Delta, ?A} \quad \frac{\vdash \Delta, ?A, ?A}{\vdash \Delta, ?A}$$

These observations suggest a logically justified methodology for adding new monadic operators to the basic linear logic framework, by means of independent monad / comonad pairs in which the monad semantics is defined by specific additional logical principles. We develop our type system on top of (classical) linear logic, conservatively extended with two operators capturing a (co)monad defined by (a refined version of the) following principles, which can be verified to be sound for an (additive) monad $\& -$ and comonad $\oplus -$.

$$\frac{\vdash \Delta, A}{\vdash \Delta, \& A} \qquad \frac{\vdash \& \Delta, A}{\vdash \& \Delta, \oplus A} \qquad \frac{\vdash}{\vdash \& A} \qquad \frac{\vdash \& \Delta \qquad \vdash \& \Delta}{\vdash \& \Delta}$$

The resulting proof (and type) system provides a Curry-Howard interpretation of a realistic programming language with built-in internal non-determinism and failure.

Getting back to the presentation of our type system, we capture non-deterministic behavior in the type structure by operators $\& A$ and $\oplus A$ related by duality ($\overline{\& A} = \oplus \overline{A}$) and defined by the following rules:

$$\frac{P \vdash \Delta, x{:}A; \Theta}{x.\overline{\texttt{some}}; P \vdash \Delta, x{:}\& A; \Theta}\ (\mathrm{T}\&^{x}_{d}) \qquad \frac{}{x.\overline{\texttt{none}} \vdash x{:}\& A; \Theta}\ (\mathrm{T}\&^{x})$$

$$\frac{P \vdash \overline{w}{:}\& \Delta, x{:}A; \Theta}{x.\texttt{some}_{\overline{w}}; P \vdash \overline{w}{:}\& \Delta, x{:}\oplus A; \Theta}\ (\mathrm{T}\oplus^{x}_{\overline{w}}) \qquad \frac{P \vdash \& \Delta; \Theta \quad Q \vdash \& \Delta; \Theta}{P \oplus Q \vdash \& \Delta; \Theta}\ (\mathrm{T}\&)$$

Intuitively, $\& A$ is the type of a session that *may produce* a behavior of type A: this potential is made concrete in Rule ($\mathrm{T}\&^{x}_{d}$) where the behavior $x{:}A$ is indeed available (some), whereas Rule ($\mathrm{T}\&^{x}$) describes the case in which $x{:}A$ is not available (none). Dually, the type $\oplus A$ is the type of a session that *may consume* a behavior of type A. Rule ($\mathrm{T}\oplus^{x}_{\overline{w}}$) accounts for the possibility of not being able to consume an A by considering sessions different from x as potentially not available (i.e., abortable - cf. $\overline{w}{:}\& \Delta$ in the rule, where $\overline{w}$ denotes a sequence $w_1, \ldots, w_n$ of names). Rule (T&) expresses non-deterministic choice. While it may be seem to correspond to a formal sum of proofs (cf. [22]), in our case it corresponds exactly to non-deterministic choice $P \oplus Q$ of processes[3], and can only be used inside the monad $\& A$. The principal cut reductions are:

$$\begin{array}{l} \mathsf{Tcut}(x)(\mathrm{T}\&^{x}_{d}(D_1), \mathrm{T}\oplus^{x}_{\overline{w}}(D_2)) \to \mathsf{Tcut}(x)(D_1, D_2) \\ \mathsf{Tcut}(x)(\mathrm{T}\&^{x}, \mathrm{T}\oplus^{x}_{\overline{w}}(D_2)) \to \mathrm{T} \mid (\mathrm{T}\&^{w_1}, \cdots, \mathrm{T}\&^{w_i}) \end{array}$$

[3] We use $\oplus$ for denoting internal non-determinism in processes since this is rather standard; indeed, this notation goes back at least to De Nicola and Hennessy [19].

At the level of the process interpretation, these reduction rules are expressed by

$$\begin{array}{l}(\nu x)(x.\overline{\texttt{some}}; P \mid x.\texttt{some}_{\overline{w}}; Q) \to (\nu x)(P \mid Q)\\(\nu x)(x.\overline{\texttt{none}} \mid x.\texttt{some}_{\overline{w}}; Q) \to w_1.\overline{\texttt{none}} \mid \cdots \mid w_n.\overline{\texttt{none}}\end{array}$$

Notice how the reduction for $\overline{\texttt{none}}$ safely discards the continuation Q. We also consider the following proof conversion (and corresponding process congruence) that expresses the distribution of parallel composition over internal choice:

$$\begin{array}{l}\mathsf{Tcut}(x)(\mathrm{T}\&(D_1, D_2), D_3) \equiv \mathrm{T}\&(\mathsf{Tcut}(x)(D_1, D_3), \mathsf{Tcut}(x)(D_2, D_3))\\(\nu x)(P \mid (Q \oplus R)) \equiv (\nu x)(P \mid Q) \oplus (\nu x)(P \mid R)\end{array}$$

Notice that, in principle, the two computational reduction rules above could be formally used to express the reduction rules for the "sharing" exponentials (cf. [48]) in presentations of linear logic with explicit weakening and dereliction rules, instead of the DILL-style presentation we have adopted here. Indeed, we prefer the DILL-style presentation as it more tightly express the behavior of sharing present in traditional session types. On the other hand, together with the conversion principle just shown, the primitives and reduction rules just presented turn out to be quite adequate to express the behavior of non-determinism and failure.

Before closing the section, we discuss examples that use $\oplus A$ and $\& A$ types.

Example 2.5 (Movie Server (3)). Getting back to our movie server scenario we illustrate how to model a system with a client $\mathit{Randy}(s)$ that non-deterministically decides between either buying a movie or just seeing its trailer. Recalling process definitions for $\mathit{SBody}(s)$, $\mathit{Alice}(s)$, and $\mathit{Bob}(s)$ from Examples 2.3 and 2.4, we would have:

$$\begin{array}{r}\mathit{Randy}(s) \triangleq s.\overline{\texttt{some}}; \mathit{Alice}(s) \oplus s.\overline{\texttt{some}}; \mathit{Bob}(s)\\\mathit{USystem} \triangleq (\nu s)(s.\texttt{some}_{\emptyset}; \mathit{SBody}(s) \mid \mathit{Randy}(s))\end{array}$$

where the suitable types and type assignments are now given by

$$\mathit{Randy}(s) \vdash s : \&\overline{\mathit{SBT}}\ ;\ \cdot \qquad s.\texttt{some}_{\emptyset}; \mathit{SBody}(s) \vdash s : \oplus \mathit{SBT}\ ;\ \cdot$$

Process $\mathit{Randy}(s)$ is typed by using Rule (T$\&^s_d$) on each individual client; then, using Rule (T$\&$) one would obtain a typed non-deterministic choice between them. The server is typed using Rule (T$\oplus^s_{\overline{w}}$) with $\overline{w} = \emptyset$, for there are no sessions (besides s) in the linear context (recall that $\mathit{SBody}(s) \vdash s : \mathit{SBT}\ ; \cdot$). This way, we derive $\mathit{USystem} \vdash \cdot\ ;\ \cdot$. □

Interestingly, the non-deterministic choices enabled at the level of types by $\& A$ and $\oplus A$ (and at the process level by $\oplus$) are completely orthogonal to the usual deterministic choices enabled by labeled internal and external choices. The following example illustrates the pleasant interaction between deterministic and non-deterministic choices:

Example 2.6 (Movie Server (4)). Consider now a variant of the movie server that logs the request made by the client on a log service l of (boolean) type $\mathbf{B} = \mathbf{1} \oplus \mathbf{1}$. We extend the process $\mathit{SBody}(s)$ from Example 2.3 as follows:

$$\begin{array}{rl}\mathit{SBodyL}(s) \triangleq s.\texttt{case}(& s(\mathit{title}).s(\mathit{card}).s\langle \mathit{movie}\rangle.s.\overline{\texttt{close}} \mid l.\texttt{inl}; l.\overline{\texttt{close}},\\ & s(\mathit{title}).s\langle \mathit{trailer}\rangle.s.\overline{\texttt{close}} \mid l.\texttt{inr}; l.\overline{\texttt{close}})\end{array}$$

We may provide a typing $\mathit{SBodyL}(s) \vdash s{:}\mathit{SBT}, l{:}\mathbf{B}\ ;\cdot$ which cannot be composed with the non-deterministic client $\mathit{Randy}(s)$ from Example 2.5. However, process

$$s.\mathtt{some}_l; l.\overline{\mathtt{some}}; \mathit{SBodyL}(s)$$

may now be composed with client process $\mathit{Randy}(s)$ as

$$\mathit{ULSystem} \triangleq (\nu s)(s.\mathtt{some}; l.\overline{\mathtt{some}}; \mathit{SBodyL}(s) \mid \mathit{Randy}(s))$$

Now we may derive: $l.\overline{\mathtt{some}}; \mathit{SBodyL}(s) \vdash s{:}\mathit{SBT}, l{:}\&\mathbf{B}\ ;\cdot$ and

$$s.\mathtt{some}_l; l.\overline{\mathtt{some}}; \mathit{SBodyL}(s) \vdash s{:}\oplus\mathit{SBT}, l{:}\&\mathbf{B}\ ;\cdot \qquad \mathit{ULSystem} \vdash l{:}\&\mathbf{B}\ ;\cdot$$

Writing $P \Rightarrow Q$ to denote the reflexive-transitive closure of $P \to Q$, we obtain the reduction sequence $\mathit{ULSystem} \Rightarrow (l.\mathtt{inr}; l.\overline{\mathtt{close}} \oplus l.\mathtt{inl}; l.\overline{\mathtt{close}})$.

Notice that the visible behavior of log channel l in $\mathit{ULSystem}$ must be given the non-deterministic type $\&\mathbf{B}$: there is no typing $\mathit{ULSystem} \vdash l{:}\mathbf{B}$, since the resulting interaction is essentially non-deterministic. □

In our system, the ability of representing (internal) non-determinism is intrinsically tied to that of describing, in a completely logically motivated manner, abortable behaviors as typical of programming constructs such as exceptions and compensations [23]. Our following example illustrates this distinctive aspect of our model.

Example 2.7 (Movie Server (5)). To consider the possibility of modeling failure, we introduce the code for a "faulty" client, that non-deterministically behaves like $\mathit{Bob}(s)$ (cf. Example 2.4) or does not produce any behavior at all. Consider the non-deterministic server $\mathit{SBodyNDL}(s)$ from Example 2.6; we may now have:

$$\mathit{Buzz}(s) \triangleq s.\overline{\mathtt{some}}; \mathit{Bob}(s) \oplus s.\overline{\mathtt{none}} \qquad \mathit{Buzz}(s) \vdash s : \&\overline{\mathit{SBT}}\ ;\cdot$$
$$(\nu s)(\mathit{SBodyNDL}(s) \mid \mathit{Buzz}(s)) \vdash l{:}\&\mathbf{B}\ ;\cdot$$

Notice how failure of sub-computations propagates inside the monad $\&-$, encapsulated in a hereditarily safe way. Here, we have the reduction sequence

$$(\nu s)(\mathit{SBodyNDL}(s) \mid \mathit{Buzz}(s)) \Rightarrow (l.\overline{\mathtt{none}} \oplus l.\mathtt{inl}; l.\overline{\mathtt{close}})$$

reflecting that the composed system either aborts or chooses $l.\mathtt{inl}$ on the log. □

We now illustrate how systems encapsulating non-deterministic behavior can nevertheless be given a globally deterministic type, thus showing that internal non-determinism and failure are not visible as long as they are typed by "plain" deterministic types.

Example 2.8. Consider the following processes and typings:

$$\mathit{Some}(y) \triangleq y.\overline{\mathtt{some}}; y.\mathtt{inl}; y.\overline{\mathtt{close}} \oplus y.\overline{\mathtt{some}}; y.\mathtt{inr}; y.\overline{\mathtt{close}}$$
$$\mathit{Prod} \triangleq \overline{x}(y).(\mathit{Some}(y) \mid x.\mathtt{close}; b\langle\text{“}\mathit{done}\text{”}\rangle.b.\overline{\mathtt{close}})$$
$$\mathit{Cons} \triangleq x(u).(u.\mathtt{some}; u.\mathtt{case}(u.\mathtt{close}; \mathbf{0}, u.\mathtt{close}; \mathbf{0}) \mid x.\overline{\mathtt{close}})$$
$$\mathit{Blob} \triangleq (\nu x)(\mathit{Prod} \mid \mathit{Cons})$$
$$\mathit{Some}(y) \vdash y : \&\mathbf{B} \qquad \mathit{Prod} \vdash x : (\&\mathbf{B}) \otimes \bot, b : \mathit{Str} \otimes \mathbf{1} \qquad \mathit{Cons} \vdash x{:}(\oplus\overline{\mathbf{B}}) \parr \mathbf{1}$$

(Processes)

$$\begin{array}{rl} P ::= & [x \leftrightarrow y] \mid P|Q \mid (\nu y)P \mid \mathbf{0} \\ & \mid \overline{x}(y).P \mid x(y).P \mid \overline{x}?(y).P \mid !x(y).P \\ & \mid x.\texttt{case}(P,Q) \mid x.\texttt{inr};P \mid x.\texttt{inl};P \mid x.\overline{\texttt{close}} \mid x.\texttt{close};P \\ & \mid P \oplus Q \mid x.\overline{\texttt{some}};P \mid x.\overline{\texttt{none}};P \mid x.\texttt{some}_{\overline{w}};P \end{array}$$

(Reduction - Contextual Congruence Rules omitted)

$$\begin{array}{ll} \overline{x}(y).Q \mid x(y).P \to (\nu y)(Q \mid P) & \overline{x}?(y).Q \mid !x(y).P \to (\nu y)(Q \mid P) \mid !x(y).P \\ x.\texttt{inr};P \mid x.\texttt{case}(Q,R) \to P \mid R & x.\texttt{inl};P \mid x.\texttt{case}(Q,R) \to P \mid Q \\ (\nu x)([x \leftrightarrow y] \mid P) \to P\{y/x\} & x.\overline{\texttt{some}};P \mid x.\texttt{some}_{\overline{w}};Q \to P \mid Q \\ x.\overline{\texttt{close}} \mid x.\texttt{close};P \to P & x.\overline{\texttt{none}} \mid x.\texttt{some}_{\overline{w}};Q \to w_1.\overline{\texttt{none}} \mid \ldots \mid w_n.\overline{\texttt{none}} \\ P \to Q \Rightarrow (\nu y)P \to (\nu y)Q & P \equiv P', P' \to Q', Q' \equiv Q \Rightarrow P \to Q \\ Q \to Q' \Rightarrow P \mid Q \to P \mid Q' & Q \to Q' \Rightarrow P \oplus Q \to P \oplus Q' \end{array}$$

(Structural Congruence - Contextual Congruence Rules omitted)

$$\begin{array}{c} P \mid \mathbf{0} \equiv P \qquad P \equiv_\alpha Q \Rightarrow P \equiv Q \qquad (\nu x)\mathbf{0} \equiv \mathbf{0} \qquad P \mid Q \equiv Q \mid P \\ P \mid (Q \mid R) \equiv (P \mid Q) \mid R \qquad x \notin \mathit{fn}(P) \Rightarrow P \mid (\nu x)Q \equiv (\nu x)(P \mid Q) \\ [x \leftrightarrow y] \equiv [y \leftrightarrow x] \qquad (\nu x)(\nu y)P \equiv (\nu y)(\nu x)P \qquad \mathbf{0} \oplus \mathbf{0} \equiv \mathbf{0} \qquad P \oplus Q \equiv Q \oplus P \\ P \oplus (Q \oplus R) \equiv (P \oplus Q) \oplus R \qquad (\nu x)(P \mid (Q \oplus R)) \equiv (\nu x)(P \mid Q) \oplus (\nu x)(P \mid R) \end{array}$$

Fig. 3. The Process Language.

Notice that the although the producer process Prod sends a non-deterministic boolean to the consumer process Cons, the type of the composed system Blob is $b : \mathit{Str} \otimes \mathbf{1}$, a deterministic type. In fact, we may easily verify that $\mathit{Blob} \Rightarrow b\langle\text{“}\mathit{done}\text{”}\rangle.b.\overline{\texttt{close}}$. □

Figure 3 summarizes our process language, and associated reduction and structural congruence relations. The main properties of our system will be established next.

3 Main Results

We collect in this section main sanity results for our non-deterministic linear logic-based type system for session process behavior. First, our system enjoys the cut-elimination property. Cut elimination may be derived given a suitable congruence $\cong_s$ on processes consisting of reduction (computational conversions), structural congruence (structural conversions), and some key commuting conversions (cf.[11,37,13]).

Theorem 3.1 (Cut Elimination). *If $P \vdash \Delta; \Theta$ then there is a process Q such that $P \cong_s Q$ and $Q \vdash \Delta; \Theta$ is derivable without using rules ($T\mathsf{cut}$) and ($T\mathsf{cut}^?$).*

The proof is an extension of the proof for classical linear logic with mix, but considering the new reductions and conversions introduced above for revealing and reducing principal cuts involving the $\&A$ and $\oplus A$ modalities.

Then, we may state type safety, witnessed by theorems of type preservation and global progress for closed systems. Type preservation states that the observable interface of a system is invariant under reduction.

Theorem 3.2 (Type Preservation). *If $P \vdash \Delta; \Theta$ and $P \to Q$ then $Q \vdash \Delta; \Theta$.*

Proof. (Sketch) By induction on typing derivations, and case analysis on reduction steps. In each case, the result easily follows, given that reductions come from well-defined proof conversions, which by construction preserve typing. ■

Unlike standard type systems for session types, our logical interpretation satisfies *global process*, meaning that well-typed processes never get stuck on pending linear communications. More precisely, we say that a process P is *live*, noted $live(P)$, if and only if $P \equiv C[\pi.Q]$ where $C[-]$ is a static context (e.g. a process term context in which the hole is not behind an action prefix, but only under parallel composition $- \mid -$, name restriction $(\nu x)-$, or sum $-\oplus-$ operators) and $\pi.Q$ is not a replicated process (i.e., π is a session input, output, offer, choice, or non-deterministic action). We then have:

Theorem 3.3 (Progress). *If $P \vdash ; \Theta$ and $live(P)$ then there is Q such that $P \to Q$.*

Proof. (Sketch) By induction on the typing derivation. Our proof relies on a contextual progress lemma, which uses a labeled transition system for processes, compatible with reduction (cf. [13]). This lemma yields a more general progress property for processes with free linear channels that transition by means of immediate external interactions. It extends Lemma 4.3 in [13] (which holds for a language without non-determinism) as follows: If $P \vdash \Delta; \Theta$ and $live(P)$ then either (1) there is Q such that $P \to Q$ or (2) there are P_i $(i = 1..n)$ such that $P \equiv \oplus P_i$ and for all P_i there exist Q_i and α such that $P_i \xrightarrow{\alpha} Q_i$. The proof of this extended lemma is by induction on derivations. ■

We now discuss additional results that clarify some key features of the our type system. We say that a process P is *prime* if it is not structurally congruent to a process of the form $Q \oplus R$ with non-trivial (i.e., equivalent to $\mathbf{0}$) Q and R. We can then prove:

Proposition 3.4. *Let $P \vdash \Delta; \Theta$ where types in $\Delta; \Theta$ are deterministic (do not contain $\& A$ or $\oplus A$ types at the top level), and let $P \Rightarrow Q \not\to$. Then Q is prime.*

Proof. (Sketch) By induction on the typing derivation. ■

Based on a logical system in which reduction matches cut-elimination, it turns out that typing in our system enforces confluence and also strong normalization. These results can be established using (linear) logical relations, as developed in [37]. Intuitively, confluence holds because non-determinism is captured equationally without losing information, by means of delaying choice in processes $P \oplus Q$, which express sets of alternative states. Still, it is interesting to relate our system with standard process calculi which explicitly commit non-deterministic states into alternative components. For that purpose, we investigate the extension of the reduction relation in Figure 3 with non-confluent rules for internal choice, standard in process calculi but clearly incompatible with any Curry-Howard interpretation, namely $P \oplus Q \to P$ and $P \oplus Q \to Q$. We denote by $P \to_c Q$ (and $P \Rightarrow_c Q$) the extended reduction relation, which can be proven to still satisfy preservation and progress in the sense of Theorems 3.2 and 3.3. We may then show the following property, expressing postponing of internal non-deterministic collapse of non-deterministic states into prime states.

Theorem 3.5 (Postponing). *Let $P \vdash \Delta; \Theta$. We have*

1. If $P \Rightarrow P_1 \oplus \ldots \oplus P_n \not\Rightarrow$ with P_i prime for all i, then $P \Rightarrow_c P_i$ for all $0 < i \leq n$.

2. Let $\mathcal{C} = \{P_i \mid P \Rightarrow_c P_i \not\Rightarrow_c \text{ and } P_i \text{ is prime } \}$. Then $\mathcal{C}$ is finite up to $\equiv$, with $\#\mathcal{C} = n$, and for all $0 < i \leq n$, $P \Rightarrow P_1 \oplus \ldots \oplus P_n \rightarrow_c P_i$.

Proof. 1. Trivial by definition. 2. By induction on the reduction sequence, using the fact that we may commute $\Rightarrow$ reduction steps backwards with $\Rightarrow_c$ reduction steps. ■

Theorem 3.5(2) shows that no information is lost by $\Rightarrow$ with respect to the (standard) non-deterministic (and non-confluent) semantics of internal choice $P \oplus Q$ expressed by $\Rightarrow_c$. We may therefore tightly relate our system, based on a logically motivated reduction relation, with a standard non-confluent reduction relation including rules for internal choice, in the sense that the former precisely captures the multiset of observable alternatives defined by the latter, while preserving compositional and equational reasoning about system behavior as expected from a Curry-Howard interpretation.

4 Higher-Order Concurrency, Non Determinism, and Exceptions

We illustrate the expressive power of our typed process model by embedding λ^{exc}, a linear higher-order functional, concurrent programming language with concurrency, non-determinism—including failure—, and exceptions. Defined by a typed compositional encoding, this embedding allows us to showcase the generality of our developments and the relevance of our Curry-Howard correspondence in a broader setting; it will also enable us to give a rigorous footing to our motivating examples (cf. Figures 1 and 2).

The Target Calculus. λ^{exc} is a typed call-by-value functional calculus, defined by the grammar below. We use $e, e', \ldots$ to range over expressions; $v, v', \ldots$ to range over values; $x, y, z, \ldots$ to range over variables; $c, c', \ldots$ to range over channels, and T, U, A, B to range over types. The syntax of values, expressions, and types (T) is as follows:

$$\begin{array}{rl}
v ::= & x \mid * \mid \lambda z.e \mid \langle\!\langle v \rangle\!\rangle \\
e ::= & v \mid (f\,x) \mid \mathtt{LET}\,a = e_1\,\mathtt{IN}\,e_2 \\
\mid & \mathtt{TRY}\,e_1\,\mathtt{CATCH}\,z.\,e_2 \mid \mathtt{THROW}\,z \\
\mid & \mathtt{LIFT}\,e \mid \mathtt{SOME}\,!\,z; e \mid \mathtt{SOME}\,?\,z; e \mid \mathtt{NONE}\,!\,z; e \mid e_1 \oplus e_2 \\
\mid & \mathtt{FORK}\,c.e \mid \mathtt{SEND}(c, e_1); e_2 \mid \mathtt{RECV}(c, z); e \mid \mathtt{CLOSE}\,!\,c\,; e \mid \mathtt{CLOSE}\,?\,c \\
T ::= & \mathbf{unit} \mid A \xrightarrow{T} B \mid A \xrightarrow{0} B \mid !T.T' \mid ?T.T' \mid \mathsf{end}_! \mid \mathsf{end}_? \mid \oplus T \mid \& T
\end{array}$$

We say expressions are *effectful* if they can raise an exception, and *pure* otherwise. Besides the unit type, types for λ^{exc} include (linear) arrow types of two forms: $A \xrightarrow{0} B$ is the type of functions that do not raise exceptions, whereas $A \xrightarrow{T} B$ is the type of functions that may raise an exception of type T. We also have session types $!T.T'$ and $?T.T'$ for output and input channel-based communication. Types for labeled selection and choice are not included but can be easily accommodated. Types $\mathsf{end}_!$ and $\mathsf{end}_?$ denote the dual views of terminated endpoints. Furthermore, we have types $\oplus T$ and

$\&T$ for expressions that may produce and consume values of a type T, respectively. We write S, S' to denote the session fragment of the type structure (i.e., no unit nor arrow types). On this fragment, we assume a duality relation, denoted $\overline{S}$, defined as expected. The type syntax does not include general (non-linear) functional values nor shared sessions; the integration of these constructs is orthogonal and unsurprising.

As values, we consider variables, abstractions, and the unit value $*$; we also have the *abortable value* $\langle\!\langle v \rangle\!\rangle$, which represents discardable (affine) values: given a value v of type T, value $\langle\!\langle v \rangle\!\rangle$ will be of type $\oplus T$. For convenience, the language is let-expanded; as a result, application is of the form $(f\, x)$, for variables f and x. Expressions also include a try-catch construct for scoped exceptions, with the expected meaning, and a construct for raising/throwing exceptions with an explicit value. The key features of the process model in § 2 appear as expressions that may produce a value of a certain type and one construct that may consume a value of a certain type. Non-deterministic choices between two expressions are also supported. Concurrency is enabled by spawning threads, using a forking construct. Moreover, λ^{exc} includes expressions for channel-based communication, enabling the exchange of values of any type (including channels).

As mentioned above, the intended operational model for λ^{exc} is call-by-value; rather than directly giving the operational semantics for the language, we first delineate its behavior via a type system and then give its semantics indirectly, via a type respecting encoding into the basic type system introduced in § 2.

The type system we consider here is actually a type-and-effect system in which the effect represents the type of the exception that can be raised by the typed expression. Judgments are then the form $\mathsf{D} \vdash^{U} e : T$: under an environment D (a set of typing assignments), the expression e has return type T, while the effect type U is either $\mathsf{0}$ (the expression is pure) or T (the expected type of exceptions).

The typing rules for λ^{exc} are shown in Figure 4. Rule (ABS) types abstractions; it decrees that the type of the exception possibly raised by the abstraction body will be used as the effect associated to the arrow type. Rule (PRO) types abortable values $\langle\!\langle v \rangle\!\rangle$, as motivated earlier: it closely follows the principles of Rule (T$\oplus\frac{x}{w}$) for session-based processes; in particular, it requires all free variables in v to be abortable (cf. the premise $\oplus\mathsf{D}$). Rule (LIFT) allows to cast an (trivially) effecful expression from a pure one.

There are three typing rules for let expressions $\mathtt{LET}\, a = e_1\, \mathtt{IN}\, e_2$; the actual rule used depends on the exceptions possibly raised by its constituent sub-expressions e_1 and e_2. Rule (LET1) is used when both e_1 and e_2 are effectful. Observe that e_2 must be typable in an abortable environment, in order to safely account for an exception raised in e_1. Rule (LET2) handles the case in which both e_1 and e_2 are pure, while Rule (LET3) covers the case in which only e_1 is pure. These three typing rules are crucial to isolate effects and to exploit the combination of pure with effectful computations.

Rule (TRY) types the construct $\mathtt{TRY}\, e_1\, \mathtt{CATCH}\, x.\, e_2$; the type of the exception possibly raised by e_1 must match with the type of x in e_2. Notice that e_1 and e_2 must be of the same type (T in the rule). Rule (THROW) ensures that the type of the thrown value is propagated as an effect. Rule (FORK) captures the essence of thread spawning for communication types, creating a new (linear) session channel where one endpoint is handed to the thread body and the other endpoint returned by the fork operation.

$$\text{VAR}\ \frac{}{x : T \vdash^{U} x : T} \qquad \text{ABS}\ \frac{\mathsf{D}, x : A \vdash^{U} e : B}{\mathsf{D} \vdash^{U'} \lambda x.e : A \xrightarrow{U} B} \qquad \text{APP}\ \frac{\mathsf{D} \vdash^{U} f : A \xrightarrow{U} B \quad \mathsf{D}' \vdash^{V} a : A}{\mathsf{D}, \mathsf{D}' \vdash^{U} (f\, a) : B} \qquad \text{UNIT}\ \frac{}{\vdash^{U} * : \mathbf{unit}}$$

$$\text{PRO}\ \frac{\oplus\mathsf{D} \vdash^{0} v : T}{\oplus\mathsf{D} \vdash^{0} \langle\!\langle v \rangle\!\rangle : \oplus T} \qquad \text{LIFT}\ \frac{\mathsf{D} \vdash^{0} e : B}{\mathsf{D} \vdash^{T} \mathtt{LIFT}\, e : B} \qquad \text{LET1}\ \frac{\mathsf{D}_1 \vdash^{T} e_1 : A \qquad \oplus\mathsf{D}_2, a{:}A \vdash^{T} e_2 : B}{\mathsf{D}_1, \oplus\mathsf{D}_2 \vdash^{T} \mathtt{LET}\, a = e_1\, \mathtt{IN}\, e_2 : B}$$

$$\text{LET2}\ \frac{\mathsf{D}_1 \vdash^{0} e_1 : A \qquad \mathsf{D}_2, a{:}A \vdash^{0} e_2 : B}{\mathsf{D}_1, \mathsf{D}_2 \vdash^{0} \mathtt{LET}\, a = e_1\, \mathtt{IN}\, e_2 : B} \qquad \text{LET3}\ \frac{\mathsf{D}_1 \vdash^{0} e_1 : A \qquad \mathsf{D}_2, a{:}A \vdash^{T} e_2{:}B}{\mathsf{D}_1, \mathsf{D}_2 \vdash^{T} \mathtt{LET}\, a = e_1\, \mathtt{IN}\, e_2 : B}$$

$$\text{TRY}\ \frac{\mathsf{D}_1 \vdash^{U} e_1 : T \qquad x : U \vdash^{0} e_2 : T}{\mathsf{D}_1 \vdash^{0} \mathtt{TRY}\, e_1\ \mathtt{CATCH}\, x.e_2\ : T} \qquad \text{THROW}\ \frac{\mathsf{D} \vdash^{0} z : T}{\mathsf{D} \vdash^{T} \mathtt{THROW}\, z : U} \qquad \text{FORK}\ \frac{\mathsf{D}, x : \overline{S} \vdash^{0} e : \mathbf{unit}}{\mathsf{D} \vdash^{0} \mathtt{FORK}\, x.e : S}$$

$$\text{CLOSE1}\ \frac{}{c : \mathsf{end}_{?} \vdash^{T} \mathtt{CLOSE}\,?\,c : \mathbf{unit}} \qquad \text{CLOSE2}\ \frac{\mathsf{D} \vdash^{T} e : \mathbf{unit}}{\mathsf{D}, c : \mathsf{end}_{!} \vdash^{T} \mathtt{CLOSE}\,!\,c\,;e : \mathbf{unit}}$$

$$\text{SEND}\ \frac{\mathsf{D} \vdash^{0} a : T_1 \qquad \mathsf{D}', c : S \vdash^{U} e : T}{\mathsf{D}, \mathsf{D}', c : !T_1.S \vdash^{U} \mathtt{SEND}(c, a); e : T} \qquad \text{RECV}\ \frac{\mathsf{D}, z : T_1, c : S \vdash^{U} e : T}{\mathsf{D}, c : ?T_1.S \vdash^{U} \mathtt{RECV}(c, z); e : T}$$

$$\text{SOME1}\ \frac{\mathsf{D}, z : T_1 \vdash^{U} e : T}{\mathsf{D}, z : \oplus T_1 \vdash^{U} \mathtt{SOME}\,!\,z; e : T} \qquad \text{SOME2}\ \frac{\oplus\mathsf{D}, z : T \vdash^{0} e : \mathbf{unit}}{\oplus\mathsf{D}, z : \& T \vdash^{0} \mathtt{SOME}\,?\,z; e : \mathbf{unit}}$$

$$\text{NONE}\ \frac{\mathsf{D} \vdash^{U} e : T}{\mathsf{D}, z : \oplus T_1 \vdash^{U} \mathtt{NONE}\,!\,z; e : T} \qquad \text{NONDET}\ \frac{\oplus\mathsf{D} \vdash^{0} e_1 : \mathbf{unit} \quad \oplus\mathsf{D} \vdash^{0} e_2 : \mathbf{unit}}{\oplus\mathsf{D} \vdash^{0} e_1 \oplus e_2 : \mathbf{unit}}$$

Fig. 4. Typing rules for λ^{exc}.

Rules (CLOSE1) and (CLOSE2) type session channel closing operations; Rules (SEND) and (RECV) type operations for sending and receiving values along session channels.

Rule (SOME1) and Rule (SOME2) type the production and consumption of a non-deterministic value as z, respectively. In particular, Rule (SOME2) applies to expressions that do not return values, but that may interact with expressions that do return values via channel-based communication. Notice the similarities between Rules (SOME1) and (SOME2) (for functional expressions) and Rules (T$\&_{d}^{x}$) and (T$\oplus\frac{x}{w}$) (for process terms), respectively. In the same vein, Rule (NONE) can be seen as the analogue of Rule (T$\&^{x}$) but for abortable expressions in our functional language. Rule (NONDET) enables the non-deterministic choice between two pure expressions that do not return values; this allows us to define, e.g., non-deterministic sessions.

In general, the (two-sided) typing rules in Figure 4 encompass a notion of duality, in the sense that a connective appearing in the left-hand side of the turnstile in the Figure 4 corresponds to its dual in the right-hand side of the turnstile. This intuition will be captured in our embedding of functional expressions as processes, detailed next.

$$
\begin{aligned}
[\![\mathbf{unit}]\!] &= \mathbf{1}\\
[\![A \xrightarrow{\mathsf{0}} B]\!] &= \big(\overline{[\![A]\!]} \parr ((\uparrow[\![B]\!]) \parr \bot)\big)\\
[\![A \xrightarrow{U} B]\!] &= \big(\overline{[\![A]\!]} \parr ((\&\big([\![B]\!] \div \overline{[\![U]\!]}\big)) \parr (([\![B]\!] \boxplus [\![U]\!]) \parr \bot))\big)\\
[\![!T_1.T_2]\!] &= [\![T_1]\!] \otimes \overline{[\![T_2]\!]} \qquad [\![?T_1.T_2]\!] = [\![T_1]\!] \parr [\![T_2]\!]\\
[\![\oplus T]\!] &= \oplus[\![T]\!] \qquad [\![\& T]\!] = \&[\![T]\!] \qquad [\![\mathsf{end}_{!}]\!] = \bot \qquad [\![\mathsf{end}_{?}]\!] = \mathbf{1}
\end{aligned}
$$

Fig. 5. Encoding of λ^{exc} types into logical propositions.

Example 4.1. We can now return to the code snippet in Fig. 2 and give some typings using the type structure just introduced. As mentioned in the introduction, there is a precise stage of the protocol along (dual) names res and f after which failure is safe. In our type structure we can precisely delineate such a place. We would have:

$$
\begin{array}{ll}
l : ?\mathtt{string}.?\mathtt{string}.\mathsf{end}_{?} & log : !\mathtt{string}.!\mathtt{string}.\mathsf{end}_{!}\\
f : ?\mathtt{string}.!\mathtt{int}.\oplus(?\mathtt{string}.\mathsf{end}_{?}) & res : !\mathtt{string}.?\mathtt{int}.\&(!\mathtt{string}.\mathsf{end}_{!})
\end{array}
$$

These typings require minor modifications in the code of Fig. 2: we add prefix '$\mathtt{SOME}\,?\,f$' before '$\mathtt{RECV}(f, bk)$', and prefix '$\mathtt{SOME}\,!\,res$' before '$\mathtt{SEND}(res, book)$'.

Embedding λ^{exc} Into Session Typed Processes. We now present a typeful encoding of λ^{exc} into the logically motivated typed process model of § 2, and establish its correctness (Theorem 4.2). The encoding has two main components: the encoding of (functional) types into linear logic based session types, and the encoding of λ^{exc} expressions into (non-deterministic) concurrent processes.

Figure 5 gives the encoding of types. We use the following shorthand notations:

$$\uparrow[\![T]\!] \triangleq [\![T]\!] \otimes \mathbf{1} \tag{1}$$

$$[\![U]\!] \boxplus [\![T]\!] \triangleq ([\![U]\!] \otimes \mathbf{1}) \oplus ([\![T]\!] \otimes \mathbf{1}) \tag{2}$$

$$\&\big([\![T]\!] \div \overline{[\![U]\!]}\big) \triangleq \&([\![T]\!] \otimes ((\overline{[\![U]\!] \boxplus [\![T]\!]}) \otimes \mathbf{1})) \tag{3}$$

Also, we assume the expected extension of the encoding of types to typing environments: given $\mathsf{D} = x_1{:}T_1, \cdots, x_n{:}T_n$ then $[\![\mathsf{D}]\!] = x_1{:}[\![T_1]\!], \cdots, x_n{:}[\![T_n]\!]$.

The encoding of expressions is typeful: for each typing rule in Figure 4 we give a corresponding type derivation for session-typed processes. Figures 6 and 7 give a complete account; for readability, in those figures we show only the conclusion (final judgment) in the derivation. Also, we use the following abbreviations for processes:

- we write $y\langle z\rangle.P$ (where z is free in P) for the free output process, represented as $\overline{y}(w).([w \leftrightarrow z] \mid P)$ (cf. § 2);
- we write $y.\mathsf{0}; P$ and $y.\overline{\mathsf{0}}$ to stand for $y.\mathtt{close}; P$ and $y.\overline{\mathtt{close}}$, respectively;
- we define S_q as the process $q(u).q.\mathsf{0}; u.\mathsf{0}; \mathbf{0}$. Notice that $\mathsf{S}_q \vdash q : \uparrow[\![\mathbf{unit}]\!]$.

As usual in encodings of (call-by-value) functional languages into the π-calculus, our encoding of expressions is indexed by names, which are used to interact with the environment; they can be seen as *continuations* or as *locations* where the value returned by an expression will be made available. In our case, these names are related to the effects of the source expression e:

- If e is pure then its encoding will be indexed by a single continuation name y. This will be denoted $\llbracket e \rrbracket_y$.
- If e is effectful then its encoding will be indexed by names y and x. This will be denoted $\llbracket e \rrbracket_{y,x}$: name y represents an *non-deterministic* continuation, along which the value to which e reduces *may be produced*; name x represents the continuation to the enclosing try-catch block exception handler.

These intuitive distinctions are made precise in our main technical result, which exploits the shorthand notations (1), (2), and (3) above:

Theorem 4.2 (Typability). *Suppose* $\mathsf{D} \vdash^{U} e : T$. *Then, for some names* y, x, *we have:*

- $\llbracket e \rrbracket_y \vdash \overline{\llbracket \mathsf{D} \rrbracket}, y{:} \uparrow \llbracket T \rrbracket$, *if* $U = \mathsf{0}$.
- $\llbracket e \rrbracket_{y,x} \vdash \overline{\llbracket \mathsf{D} \rrbracket}, y{:}\&\big(\llbracket T \rrbracket \div \overline{\llbracket U \rrbracket}\big), x{:}\llbracket U \rrbracket \boxplus \llbracket T \rrbracket$, *if* $U \neq \mathsf{0}$.

Consequently, our source language λ^{exc} (which combines functions, concurrency, nondeterminism, and exceptions) will inherit key guarantees from the target process language, namely preservation and global progress (deadlock absence and lock-freedom).

Due to space limitations, in the following we only discuss selected cases of Figures 6 and 7. As already mentioned, the type of a let expression $\mathtt{LET}\, a = e_1 \,\mathtt{IN}\, e_2$ considers different possibilities for the interplay of pure and effectful computations in e_1 and e_2. If both expressions are pure (cf. Rule (LET2)) then the encoding is simple:

$$\llbracket \mathtt{LET}\, a = e_1 \,\mathtt{IN}\, e_2 \rrbracket_y = (\nu q)(\llbracket e_1 \rrbracket_q \mid q(a).q.\mathsf{0}; \llbracket e_2 \rrbracket_y)$$

Since e_1 is pure, we know $\llbracket e_1 \rrbracket_q$ will surely produce a value, which will be made available to $\llbracket e_2 \rrbracket_y$ along the private (linear) name q. The case in which both e_1 and e_2 may raise exceptions (cf. Rule (LET1)) is more interesting:

$$\llbracket \mathtt{LET}\, a = e_1 \,\mathtt{IN}\, e_2 \rrbracket_{y,x} = (\nu q)(\llbracket e_1 \rrbracket_{q,x} \mid q.\mathtt{some}_D; q(a).q(s).q.\mathsf{0}; \llbracket e_2 \rrbracket_{y,s})$$

In this case, since e_1 may raise an exception, we account for this possibility via the prefix $q.\mathtt{some}_D$, which requires all values (including sessions) in $\llbracket e_2 \rrbracket_{y,s}$ (excepting a) to be in abortable state. The production of a value within $\llbracket e_1 \rrbracket_{q,x}$ will be signaled by a prefix $q.\overline{\mathtt{some}}$, while throwing of an exception will be signaled by a prefix $q.\overline{\mathtt{none}}$ (see next). Therefore, if $\llbracket e_1 \rrbracket_{q,x}$ produces a value ($q.\overline{\mathtt{some}}$ is executed) then this value will be passed to $\llbracket e_2 \rrbracket_{q,s}$ using the private name q; subsequently, the reference to the enclosing try-block x will also be passed to $\llbracket e_2 \rrbracket_{q,s}$ as parameter s, exploiting linearity of name-passing (delegation). Otherwise, if $\llbracket e_1 \rrbracket_{q,x}$ ever raises an exception ($q.\overline{\mathtt{none}}$ is executed) then all the values in D will be safely and hereditarily discarded.

The encoding of values takes into account that a value may occur in an abortable context. The encoding of a variable z of type T is as follows:

$$\llbracket z \rrbracket_y = y\langle z \rangle.y.\overline{\mathsf{0}} \qquad\qquad \llbracket z \rrbracket_{y,x} = y.\overline{\mathtt{some}}; y\langle z \rangle.y\langle x \rangle.y.\overline{\mathsf{0}}$$

If z occurs in a pure context/expression, then its encoding, given on the left, is standard; name y will have type $\uparrow \llbracket T \rrbracket$ (cf. (1)). Otherwise, if z occurs in an effectful (abortable) context, then its encoding, given on the right, first announces the production of a value using prefix $y.\overline{\mathtt{some}}$; after z is sent along y, name x (representing the continuation of the enclosing try-catch block exception handler) will be sent along y. The type of x will be $\llbracket U \rrbracket \boxplus \llbracket T \rrbracket$, where U is the type of the enclosing exception (cf. (2)). Thus, intuitively, x encompasses the potential for a normal execution ($\llbracket T \rrbracket$) but also contains information on the (exceptional) behavior to be triggered upon failure ($\llbracket U \rrbracket$). A more concrete justification for the typing $x{:}\llbracket U \rrbracket \boxplus \llbracket T \rrbracket$ will become apparent next, when discussing the deterministic choice that underlies the encoding of try-catch and throw expressions.

To encode an abstraction $\lambda z.e$, we distinguish several cases, depending on whether e and $\lambda z.e$ are effectful or not. The simplest case is when both e and $\lambda z.e$ are pure:

$$\llbracket \lambda z.e \rrbracket_y = \overline{y}(f).(y.\overline{\mathtt{0}} \mid f(z).f(k).f.\mathtt{0}; \llbracket e \rrbracket_k)$$

We follow closely known encodings of λ-calculus in the π-calculus, here adapted to a linear setting in which the continuation y and the reference to the function body f are session-typed [43]. When both $\lambda z.e$ and e are effectful we follow a similar principle:

$$\llbracket \lambda z.e \rrbracket_{y,x} = y.\overline{\mathtt{some}}; \overline{y}(f).(y\langle x \rangle.y.\overline{\mathtt{0}} \mid f(z).f(k).f(j).f.\mathtt{0}; \llbracket e \rrbracket_{k,j})$$

The prefix $y.\overline{\mathtt{some}}$ declares the production of a value, namely the reference to the function body f. An invocation to f must supply the parameter of the function (z) but also the continuations k and j, to be linearly used by the encoding $\llbracket e \rrbracket_{k,j}$.

The encoding of applications goes hand in hand with the encoding of let expressions. Given the let-expanded semantics (which forces an expression's context to deal with potentially abortable expressions), the encoding of applications $(f\ a)$ is simple:

$$\llbracket (f\ a) \rrbracket_{y,x} = f\langle a \rangle.f\langle y \rangle.f\langle x \rangle.f.\overline{\mathtt{0}}$$

We may now discuss the encodings of try-catch and throw expressions:

$$\begin{aligned}\llbracket \mathtt{TRY}\, e_1\, \mathtt{CATCH}\, z.\, e_2 \rrbracket_y = (\nu j)((\nu k)(\llbracket e_1 \rrbracket_{k,j} \mid \underbrace{k.\mathtt{some}_{\emptyset}; k(u).k(z).k.\mathtt{0}; z.\mathtt{inl}; z\langle u \rangle.z.\overline{\mathtt{0}})}_{\text{(I)}} \mid \\ \underbrace{j.\mathtt{case}(j(u).j.\mathtt{0}; y\langle u \rangle.y.\overline{\mathtt{0}}\,,\ j(z).j.\mathtt{0}; \llbracket e_2 \rrbracket_y)}_{\text{(II)}})\end{aligned}$$

$$\llbracket \mathtt{THROW}\, z \rrbracket_{y,x} = y.\overline{\mathtt{none}} \mid x.\mathtt{inr}; x\langle z \rangle.x.\overline{\mathtt{0}}$$

The encoding of $\mathtt{TRY}\, e_1\, \mathtt{CATCH}\, z.\, e_2$ is in two parts, denoted (I) and (II) above. Part (I) concerns normal behaviors only; Part (II) concerns normal and exceptional behaviors:

- If e_1 does not raise an exception then $\llbracket e_1 \rrbracket_{k,j}$ will trigger a prefix $k.\overline{\mathtt{some}}$, which will synchronize with Part (I). Subsequently, the obtained value and the reference to the enclosing exception block will be passed around; in this case, z will be substituted by j, and the prefix $j.\mathtt{inl}$ will synchronize with the choice on j (Part (II)) to send the resulting value along y. This choice discards the right branch containing $\llbracket e_2 \rrbracket_y$.
- If e_1 raises an exception then, because of the encoding of throw, process $\llbracket e_1 \rrbracket_{k,j}$ will trigger a prefix $k.\overline{\mathtt{none}}$ which will synchronize with Part (I). As a result, the remaining behavior on k and z will be discarded. However, the choice on j (Part (II)) will continue to be available: this is used by the encoding of throw, which by executing $j.\mathtt{inr}$ will select the right branch of Part (II). The value raised by the exception will be then passed to $\llbracket e_2 \rrbracket_y$, which can now be executed.

Our encoding of try-catch therefore elegantly amalgamates the key features of our process model: most notably, the presence of abortable behaviors in a pleasant coexistence with non-abortable behaviors, and the interplay between non-deterministic and deterministic choices—indeed, it is the deterministic choice that underlies the exception mechanism what ultimately justifies the type $\llbracket U \rrbracket \boxplus \llbracket T \rrbracket$ for x, given in (2).

In the typed model presented here (and its encoding into processes), we consider try-catch constructs $\texttt{TRY}\, e_1\, \texttt{CATCH}\, z.\, e_2$ in which e_2 is pure (cf. Fig. 4). However, there is no fundamental obstacle to address the general case in which both e_1 and e_2 may raise exceptions; the encoding given in Fig. 7 can be extended following expected lines.

Constructs for non-deterministic behaviors have fairly straightforward encodings:

$$\llbracket \texttt{SOME}\,!\, z; e \rrbracket_{y,x} = z.\overline{\texttt{some}}; \llbracket e \rrbracket_{y,x} \qquad\qquad \llbracket \texttt{NONE}\,!\, z; e \rrbracket_{y,x} = z.\overline{\texttt{none}} \mid \llbracket e \rrbracket_{y,x}$$
$$\llbracket \texttt{SOME}\,?\, z; e \rrbracket_{y,x} = z.\texttt{some}_D; (\nu q)(\llbracket e \rrbracket_q \mid \mathsf{S}_q) \mid \overline{y}(v).(v.\overline{\mathsf{0}} \mid y.\overline{\mathsf{0}})$$

In $\llbracket \texttt{SOME}\,?\, z; e \rrbracket_{y,x}$, notice that typing ensures that e does not return a value; also, set D enables to safely discard behaviors in e in the event of an exception. Given the conditions ensured by typing, the encoding of non-deterministic choices is unsurprising:

$$\llbracket e_1 \oplus e_2 \rrbracket_y = ((\nu z)(\llbracket e_1 \rrbracket_z \mid \mathsf{S}_z) \oplus (\nu z)(\llbracket e_2 \rrbracket_z \mid \mathsf{S}_z)) \mid \llbracket * \rrbracket_y$$

In essence, processes S_z consume the (unit) value produced by $\llbracket e_1 \rrbracket_y$ and $\llbracket e_2 \rrbracket_y$ through z. The resulting processes can then be composed first in a non-deterministic choice, and then in an independent parallel composition with $\llbracket * \rrbracket_y$. It would not be hard to extend this encoding to handle the general case in which e_1 and e_2 may raise exceptions and return values different from **unit**. To that end, typing should ensure that e_1 and e_2 are each typable in an abortable context (cf. Rule (T&)), but also that the name representing the continuation to the enclosing exception handlers (i.e., x) is given an abortable type.

5 Further Related Work

In the purely functional (and sequential) programming setting, control operators have been given Curry-Howard interpretations in the context of classical logic [26,36,2]. To our best knowledge, this paper presents the first attempt at tackling state-aware concurrent programming features, involving linearity (our main focus herein), while building on a Curry-Howard interpretation of classical linear logic as session types. A very tentative sketch of some ideas behind this work was presented at Cardelli's Fest [8]; here we provide a complete account of non-determinism and failure, introduce new computational primitives, present associated results, and provide non-trivial examples, including the typeful embedding of a realistic functional, concurrent language with exceptions.

The tensions between affinity, linearity and control effects have been widely investigated in different settings, and already referred in the introduction. The work [45] considers a form of affinity in stateful settings (including session types) and explores how to safely interface an affine language with a conventional one. We share several high level aims with [45], although following a fundamentally different approach, and obtaining results of different relevance; in particular, we consider a unified (concurrent) language that admits a fundamental Curry-Howard correspondence with linear logic,

$$\left\llbracket \frac{}{z{:}T \vdash^{\mathsf{0}} z{:}T} \right\rrbracket y \quad = \frac{}{y\langle z\rangle.y.\overline{\mathsf{0}} \vdash z{:}\overline{\llbracket T\rrbracket}, y{:}\uparrow\llbracket T\rrbracket}$$

$$\left\llbracket \frac{U \neq \mathsf{0}}{z{:}T \vdash^{U} z{:}T} \right\rrbracket y, x = \frac{}{y.\overline{\mathtt{some}}; y\langle z\rangle.y\langle x\rangle.y.\overline{\mathsf{0}} \vdash z{:}\overline{\llbracket T\rrbracket}, y{:}\&\big(\llbracket T\rrbracket \div \overline{\llbracket U\rrbracket}\big), x{:}\llbracket U\rrbracket \boxplus \llbracket T\rrbracket}$$

$$\left\llbracket \frac{\mathsf{D}, z : A \vdash^{\mathsf{0}} e : B}{\mathsf{D} \vdash^{\mathsf{0}} \lambda z.e : A \xrightarrow{\mathsf{0}} B} \right\rrbracket y = \frac{}{\overline{y}(f).(y.\overline{\mathsf{0}} \mid f(z).f(k).f.\mathsf{0}; \llbracket e\rrbracket_{k}) \vdash \overline{\llbracket \mathsf{D}\rrbracket}, y{:}\uparrow\llbracket A \xrightarrow{\mathsf{0}} B\rrbracket}$$

$$\left\llbracket \frac{\mathsf{D}, z : A \vdash^{U} e : B}{\mathsf{D} \vdash^{\mathsf{0}} \lambda z.e : A \xrightarrow{U} B} \right\rrbracket y = \frac{}{\overline{y}(f).(y.\overline{\mathsf{0}} \mid f(z).f(k).f(j).f.\mathsf{0}; \llbracket e\rrbracket_{k,j}) \vdash \overline{\llbracket \mathsf{D}\rrbracket}, y{:}\uparrow\llbracket A \xrightarrow{U} B\rrbracket}$$

$$\left\llbracket \frac{\mathsf{D}, z : A \vdash^{U} e : B}{\mathsf{D} \vdash^{V} \lambda z.e : A \xrightarrow{U} B} \right\rrbracket y, x = \frac{}{\begin{array}{l} y.\overline{\mathtt{some}}; \overline{y}(f).(y\langle x\rangle.y.\overline{\mathsf{0}} \mid f(z).f(k).f(j).f.\mathsf{0}; \llbracket e\rrbracket_{k,j}) \\ \quad \vdash \overline{\llbracket \mathsf{D}\rrbracket}, y{:}\&\big(\llbracket A \xrightarrow{U} B\rrbracket \div \overline{\llbracket V\rrbracket}\big), x{:}\llbracket V\rrbracket \boxplus \llbracket A \xrightarrow{U} B\rrbracket \end{array}}$$

$$\left\llbracket \frac{\mathsf{D} \vdash^{U} f : A \xrightarrow{T} B \quad \mathsf{D}' \vdash^{V} a : A}{\mathsf{D}, \mathsf{D}' \vdash^{T} (f\, a) : B} \right\rrbracket y, x = \frac{}{\begin{array}{r} f\langle a\rangle.f\langle y\rangle.f\langle x\rangle.f.\overline{\mathsf{0}} \vdash f{:}\overline{\llbracket A \xrightarrow{T} B\rrbracket}, \\ y{:}\&\big(\llbracket B\rrbracket \div \overline{\llbracket T\rrbracket}\big), x{:}\llbracket T\rrbracket \boxplus \llbracket B\rrbracket, a{:}\overline{\llbracket A\rrbracket} \end{array}}$$

$$\left\llbracket \frac{\mathsf{D} \vdash^{U} f : A \xrightarrow{\mathsf{0}} B \quad \mathsf{D}' \vdash^{V} a : A}{\mathsf{D}, \mathsf{D}' \vdash^{\mathsf{0}} (f\, a) : B} \right\rrbracket y = \frac{}{f\langle a\rangle.f\langle y\rangle.f\langle x\rangle.f.\overline{\mathsf{0}} \vdash f{:}\overline{\llbracket A \xrightarrow{\mathsf{0}} B\rrbracket}, y{:}\uparrow\llbracket B\rrbracket, a{:}\overline{\llbracket A\rrbracket}}$$

$$\left\llbracket \frac{\mathsf{D}_1 \vdash^{\mathsf{0}} e_1 : A \quad \mathsf{D}_2, a{:}A \vdash^{\mathsf{0}} e_2 : B}{\mathsf{D}_1, \mathsf{D}_2 \vdash^{\mathsf{0}} \mathtt{LET}\, a = e_1\, \mathtt{IN}\, e_2 : B} \right\rrbracket y = \frac{}{(\nu q)(\llbracket e_1\rrbracket_q \mid q(a).q.\mathsf{0}; \llbracket e_2\rrbracket_y) \vdash \overline{\llbracket \mathsf{D}_1\rrbracket}, \overline{\llbracket \mathsf{D}_2\rrbracket}, y{:}\uparrow\llbracket B\rrbracket}$$

$$\left\llbracket \frac{\mathsf{D}_1 \vdash^{\mathsf{0}} e_1 : A \quad \mathsf{D}_2, a{:}A \vdash^{T} e_2{:}B}{\mathsf{D}_1, \mathsf{D}_2 \vdash^{T} \mathtt{LET}\, a = e_1\, \mathtt{IN}\, e_2 : B} \right\rrbracket y, x = \frac{}{\begin{array}{l} (\nu q)(\llbracket e_1\rrbracket_q \mid q(a).q.\mathsf{0}; \llbracket e_2\rrbracket_{y,x}) \\ \vdash \overline{\llbracket \mathsf{D}_1\rrbracket}, \overline{\llbracket \mathsf{D}_2\rrbracket}, y{:}\&\big(\llbracket B\rrbracket \div \overline{\llbracket T\rrbracket}\big), x{:}\llbracket T\rrbracket \boxplus \llbracket B\rrbracket \end{array}}$$

$$\left\llbracket \frac{\mathsf{D}_1 \vdash^{T} e_1 : A \;\; \oplus\mathsf{D}_2, a{:}A \vdash^{T} e_2 : B}{\mathsf{D}_1, \oplus\mathsf{D}_2 \vdash^{T} \mathtt{LET}\, a = e_1\, \mathtt{IN}\, e_2 : B} \right\rrbracket y, x = \frac{}{\begin{array}{l} (\nu q)(\llbracket e_1\rrbracket_{q,x} \mid q.\mathtt{some}_D; q(a).q(s).q.\mathsf{0}; \llbracket e_2\rrbracket_{y,s}) \\ \quad \vdash \overline{\llbracket \mathsf{D}_1\rrbracket}, \&\overline{\llbracket \mathsf{D}_2\rrbracket}, y{:}\&\big(\llbracket B\rrbracket \div \overline{\llbracket T\rrbracket}\big), x{:}\llbracket T\rrbracket \boxplus \llbracket B\rrbracket \end{array}}$$

$$\left\llbracket \frac{\mathsf{D} \vdash^{\mathsf{0}} e : B}{\mathsf{D} \vdash^{T} \mathtt{LIFT}\, e : B} \right\rrbracket y, x = \frac{}{\begin{array}{c} (\nu q)(\llbracket e\rrbracket_q \mid q(v).q.\mathsf{0}; y.\overline{\mathtt{some}}; y\langle v\rangle.y\langle x\rangle.y.\overline{\mathsf{0}}) \\ \vdash \overline{\llbracket \mathsf{D}\rrbracket}, y{:}\&\big(\llbracket T\rrbracket \div \overline{\llbracket B\rrbracket}\big), x{:}\llbracket T\rrbracket \boxplus \llbracket B\rrbracket \end{array}}$$

$$\left\llbracket \frac{\oplus\mathsf{D} \vdash^{\mathsf{0}} v : T}{\oplus\mathsf{D} \vdash^{\mathsf{0}} \langle\!\langle v\rangle\!\rangle : \oplus T} \right\rrbracket y = \frac{}{\begin{array}{l} \overline{y}(z).(z.\mathtt{some}_D; (\nu q)(\llbracket v\rrbracket_q \mid q(u).q.\mathsf{0}; [u \leftrightarrow z]) \mid y.\overline{\mathsf{0}}) \\ \vdash \&\overline{\llbracket \mathsf{D}\rrbracket}, y{:}\uparrow(\oplus\llbracket T\rrbracket) \end{array}}$$

Fig. 6. Typeful encoding of λ^{exc} terms into basic processes (Part 1).

and offers strong guarantees by static typing such as deadlock-freedom. Within the session types literature, the interplay of session types and functional languages (including encodings of functional calculi) has received much attention (see, e.g., [25,48,35,32]) but non-determinism/failure do not seem to have been addressed. The paper [35] relates effect and session type systems, but effects such as exceptions are not addressed. A work exploring affinity in session calculi is [34]. Existing works on exception mechanisms for session types impose severe syntactic restrictions to typable programs and/or do

$$\left\llbracket \frac{\mathsf{D}_1 \vdash^U e_1 : T \quad z : U \vdash^0 e_2 : T}{\mathsf{D}_1 \vdash^0 \texttt{TRY}\, e_1\, \texttt{CATCH}\, z.\, e_2 : T} \right\rrbracket y \;=\; \overline{\begin{array}{l} (\nu j)((\nu k)(\llbracket e_1\rrbracket_{k,j} \mid \\ \quad k.\texttt{some}_\emptyset; k(u).k(z).k.\mathsf{0}; z.\texttt{inl}; z\langle u\rangle.z.\overline{\mathsf{0}}) \mid \\ \quad j.\texttt{case}(j(u).j.\mathsf{0}; y\langle u\rangle.y.\overline{\mathsf{0}}\,,\ j(z).j.\mathsf{0}; \llbracket e_2\rrbracket_y)) \\ \quad \vdash \overline{\llbracket \mathsf{D}_1 \rrbracket}, y{:}\uparrow\llbracket T\rrbracket \end{array}}$$

$$\left\llbracket \frac{\mathsf{D} \vdash^0 z : T}{\mathsf{D} \vdash^T \texttt{THROW}\, z : U} \right\rrbracket y, x = \overline{\begin{array}{l} y.\overline{\texttt{none}} \mid x.\texttt{inr}; x\langle z\rangle.x.\overline{\mathsf{0}} \\ \quad \vdash \overline{\llbracket \mathsf{D}\rrbracket}, y{:}\&\big(\llbracket T\rrbracket \div \overline{\llbracket U\rrbracket}\big), x{:}\llbracket T\rrbracket \boxplus \llbracket U\rrbracket \end{array}}$$

$$\left\llbracket \frac{\mathsf{D}, x : S \vdash^0 e : \mathbf{unit}}{\mathsf{D} \vdash^0 \texttt{FORK}\, x.e : \overline{S}} \right\rrbracket y \;=\; \overline{\overline{y}(x).((\nu q)(\llbracket e\rrbracket_q \mid \mathsf{S}_q) \mid y.\overline{\mathsf{0}}) \vdash \overline{\llbracket \mathsf{D}\rrbracket}, y{:}\uparrow\llbracket \overline{S}\rrbracket}$$

$$\left\llbracket \frac{\mathsf{D}, z : S \vdash^U e : T}{\mathsf{D}, z : \oplus S \vdash^U \texttt{SOME}\,!\, z; e : T} \right\rrbracket y, x = \overline{\begin{array}{c} z.\overline{\texttt{some}}; \llbracket e\rrbracket_{y,x} \vdash \overline{\llbracket \mathsf{D}\rrbracket}, z{:}\&\overline{\llbracket S\rrbracket}, y{:}\&\big(\llbracket T\rrbracket \div \overline{\llbracket U\rrbracket}\big), \\ x{:}\llbracket U\rrbracket \boxplus \llbracket T\rrbracket \end{array}}$$

$$\left\llbracket \frac{\oplus\mathsf{D}, z : S \vdash^0 e : \mathbf{unit}}{\oplus\mathsf{D}, z : \&S \vdash^0 \texttt{SOME}\,?\, z; e : \mathbf{unit}} \right\rrbracket y \;=\; \overline{\begin{array}{l} z.\texttt{some}_D; (\nu q)(\llbracket e\rrbracket_q \mid \mathsf{S}_q) \mid \overline{y}(v).(v.\overline{\mathsf{0}} \mid y.\overline{\mathsf{0}}) \\ \vdash \&\overline{\llbracket \mathsf{D}\rrbracket}, z{:}\oplus\overline{\llbracket S\rrbracket}, y{:}\uparrow\llbracket \mathbf{unit}\rrbracket \end{array}}$$

$$\left\llbracket \frac{\mathsf{D} \vdash^U e : T}{\mathsf{D}, z : \oplus S \vdash^U \texttt{NONE}\,!\, z; e : T} \right\rrbracket y, x = \overline{\begin{array}{c} z.\overline{\texttt{none}} \mid \llbracket e\rrbracket_{y,x} \vdash \overline{\llbracket \mathsf{D}\rrbracket}, z{:}\&\overline{\llbracket S\rrbracket}, y{:}\&\big(\llbracket T\rrbracket \div \overline{\llbracket U\rrbracket}\big), \\ x{:}\llbracket U\rrbracket \boxplus \llbracket T\rrbracket \end{array}}$$

$$\left\llbracket \frac{\mathsf{D} \vdash^0 a : T_1 \qquad \mathsf{D}', c : T_2 \vdash^U e : T}{\mathsf{D}, \mathsf{D}', c : !T_1.T_2 \vdash^U \texttt{SEND}(c, a); e : T} \right\rrbracket y, x = \overline{\begin{array}{l} c\langle a\rangle.\llbracket e\rrbracket_{y,x} \vdash \llbracket \mathsf{D}'\rrbracket, c{:}\llbracket T_1\rrbracket \otimes \overline{\llbracket T_2\rrbracket}, \\ \qquad y{:}\&\big(\llbracket T\rrbracket \div \overline{\llbracket U\rrbracket}\big), x{:}\llbracket U\rrbracket \boxplus \llbracket T\rrbracket, a{:}\overline{\llbracket T_1\rrbracket} \end{array}}$$

$$\left\llbracket \frac{\mathsf{D}, z : T_1, c : T_2 \vdash^U e : T}{\mathsf{D}, c : ?T_1.T_2 \vdash^U \texttt{RECV}(c, z); e : T} \right\rrbracket y, x = \overline{\begin{array}{l} c(z).\llbracket e\rrbracket_{y,x} \vdash \overline{\llbracket \mathsf{D}\rrbracket}, c{:}\overline{\llbracket T_1\rrbracket} \parr \overline{\llbracket T_2\rrbracket}, \\ \qquad y{:}\&\big(\llbracket T\rrbracket \div \overline{\llbracket U\rrbracket}\big), x{:}\llbracket U\rrbracket \boxplus \llbracket T\rrbracket \end{array}}$$

$$\left\llbracket \frac{\oplus\mathsf{D} \vdash^0 e_1 : \mathbf{unit} \quad \oplus\mathsf{D} \vdash^0 e_2 : \mathbf{unit}}{\oplus\mathsf{D} \vdash^0 e_1 \oplus e_2 : \mathbf{unit}} \right\rrbracket y = \overline{\begin{array}{l} ((\nu z)(\llbracket e_1\rrbracket_z \mid \mathsf{S}_z) \oplus (\nu z)(\llbracket e_2\rrbracket_z \mid \mathsf{S}_z)) \mid \llbracket *\rrbracket_y \\ \vdash \&\overline{\llbracket \mathsf{D}\rrbracket}, y{:}\uparrow\llbracket \mathbf{unit}\rrbracket \end{array}}$$

$$\left\llbracket \frac{\mathsf{D} \vdash^T e : \mathbf{unit}}{\mathsf{D}, c : \mathsf{end}_! \vdash^T \texttt{CLOSE}\,!\, c; e : \mathbf{unit}} \right\rrbracket y, x = \overline{\begin{array}{l} c.\overline{\mathsf{0}} \mid \llbracket e\rrbracket_{y,x} \\ \vdash c{:}\overline{\llbracket \mathsf{end}_!\rrbracket}, y{:}\&\big(\llbracket \mathbf{unit}\rrbracket \div \overline{\llbracket T\rrbracket}\big), x{:}\llbracket \mathbf{unit}\rrbracket \boxplus \llbracket T\rrbracket \end{array}}$$

$$\left\llbracket \frac{}{c : \mathsf{end}_? \vdash^T \texttt{CLOSE}\,?\, c : \mathbf{unit}} \right\rrbracket y, x = \overline{\begin{array}{l} c.\mathsf{0}; \mathbf{0} \mid y.\overline{\texttt{some}}; \overline{y}(r).(r.\overline{\mathsf{0}} \mid y\langle x\rangle.y.\overline{\mathsf{0}}) \\ \vdash c{:}\overline{\llbracket \mathsf{end}_?\rrbracket}, y{:}\&\big(\llbracket \mathbf{unit}\rrbracket \div \overline{\llbracket T\rrbracket}\big), x{:}\llbracket T\rrbracket \boxplus \llbracket \mathbf{unit}\rrbracket \end{array}}$$

$$\left\llbracket \frac{}{\cdot \vdash^0 * : \mathbf{unit}} \right\rrbracket y \;=\; \overline{\overline{y}(u).(u.\overline{\mathsf{0}} \mid y.\overline{\mathsf{0}}) \vdash y{:}\uparrow\llbracket \mathbf{unit}\rrbracket}$$

Fig. 7. Typeful encoding of λ^{exc} terms into basic processes (Part 2).

not ensure progress: this observation applies to models of interactional exceptions and interruptible sessions based on both binary sessions (cf. [15]) and multiparty sessions (cf. [14,21]). Further work is required to connect our process model (based on binary session types) with multiparty structured interactions with exceptions/interruptions, following logic-based relationships between binary and multiparty session types [9].

Also related are [3,17]. The work in [3] explores forms of non-determinism and failure via the conflation of additive connectives. This is quite different from our approach, which is based on a new pair of monadic/comonadic connectives, fully justified by a Curry-Howard interpretation and expressive enough to represent forms of affinity and exceptions. The work in [17] does have non-determinism at the level of processes, but its expressiveness is not analyzed, and non-determinism at the level of types is not addressed. In contrast, we provide types for non-determinism via specific connectives in the context of a Curry-Howard correspondence, and exploit the expressiveness of the non-deterministic process model by modeling a realistic functional language.

As explained in the introduction, a main aim of this work is not just to propose yet another point in the design space solution for exceptions, affinity, or linearity. Instead, we show how a small set of logically motivated primitives is expressive enough to model fairly general notions of (controlled) affinity and non-determinism in higher-order concurrent programs (including exception handling) while preserving all the fundamental properties of a Curry-Howard interpretation for linear logic. We leave for future work a deeper study of the expressiveness of our model, as exceptions and compensations are key programming abstractions in models of service-oriented computing (see, e.g., [23]).

6 Concluding Remarks

We have presented the first type system that accommodates non-deterministic and abortable behaviors within session-based concurrent programs while building on a Curry-Howard correspondence with linear logic. Conceptually simple, our approach conservatively extends classical linear logic with two dual modal connectives, related to linear logic exponentials, but that express non-determinism and failure rather than sharing.

We have shown that our type system enforces progress and session fidelity; its underlying operational semantics, based on Curry-Howard principles, is actually compatible with standard non-confluent formulations of internal non-determinism for process algebra, in the sense of our postponing result (Theorem 3.5). Our system is very expressive, as illustrated by several examples, including a typed embedding of a higher-order linear functional language with threads, sessions, non-determinism, and exceptions.

We have not discussed the presence of intuitionistic (unrestricted) types in the functional language of § 4, as the main focus in the paper is on linearity and its challenging combination with non-determinism and failure. The combination of these ingredients with general (non-linear) functional values and shared sessions would be as expected, resulting from the type discipline of the interpretation of the exponentials in the basic model. Also, key properties of our type system such as strong normalization and confluence can be established along predictable lines [37]. A further advantage of our approach is its natural compatibility with other extensions to the basic framework, for example behavioral polymorphism [10]. Another interesting direction for future work is to better understand the behavioral equivalences induced by our interpretation.

Acknowledgments. Thanks to the anonymous reviewers for useful remarks and suggestions. This work has been partially sponsored by FCT PEst/UID/CEC/04516/2013; by FCT CLAY PTDC/EEI-CTP/4293/2014; by EU COST Actions IC1201 (BETTY), IC1402 (ARVI), and IC1405 (Reversible Computation); and by CNRS PICS project 07313 (SuCCeSS).

References

1. J.-M. Andreoli. Logic Programming with Focusing Proofs in Linear Logic. *J. Log. Comput.*, 2(3):297–347, 1992.
2. Z. M. Ariola and H. Herbelin. Minimal Classical Logic and Control Operators. In *ICALP 2003*, pages 871–885, 2003.
3. R. Atkey, S. Lindley, and J. G. Morris. Conflation confers concurrency. In *A List of Successes That Can Change the World*, volume 9600 of *LNCS*, pages 32–55. Springer, 2016.
4. A. Barber. Dual Intuitionistic Linear Logic. Technical Report LFCS-96-347, Univ. of Edinburgh, 1996.
5. P. N. Benton, G. M. Bierman, and V. de Paiva. Computational Types from a Logical Perspective. *J. Funct. Program.*, 8(2):177–193, 1998.
6. P. N. Benton, G. M. Bierman, V. de Paiva, and M. Hyland. A term calculus for intuitionistic linear logic. In *Proc. of TLCA '93*, volume 664 of *LNCS*, pages 75–90. Springer, 1993.
7. M. Boreale. On the Expressiveness of Internal Mobility in Name-Passing Calculi. *Theor. Comput. Sci.*, 195(2):205–226, 1998.
8. L. Caires. Types and Logic, Concurrency and Non-Determinism. In M. Abadi, P. Gardner, A. D. Gordon, and R. Mardare, editors, *Essays for the Luca Cardelli Fest*, pages 69–83. Microsoft Research TR MSR-TR-2014-104, 2014.
9. L. Caires and J. A. Pérez. Multiparty session types within a canonical binary theory, and beyond. In *FORTE 2016*, pages 74–95, 2016.
10. L. Caires, J. A. Pérez, F. Pfenning, and B. Toninho. Behavioral Polymorphism and Parametricity in Session-Based Communication. In *ESOP'13*, number 7792 in LNCS, 2013.
11. L. Caires and F. Pfenning. Session Types as Intuitionistic Linear Propositions. In *CONCUR'10*, number 6269 in LNCS, pages 222–236, 2010.
12. L. Caires, F. Pfenning, and B. Toninho. Towards concurrent type theory. In *Types in Language Design and Implementation*, pages 1–12, 2012.
13. L. Caires, F. Pfenning, and B. Toninho. Linear Logic Propositions as Session Types. *Math. Struct. in Comp. Sci.*, 2016.
14. S. Capecchi, E. Giachino, and N. Yoshida. Global Escape in Multiparty Sessions. *Mathematical Structures in Computer Science*, 26(2):156–205, 2016.
15. M. Carbone, K. Honda, and N. Yoshida. Structured Interactional Exceptions in Session Types. In *CONCUR 2008*, volume 5201 of *LNCS*, pages 402–417. Springer, 2008.
16. M. Carbone, S. Lindley, F. Montesi, C. Schürmann, and P. Wadler. Coherence generalises duality: A logical explanation of multiparty session types. In *CONCUR 2016*, pages 33:1–33:15, 2016.
17. M. Carbone, F. Montesi, C. Schürmann, and N. Yoshida. Multiparty session types as coherence proofs. In *Proc. of CONCUR 2015*, volume 42 of *LIPIcs*, pages 412–426. Schloss Dagstuhl, 2015.
18. L. Cardelli. Typeful Programming. *IFIP State-of-the-Art Reports: Formal Description of Programming Concepts*, pages 431–507, 1991.
19. R. De Nicola and M. Hennessy. CCS without tau's. In *Proc. of TAPSOFT'87*, volume 249 of *LNCS*, pages 138–152. Springer, 1987.
20. R. DeLine and M. Fähndrich. Typestates for Objects. In *ECOOP 2004 - Object-Oriented Programming, 18th European Conference*, pages 465–490, 2004.
21. R. Demangeon, K. Honda, R. Hu, R. Neykova, and N. Yoshida. Practical Interruptible Conversations: Distributed Dynamic Verification with Multiparty Session Types and Python. *Formal Methods in System Design*, 46(3):197–225, 2015.
22. T. Ehrhard and L. Regnier. Differential Interaction Nets. *Theor. Comput. Sci.*, 364(2):166–195, 2006.

23. C. Ferreira, I. Lanese, A. Ravara, H. T. Vieira, and G. Zavattaro. Advanced Mechanisms for Service Combination and Transactions. In *Results of the SENSORIA Project*, volume 6582 of *LNCS*, pages 302–325. Springer, 2011.
24. P. Gardner, C. Laneve, and L. Wischik. Linear Forwarders. *Information and Compututation*, 205(10):1526–1550, 2007.
25. S. Gay and V. T. Vasconcelos. Linear type theory for asynchronous session types. *J. Funct. Programming*, 20(1):19–50, 2010.
26. T. Griffin. A Formulae-as-Types Notion of Control. In *POPL'90*, pages 47–58, 1990.
27. K. Honda. Types for Dyadic Interaction. In *CONCUR'93*, number 715 in LNCS, pages 509–523, 1993.
28. K. Honda, V. T. Vasconcelos, and M. Kubo. Language Primitives and Type Discipline for Structured Communication-Based Programming. In *ESOP'98*, number 1381 in LNCS, 1998.
29. H. Hüttel, I. Lanese, V. T. Vasconcelos, and L. C. et. al. Foundations of Session Types and Behavioural Contracts. *ACM Comput. Surv.*, 49(1):3, 2016.
30. N. Kobayashi, B. C. Pierce, and D. N. Turner. Linearity and the pi-calculus. In *23rd Symposium on Principles of Programming Languages*, POPL'96, pages 358–371. ACM, 1996.
31. N. R. Krishnaswami, A. Turon, D. Dreyer, and D. Garg. Superficially Substructural Types. In *ICFP'12*, pages 41–54, 2012.
32. S. Lindley and J. G. Morris. Embedding session types in Haskell. In *9th International Symposium on Haskell, Haskell 2016*, pages 133–145, 2016.
33. F. Militão, J. Aldrich, and L. Caires. Rely-Guarantee Protocols. In *ECOOP 2014 - Object-Oriented Programming - 28th European Conference*, pages 334–359, 2014.
34. D. Mostrous and V. T. Vasconcelos. Affine Sessions. In *Proc. of COORDINATION 2014*, volume 8459 of *LNCS*, pages 115–130. Springer, 2014.
35. D. A. Orchard and N. Yoshida. Effects as sessions, sessions as effects. In *Proc. of POPL 2016*, pages 568–581. ACM, 2016.
36. M. Parigot. Lambda-Mu-Calculus: An Algorithmic Interpretation of Classical Natural Deduction. In *LPAR'92*, pages 190–201, 1992.
37. J. A. Pérez, L. Caires, F. Pfenning, and B. Toninho. Linear Logical Relations for Session-Based Concurrency. In *ESOP'12*, number 7211 in LNCS, 2012.
38. F. Pfenning. Structural Cut Elimination. In *10th Annual IEEE Symposium on Logic in Computer Science, LICS'95*, pages 156–166. IEEE Computer Society, 1995.
39. G. D. Plotkin. A Powerdomain Construction. *SIAM J. Comput.*, 5(3):452–487, 1976.
40. D. Sangiorgi and D. Walker. *The π-calculus: A Theory of Mobile Processes*. Cambridge University Press, 2001.
41. A. Scalas and N. Yoshida. Lightweight Session Programming in Scala. In *30th European Conference on Object-Oriented Programming, ECOOP 2016*, pages 21:1–21:28, 2016.
42. B. Toninho, L. Caires, and F. Pfenning. Dependent session types via intuitionistic linear type theory. In *PPDP'11*, pages 161–172, 2011.
43. B. Toninho, L. Caires, and F. Pfenning. Functions as Session-Typed Processes. In *FoSSaCS'12*, number 7213 in LNCS, 2012.
44. B. Toninho, L. Caires, and F. Pfenning. Higher-order processes, functions, and sessions: A monadic integration. In *ESOP'13*, pages 350–369, 2013.
45. J. A. Tov and R. Pucella. Stateful Contracts for Affine Types. In *ESOP 2010*, pages 550–569, 2010.
46. J. A. Tov and R. Pucella. A theory of Substructural Types and Control. In *OOPSLA 2011*, pages 625–642, 2011.
47. J. A. Tov and R. Pucella. Practical Affine Types. In *POPL 2011*, pages 447–458, 2011.
48. P. Wadler. Propositions as Sessions. In *ICFP'12*, ACM, pages 273–286, 2012.

A Typing the Encoding of Lambda with Exceptions

This section is devoted to the proof of Theorem 4.2 (Page 20). The proof proceeds by induction on the structure of the typed expression e. Each case is treated separately in its own subsection (below, from A.1 until A.13).

Next, we present type derivations in the target language (the process model with non determinism) avoiding abbreviations as much as possible; then, in the boxed frames, we report the resulting typing judgment as given in Figures 6 and 7. Such figures use the following abbreviations for types, which for convenience, we repeat here:

$$\begin{aligned}\uparrow[\![T]\!] &\triangleq [\![T]\!] \otimes \mathbf{1}\\ [\![U]\!] \boxplus [\![T]\!] &\triangleq ([\![U]\!] \otimes \mathbf{1}) \oplus ([\![T]\!] \otimes \mathbf{1})\\ \&\big([\![T]\!] \div \overline{[\![U]\!]}\big) &\triangleq \&([\![T]\!] \otimes ((\overline{[\![U]\!] \boxplus [\![T]\!]}) \otimes \mathbf{1}))\end{aligned}$$

Also, we recall the abbreviations for processes:

$$\begin{aligned}y\langle z\rangle.P &\triangleq \overline{y}(w).([w \leftrightarrow z] \mid P) && \text{(where } z \text{ is free in } P\text{)}\\ y.0; P &\triangleq y.\mathtt{close} && \text{(similarly for } y.\overline{\mathtt{close}} \text{ and } y.\overline{0}\text{)}\\ \mathsf{S}_q &\triangleq q(u).q.0; u.0; \mathbf{0}\end{aligned}$$

A.1 Variables

See Figure 8.

A.2 Abstractions

See Figures 9 and 10.

A.3 Application

See Figure 11.

A.4 Let

See Figures 12 and 13.

In the typing derivation of Figure 13 we require the following proposition, which exploits the flexibility of labeled choices in the typing of exception handlers:

Proposition A.1. *Let e be an expression. Then we have:*

1. *If* $[\![e]\!]_{y,x} \vdash y : \&([\![T]\!] \otimes ((\overline{[\![U]\!] \boxplus [\![T]\!]}) \otimes \mathbf{1})), x : [\![U]\!] \boxplus [\![T]\!]$ *then also* $[\![e]\!]_{y,x} \vdash y : \&([\![T]\!] \otimes ((\overline{[\![U]\!] \boxplus [\![T]\!] \boxplus [\![V]\!]}) \otimes \mathbf{1})), x : [\![U]\!] \boxplus [\![T]\!] \boxplus [\![V]\!]$*, for some* V.
2. *If* $[\![e]\!]_{y,x} \vdash y : \&([\![T]\!] \otimes ((\overline{[\![U]\!] \boxplus [\![T]\!] \boxplus [\![V]\!]}) \otimes \mathbf{1})), x : [\![U]\!] \boxplus [\![T]\!] \boxplus [\![V]\!]$, *then also* $[\![e]\!]_{y,x} \vdash y : \&([\![T]\!] \otimes ((\overline{[\![U]\!] \boxplus [\![T]\!]}) \otimes \mathbf{1})), x : [\![U]\!] \boxplus [\![T]\!]$ *(with* $V \neq T$*).*

Intuitively speaking, we can modify the exception handler of an expression by adding and removing possibilities. The proposition holds because exception handlers are essentially chains of forwarders, which admit any type.

Encoding **variables** (first case):

$$\llbracket z\rrbracket_y = \overline{y}(w).([w\leftrightarrow z] \mid y.\overline{\texttt{close}})$$

Source type judgment for z:

$$\frac{}{z{:}T \vdash^{0} z{:}T}$$

Corresponding typing derivation:

$$\dfrac{\dfrac{}{[w\leftrightarrow z]\vdash w{:}\llbracket T\rrbracket, z{:}\overline{\llbracket T\rrbracket}} \qquad\qquad \dfrac{}{y.\overline{\texttt{close}}\vdash y{:}\mathbf{1}}}{\overline{y}(w).([w\leftrightarrow z]\mid y.\overline{\texttt{close}})\vdash z{:}\overline{\llbracket T\rrbracket}, y{:}\llbracket T\rrbracket\otimes\mathbf{1}}$$

$$\boxed{\left\llbracket \dfrac{}{z{:}T\vdash^{0} z{:}T}\right\rrbracket y = \dfrac{}{y\langle z\rangle.y.\overline{\mathsf{0}}\vdash z{:}\overline{\llbracket T\rrbracket}, y{:}\uparrow\llbracket T\rrbracket}}$$

Encoding **variables** (second case):

$$\llbracket z\rrbracket_{y,x} = y.\overline{\texttt{some}}; \overline{y}(w).([w\leftrightarrow z] \mid \overline{y}(w).([x\leftrightarrow w] \mid y.\overline{\texttt{close}}))$$

Source type judgment for z:

$$\frac{}{z{:}T \vdash^{U} z{:}T}(U \neq \mathsf{0})$$

Corresponding typing derivation:

$$\dfrac{\dfrac{\dfrac{}{[w\leftrightarrow z]\vdash w{:}\llbracket T\rrbracket, z{:}\overline{\llbracket T\rrbracket}} \qquad \dfrac{\dfrac{}{[x\leftrightarrow w]\vdash x{:}\llbracket U\rrbracket\boxplus\llbracket T\rrbracket, w{:}\overline{\llbracket U\rrbracket\boxplus\llbracket T\rrbracket}} \qquad \dfrac{}{y.\overline{\texttt{close}}\vdash y{:}\mathbf{1}}}{\overline{y}(w).([x\leftrightarrow w]\mid y.\overline{\texttt{close}})\vdash y{:}(\overline{\llbracket U\rrbracket\boxplus\llbracket T\rrbracket})\otimes\mathbf{1}, x{:}\llbracket U\rrbracket\boxplus\llbracket T\rrbracket}}{\overline{y}(w).([w\leftrightarrow z]\mid \overline{y}(w).([x\leftrightarrow w]\mid y.\overline{\texttt{close}}))\vdash z{:}\overline{\llbracket T\rrbracket}, y{:}\llbracket T\rrbracket\otimes((\overline{\llbracket U\rrbracket\boxplus\llbracket T\rrbracket})\otimes\mathbf{1}), x{:}\llbracket U\rrbracket\boxplus\llbracket T\rrbracket}}{y.\overline{\texttt{some}};\overline{y}(w).([w\leftrightarrow z]\mid \overline{y}(w).([x\leftrightarrow w]\mid y.\overline{\texttt{close}})))\vdash z{:}\overline{\llbracket T\rrbracket}, y{:}\&(\llbracket T\rrbracket\otimes((\overline{\llbracket U\rrbracket\boxplus\llbracket T\rrbracket})\otimes\mathbf{1})), x{:}\llbracket U\rrbracket\boxplus\llbracket T\rrbracket}$$

$$\boxed{\left\llbracket \dfrac{}{z{:}T\vdash^{U} z{:}T}(U\neq\mathsf{0})\right\rrbracket y,x = \dfrac{}{y.\overline{\texttt{some}}; y\langle z\rangle.y\langle x\rangle.y.\overline{\mathsf{0}}\vdash z{:}\overline{\llbracket T\rrbracket}, y{:}\&\big(\llbracket T\rrbracket\div\overline{\llbracket U\rrbracket}\big), x{:}\llbracket U\rrbracket\boxplus\llbracket T\rrbracket}}$$

Fig. 8. Encoding variables.

Encoding **abstractions** (first case):

$$[\![\lambda z.e]\!]_y = \overline{y}(f).(y.\overline{\mathtt{close}} \mid f(z).f(k).f.\mathtt{close}; [\![e]\!]_k)$$

Source type judgment for $\lambda z.e$:

$$\dfrac{\mathsf{D}, z : A \vdash^{0} e : B}{\mathsf{D} \vdash^{0} \lambda z.e : A \xrightarrow{0} B}$$

Corresponding typing derivation:

$$\dfrac{\dfrac{\dfrac{\dfrac{[\![e]\!]_k \vdash \overline{\llbracket \mathsf{D} \rrbracket}, z{:}\overline{\llbracket A \rrbracket}, k{:}\llbracket B \rrbracket \otimes \mathbf{1}}{f.\mathtt{close}; [\![e]\!]_k \vdash \overline{\llbracket \mathsf{D} \rrbracket}, f{:}\bot, z{:}\overline{\llbracket A \rrbracket}, k{:}\llbracket B \rrbracket \otimes \mathbf{1}}}{f(k).f.\mathtt{close}; [\![e]\!]_k \vdash \overline{\llbracket \mathsf{D} \rrbracket}, f{:}(\llbracket B \rrbracket \otimes \mathbf{1}) \parr \bot, z{:}\overline{\llbracket A \rrbracket}}}{f(z).f(k).f.\mathtt{close}; [\![e]\!]_k \vdash \overline{\llbracket \mathsf{D} \rrbracket}, f{:}\overline{\llbracket A \rrbracket} \parr ((\llbracket B \rrbracket \otimes \mathbf{1}) \parr \bot)} \qquad \dfrac{}{y.\overline{\mathtt{close}} \vdash y{:}\mathbf{1}}}{\overline{y}(f).(y.\overline{\mathtt{close}} \mid f(z).f(k).f.\mathtt{close}; [\![e]\!]_k) \vdash \overline{\llbracket \mathsf{D} \rrbracket}, y{:}\, \underbrace{\big(\overline{\llbracket A \rrbracket} \parr ((\llbracket B \rrbracket \otimes \mathbf{1}) \parr \bot)\big)}_{\llbracket A \xrightarrow{0} B \rrbracket} \otimes \mathbf{1}}$$

$$\boxed{\left\llbracket \dfrac{\mathsf{D}, z : A \vdash^{0} e : B}{\mathsf{D} \vdash^{0} \lambda z.e : A \xrightarrow{0} B} \right\rrbracket y = \dfrac{}{\overline{y}(f).(y.\overline{\mathsf{0}} \mid f(z).f(k).f.\mathsf{0}; [\![e]\!]_k) \vdash \overline{\llbracket \mathsf{D} \rrbracket}, y{:} \uparrow \llbracket A \xrightarrow{0} B \rrbracket}}$$

Encoding **abstractions** (second case):

$$[\![\lambda z.e]\!]_y = \overline{y}(f).(y.\overline{\mathtt{close}} \mid f(z).f(k).f(j).f.\mathtt{close}; [\![e]\!]_{k,j})$$

Source type judgment for $\lambda z.e$:

$$\dfrac{\mathsf{D}, z : A \vdash^{U} e : B}{\mathsf{D} \vdash^{0} \lambda z.e : A \xrightarrow{U} B}$$

Corresponding typing derivation:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{[\![e]\!]_{k,j} \vdash \overline{\llbracket \mathsf{D} \rrbracket}, z{:}\overline{\llbracket A \rrbracket}, k{:}\mathbin{\&}\big(\llbracket B \rrbracket \div \overline{\llbracket U \rrbracket}\big), j{:}\llbracket B \rrbracket \boxplus \llbracket U \rrbracket}{f.\mathtt{close}; [\![e]\!]_{k,j} \vdash \overline{\llbracket \mathsf{D} \rrbracket}, f{:}\bot, z{:}\overline{\llbracket A \rrbracket}, k{:}\mathbin{\&}\big(\llbracket B \rrbracket \div \overline{\llbracket U \rrbracket}\big), j{:}\llbracket B \rrbracket \boxplus \llbracket U \rrbracket}}{f(j).f.\mathtt{close}; [\![e]\!]_{k,j} \vdash \overline{\llbracket \mathsf{D} \rrbracket}, f{:}(\llbracket B \rrbracket \boxplus \llbracket U \rrbracket) \parr \bot, z{:}\overline{\llbracket A \rrbracket}, k{:}\mathbin{\&}\big(\llbracket B \rrbracket \div \overline{\llbracket U \rrbracket}\big)}}{f(k).f(j).f.\mathtt{close}; [\![e]\!]_{k,j} \vdash \overline{\llbracket \mathsf{D} \rrbracket}, f{:}(\mathbin{\&}\big(\llbracket B \rrbracket \div \overline{\llbracket U \rrbracket}\big)) \parr ((\llbracket B \rrbracket \boxplus \llbracket U \rrbracket) \parr \bot), z{:}\overline{\llbracket A \rrbracket}}}{f(z).f(k).f(j).f.\mathtt{close}; [\![e]\!]_{k,j} \vdash \overline{\llbracket \mathsf{D} \rrbracket}, f{:}\overline{\llbracket A \rrbracket} \parr ((\mathbin{\&}\big(\llbracket B \rrbracket \div \overline{\llbracket U \rrbracket}\big)) \parr ((\llbracket B \rrbracket \boxplus \llbracket U \rrbracket) \parr \bot))} \qquad \dfrac{}{y.\overline{\mathtt{close}} \vdash y{:}\mathbf{1}}}{\overline{y}(f).(y.\overline{\mathtt{close}} \mid f(z).f(k).f(j).f.\mathtt{close}; [\![e]\!]_{k,j}) \vdash \overline{\llbracket \mathsf{D} \rrbracket}, y{:}\, \underbrace{\big(\overline{\llbracket A \rrbracket} \parr ((\mathbin{\&}\big(\llbracket B \rrbracket \div \overline{\llbracket U \rrbracket}\big)) \parr ((\llbracket B \rrbracket \boxplus \llbracket U \rrbracket) \parr \bot))\big)}_{\llbracket A \xrightarrow{U} B \rrbracket} \otimes \mathbf{1}}$$

$$\boxed{\left\llbracket \dfrac{\mathsf{D}, z : A \vdash^{U} e : B}{\mathsf{D} \vdash^{0} \lambda z.e : A \xrightarrow{U} B} \right\rrbracket y = \dfrac{}{\overline{y}(f).(y.\overline{\mathsf{0}} \mid f(z).f(k).f(j).f.\mathsf{0}; [\![e]\!]_{k,j}) \vdash \overline{\llbracket \mathsf{D} \rrbracket}, y{:} \uparrow \llbracket A \xrightarrow{U} B \rrbracket}}$$

Fig. 9. Encoding abstractions (Part 1).

Encoding **abstractions** (third case):

$$[\lambda z.e]_{y,x} = y.\overline{\texttt{some}}; \overline{y}(f).(\overline{y}(w).([x \leftrightarrow w] \mid y.\overline{\texttt{close}}) \mid f(z).f(k).f(j).f.\texttt{close}; [e]_{k,j}))$$

Source type judgment for $\lambda z.e$:

$$\dfrac{\mathsf{D}, z : A \vdash^{U} e : B}{\mathsf{D} \vdash^{V} \lambda z.e : A \xrightarrow{U} B}$$

Corresponding typing derivation (writing $c.\mathsf{0}$ instead of $c.\texttt{close}$):

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\llbracket e\rrbracket_{k,j} \vdash \overline{\llbracket \mathsf{D}\rrbracket}, z{:}\overline{\llbracket A\rrbracket}, k{:}\with\big(\llbracket B\rrbracket \div \overline{\llbracket U\rrbracket}\big), j{:}\llbracket U\rrbracket \boxplus \llbracket B\rrbracket}{f.\mathsf{0}; \llbracket e\rrbracket_{k,j} \vdash \overline{\llbracket \mathsf{D}\rrbracket}, f{:}\bot, z{:}\overline{\llbracket A\rrbracket}, k{:}\with\big(\llbracket B\rrbracket \div \overline{\llbracket U\rrbracket}\big), j{:}\llbracket U\rrbracket \boxplus \llbracket B\rrbracket}}{f(j).f.\mathsf{0}; \llbracket e\rrbracket_{k,j} \vdash \overline{\llbracket \mathsf{D}\rrbracket}, f{:}(\llbracket U\rrbracket \boxplus \llbracket B\rrbracket) \parr \bot, z{:}\overline{\llbracket A\rrbracket}, k{:}\with\big(\llbracket B\rrbracket \div \overline{\llbracket U\rrbracket}\big)}}{f(k).f(j).f.\mathsf{0}; \llbracket e\rrbracket_{k,j} \vdash \overline{\llbracket \mathsf{D}\rrbracket}, f{:}(\with\big(\llbracket B\rrbracket \div \overline{\llbracket U\rrbracket}\big)) \parr ((\llbracket U\rrbracket \boxplus \llbracket B\rrbracket) \parr \bot), z{:}\overline{\llbracket A\rrbracket}}}{f(z).f(k).f(j).f.\mathsf{0}; \llbracket e\rrbracket_{k,j} \vdash \overline{\llbracket \mathsf{D}\rrbracket}, f{:}\overline{\llbracket A\rrbracket} \parr ((\with\big(\llbracket B\rrbracket \div \overline{\llbracket U\rrbracket}\big)) \parr ((\llbracket U\rrbracket \boxplus \llbracket B\rrbracket) \parr \bot))} \quad \dfrac{\dfrac{}{\begin{array}{c}[x \leftrightarrow w] \vdash w{:}\overline{\llbracket V\rrbracket \boxplus \llbracket A \xrightarrow{U} B\rrbracket}, \\ x{:}\llbracket V\rrbracket \boxplus \llbracket A \xrightarrow{U} B\rrbracket\end{array}} \quad \dfrac{}{y.\overline{\mathsf{0}} \vdash y{:}\mathbf{1}}}{\begin{array}{r}\overline{y}(w).([x \leftrightarrow w] \mid y.\overline{\mathsf{0}}) \vdash y{:}(\overline{\llbracket V\rrbracket \boxplus \llbracket A \xrightarrow{U} B\rrbracket}) \otimes \mathbf{1}, \\ x{:}\llbracket V\rrbracket \boxplus \llbracket A \xrightarrow{U} B\rrbracket\end{array}}}{\dfrac{\begin{array}{l}\overline{y}(f).(\overline{y}(w).([x \leftrightarrow w] \mid y.\overline{\mathsf{0}}) \mid f(z).f(k).f(j).f.\mathsf{0}; \llbracket e\rrbracket_{k,j})) \\ \quad \vdash \llbracket \mathsf{D}\rrbracket, y{:}\big(\overline{\llbracket A\rrbracket} \parr ((\with\big(\llbracket B\rrbracket \div \overline{\llbracket U\rrbracket}\big)) \parr ((\llbracket U\rrbracket \boxplus \llbracket B\rrbracket) \parr \bot))\big) \otimes \big((\overline{\llbracket V\rrbracket \boxplus \llbracket A \xrightarrow{U} B\rrbracket}) \otimes \mathbf{1}\big), x{:}\llbracket V\rrbracket \boxplus \llbracket A \xrightarrow{U} B\rrbracket\end{array}}{\begin{array}{l}y.\overline{\texttt{some}}; \overline{y}(f).(\overline{y}(w).([x \leftrightarrow w] \mid y.\overline{\mathsf{0}}) \mid f(z).f(k).f(j).f.\mathsf{0}; \llbracket e\rrbracket_{k,j})) \\ \quad \vdash \overline{\llbracket \mathsf{D}\rrbracket}, y{:}\,\underbrace{\with\Big(\underbrace{(\overline{\llbracket A\rrbracket} \parr ((\with\big(\llbracket B\rrbracket \div \overline{\llbracket U\rrbracket}\big)) \parr ((\llbracket U\rrbracket \boxplus \llbracket B\rrbracket) \parr \bot)))}_{\llbracket A \xrightarrow{U} B\rrbracket} \otimes \big((\overline{\llbracket V\rrbracket \boxplus \llbracket A \xrightarrow{U} B\rrbracket}) \otimes \mathbf{1}\big)\Big)}_{\with\left(\llbracket A \xrightarrow{U} B\rrbracket \div \overline{\llbracket V\rrbracket}\right)}, x{:}\llbracket V\rrbracket \boxplus \llbracket A \xrightarrow{U} B\rrbracket\end{array}}}$$

$$\boxed{\left\llbracket \dfrac{\mathsf{D}, z : A \vdash^{U} e : B}{\mathsf{D} \vdash^{V} \lambda z.e : A \xrightarrow{U} B} \right\rrbracket y, x = \dfrac{}{\begin{array}{r}y.\overline{\texttt{some}}; \overline{y}(f).(y\langle x\rangle.y.\overline{\mathsf{0}} \mid f(z).f(k).f(j).f.\mathsf{0}; \llbracket e\rrbracket_{k,j})) \\ \vdash \overline{\llbracket \mathsf{D}\rrbracket}, y{:}\with\big(\llbracket A \xrightarrow{U} B\rrbracket \div \overline{\llbracket V\rrbracket}\big), x{:}\llbracket V\rrbracket \boxplus \llbracket A \xrightarrow{U} B\rrbracket\end{array}}}$$

Fig. 10. Encoding abstractions (Part 2).

Encoding **applications** (first case):

$$\llbracket (f\;a)\rrbracket_{y,x} = \overline{f}(a_1).([a\leftrightarrow a_1] \mid \overline{f}(y_1).([y\leftrightarrow y_1] \mid \overline{f}(x_1).([x\leftrightarrow x_1] \mid f.\overline{\texttt{close}})))$$

Source type judgment for $(f\;a)$:

$$\dfrac{\mathsf{D} \vdash^{U} f : A \xrightarrow{T} B \qquad \mathsf{D}' \vdash^{V} a : A}{\mathsf{D}, \mathsf{D}' \vdash^{T} (f\;a) : B}$$

Corresponding typing derivation:

$$\dfrac{\dfrac{}{[a\leftrightarrow a_1] \vdash a{:}\overline{\llbracket A\rrbracket}, a_1{:}\llbracket A\rrbracket} \quad \dfrac{\dfrac{}{\begin{array}{l}[y\leftrightarrow y_1] \vdash \\ \;y{:}\&(\llbracket B\rrbracket \otimes ((\overline{\llbracket B\rrbracket \boxplus \llbracket T\rrbracket}) \otimes \mathbf{1})), \\ \;y_1{:}\oplus(\overline{\llbracket B\rrbracket} \parr ((\llbracket B\rrbracket \boxplus \llbracket T\rrbracket) \parr \bot))\end{array}} \quad \dfrac{\dfrac{}{[x\leftrightarrow x_1] \vdash x{:}\llbracket T\rrbracket \boxplus \llbracket B\rrbracket, x_1{:}\overline{\llbracket T\rrbracket \boxplus \llbracket B\rrbracket}} \quad \dfrac{}{f.\overline{\texttt{close}} \vdash f{:}\mathbf{1}}}{\begin{array}{l}\overline{f}(x_1).([x\leftrightarrow x_1] \mid f.\overline{\texttt{close}}) \vdash \\ \;f{:}(\overline{\llbracket T\rrbracket \boxplus \llbracket B\rrbracket}) \otimes \mathbf{1}, \\ \;x{:}\llbracket T\rrbracket \boxplus \llbracket B\rrbracket\end{array}}}{\begin{array}{l}\overline{f}(y_1).([y\leftrightarrow y_1] \mid \overline{f}(x_1).([x\leftrightarrow x_1] \mid f.\overline{\texttt{close}})) \vdash \\ \;f{:}(\oplus(\overline{\llbracket B\rrbracket} \parr ((\llbracket B\rrbracket \boxplus \llbracket T\rrbracket) \parr \bot))) \otimes ((\overline{\llbracket T\rrbracket \boxplus \llbracket B\rrbracket}) \otimes \mathbf{1}) \\ \;y{:}\&(\llbracket B\rrbracket \otimes ((\overline{\llbracket B\rrbracket \boxplus \llbracket T\rrbracket}) \otimes \mathbf{1})), \\ \;x{:}\llbracket T\rrbracket \boxplus \llbracket B\rrbracket\end{array}}}{\begin{array}{l}\overline{f}(a_1).([a\leftrightarrow a_1] \mid \overline{f}(y_1).([y\leftrightarrow y_1] \mid \overline{f}(x_1).([x\leftrightarrow x_1] \mid f.\overline{\texttt{close}}))) \vdash \\ \;f{:}\,\llbracket A\rrbracket \otimes \underbrace{((\underbrace{\oplus(\overline{\llbracket B\rrbracket} \parr ((\llbracket B\rrbracket \boxplus \llbracket T\rrbracket) \parr \bot))}_{\overline{\&(\llbracket B\rrbracket \div \overline{\llbracket T\rrbracket})}}) \otimes ((\overline{\llbracket T\rrbracket \boxplus \llbracket B\rrbracket}) \otimes \mathbf{1}))}_{\overline{\llbracket A \xrightarrow{T} B\rrbracket}}, \\ \;y{:}\,\underbrace{\&(\llbracket B\rrbracket \otimes ((\overline{\llbracket B\rrbracket \boxplus \llbracket T\rrbracket}) \otimes \mathbf{1}))}_{\&(\llbracket B\rrbracket \div \overline{\llbracket T\rrbracket})}, \\ \;x{:}\llbracket T\rrbracket \boxplus \llbracket B\rrbracket, \\ \;a{:}\overline{\llbracket A\rrbracket}\end{array}}$$

$$\boxed{\left\llbracket \dfrac{\mathsf{D} \vdash^{U} f : A \xrightarrow{T} B \qquad \mathsf{D}' \vdash^{V} a : A}{\mathsf{D}, \mathsf{D}' \vdash^{T} (f\;a) : B} \right\rrbracket y, x = \dfrac{}{\begin{array}{l} f\langle a\rangle.f\langle y\rangle.\overline{f}(x).f.\overline{\mathsf{0}} \\ \vdash f{:}\overline{\llbracket A \xrightarrow{T} B\rrbracket}, y{:}\&(\llbracket B\rrbracket \div \overline{\llbracket T\rrbracket}), x{:}\llbracket T\rrbracket \boxplus \llbracket B\rrbracket, a{:}\overline{\llbracket A\rrbracket}\end{array}}}$$

Encoding **applications** (second case):

$$\llbracket (f\;a)\rrbracket_{y} = \overline{f}(a_1).([a\leftrightarrow a_1] \mid \overline{f}(y_1).([y\leftrightarrow y_1] \mid f.\overline{\texttt{close}}))$$

Source type judgment for $(f\;a)$:

$$\dfrac{\mathsf{D} \vdash^{U} f : A \xrightarrow{0} B \qquad \mathsf{D}' \vdash^{V} a : A}{\mathsf{D}, \mathsf{D}' \vdash^{0} (f\;a) : B}$$

Corresponding typing derivation:

$$\dfrac{\dfrac{}{[a\leftrightarrow a_1] \vdash a{:}\overline{\llbracket A\rrbracket}, a_1{:}\llbracket A\rrbracket} \quad \dfrac{\dfrac{}{[y\leftrightarrow y_1] \vdash y{:}\llbracket B\rrbracket \otimes \mathbf{1}, y_1{:}\overline{\llbracket B\rrbracket} \parr \bot} \quad \dfrac{}{f.\overline{\texttt{close}} \vdash f{:}\mathbf{1}}}{\overline{f}(y_1).([y\leftrightarrow y_1] \mid f.\overline{\texttt{close}}) \vdash f{:}(\overline{\llbracket B\rrbracket} \parr \bot) \otimes \mathbf{1}, y{:}\llbracket B\rrbracket \otimes \mathbf{1}}}{\overline{f}(a_1).([a\leftrightarrow a_1] \mid \overline{f}(y_1).([y\leftrightarrow y_1] \mid \overline{f}(x_1).([x\leftrightarrow x_1] \mid f.\overline{\texttt{close}}))) \vdash f{:}\,\llbracket A\rrbracket \otimes \underbrace{((\underbrace{\overline{\llbracket B\rrbracket} \parr \bot}_{\overline{\uparrow B}}) \otimes \mathbf{1})}_{\overline{\llbracket A \xrightarrow{0} B\rrbracket}}, y{:}\,\underbrace{\llbracket B\rrbracket \otimes \mathbf{1}}_{\Uparrow\llbracket B\rrbracket}, a{:}\overline{\llbracket A\rrbracket}}$$

$$\boxed{\left\llbracket \dfrac{\mathsf{D} \vdash^{U} f : A \xrightarrow{0} B \qquad \mathsf{D}' \vdash^{V} a : A}{\mathsf{D}, \mathsf{D}' \vdash^{0} (f\;a) : B} \right\rrbracket y = \dfrac{}{f\langle a\rangle.f\langle y\rangle.f\langle x\rangle.f.\overline{\mathsf{0}} \vdash f{:}\overline{\llbracket A \xrightarrow{0} B\rrbracket}, y{:}\,\uparrow\llbracket B\rrbracket, a{:}\overline{\llbracket A\rrbracket}}}$$

Fig. 11. Encoding applications.

Encoding **let** (first case):

$$[\![\mathtt{LET}\, a = e_1\ \mathtt{IN}\, e_2]\!]_y = (\nu q)([\![e_1]\!]_q \mid q(a).q.\mathtt{close}; [\![e_2]\!]_y)$$

Source type judgment for $\mathtt{LET}\, a = e_1\ \mathtt{IN}\, e_2$:

$$\dfrac{\mathsf{D}_1 \vdash^{\mathsf{0}} e_1 : A \qquad \mathsf{D}_2, a{:}A \vdash^{\mathsf{0}} e_2 : B}{\mathsf{D}_1, \mathsf{D}_2 \vdash^{\mathsf{0}} \mathtt{LET}\, a = e_1\ \mathtt{IN}\, e_2 : B}$$

Corresponding typing derivation:

$$\dfrac{\dfrac{}{[\![e_1]\!]_{q,x} \vdash \overline{\llbracket \mathsf{D}_1 \rrbracket}, q{:}\llbracket A \rrbracket \otimes \mathbf{1}} \qquad \dfrac{\dfrac{\dfrac{}{[\![e_2]\!]_y \vdash \overline{\llbracket \mathsf{D}_2 \rrbracket}, a{:}\overline{\llbracket A \rrbracket}, y{:}\llbracket B \rrbracket \otimes \mathbf{1}}}{q.\mathtt{close}; [\![e_2]\!]_{y,x} \vdash q{:}\bot, \overline{\llbracket \mathsf{D}_2 \rrbracket}, a{:}\overline{\llbracket A \rrbracket}, y{:}\llbracket B \rrbracket \otimes \mathbf{1}}}{q(a).q.\mathtt{close}; [\![e_2]\!]_{y,x} \vdash q{:}\overline{\llbracket A \rrbracket} \parr \bot, \overline{\llbracket \mathsf{D}_2 \rrbracket}, y{:}\llbracket B \rrbracket \otimes \mathbf{1}}}{(\nu q)([\![e_1]\!]_q \mid q(a).q.\mathtt{close}; [\![e_2]\!]_y) \vdash \overline{\llbracket \mathsf{D}_1 \rrbracket}, \overline{\llbracket \mathsf{D}_2 \rrbracket}, y{:}\, \underbrace{\llbracket B \rrbracket \otimes \mathbf{1}}_{\Uparrow \llbracket B \rrbracket}}$$

$$\boxed{\left\llbracket \dfrac{\mathsf{D}_1 \vdash^{\mathsf{0}} e_1 : A \qquad \mathsf{D}_2, a{:}A \vdash^{\mathsf{0}} e_2 : B}{\mathsf{D}_1, \mathsf{D}_2 \vdash^{\mathsf{0}} \mathtt{LET}\, a = e_1\ \mathtt{IN}\, e_2 : B} \right\rrbracket y = \dfrac{}{(\nu q)([\![e_1]\!]_q \mid q(a).q.\mathsf{0}; [\![e_2]\!]_y) \vdash \overline{\llbracket \mathsf{D}_1 \rrbracket}, \overline{\llbracket \mathsf{D}_2 \rrbracket}, y{:} \uparrow \llbracket B \rrbracket}}$$

Encoding **let** (second case):

$$[\![\mathtt{LET}\, a = e_1\ \mathtt{IN}\, e_2]\!]_{y,x} = (\nu q)([\![e_1]\!]_q \mid q(a).q.\mathtt{close}; [\![e_2]\!]_{y,x})$$

Source type judgment for $\mathtt{LET}\, a = e_1\ \mathtt{IN}\, e_2$:

$$\dfrac{\mathsf{D}_1 \vdash^{\mathsf{0}} e_1 : A \qquad \mathsf{D}_2, a{:}A \vdash^{T} e_2{:}B}{\mathsf{D}_1, \mathsf{D}_2 \vdash^{T} \mathtt{LET}\, a = e_1\ \mathtt{IN}\, e_2 : B}$$

Corresponding typing derivation:

$$\dfrac{\dfrac{}{[\![e_1]\!]_q \vdash \overline{\llbracket \mathsf{D}_1 \rrbracket}, q{:}\llbracket A \rrbracket \otimes \mathbf{1}} \qquad \dfrac{\dfrac{\dfrac{}{[\![e_2]\!]_{y,x} \vdash \overline{\llbracket \mathsf{D}_2 \rrbracket}, a{:}\overline{\llbracket A \rrbracket}, y{:}\&(\llbracket B \rrbracket \otimes ((\overline{\llbracket T \rrbracket \boxplus \llbracket B \rrbracket}) \otimes \mathbf{1})), x{:}\llbracket T \rrbracket \boxplus \llbracket B \rrbracket}}{q.\mathtt{close}; [\![e_2]\!]_{y,x} \vdash q{:}\bot, \overline{\llbracket \mathsf{D}_2 \rrbracket}, a{:}\overline{\llbracket A \rrbracket}, y{:}\&(\llbracket B \rrbracket \otimes ((\overline{\llbracket T \rrbracket \boxplus \llbracket B \rrbracket}) \otimes \mathbf{1})), x{:}\llbracket T \rrbracket \boxplus \llbracket B \rrbracket}}{q(a).q.\mathtt{close}; [\![e_2]\!]_{y,x} \vdash q{:}\overline{\llbracket A \rrbracket} \parr \bot, \overline{\llbracket \mathsf{D}_2 \rrbracket}, y{:}\&(\llbracket B \rrbracket \otimes ((\overline{\llbracket T \rrbracket \boxplus \llbracket B \rrbracket}) \otimes \mathbf{1})), x{:}\llbracket T \rrbracket \boxplus \llbracket B \rrbracket}}{(\nu q)([\![e_1]\!]_q \mid q(a).q.\mathtt{close}; [\![e_2]\!]_{y,x}) \vdash \overline{\llbracket \mathsf{D}_1 \rrbracket}, \overline{\llbracket \mathsf{D}_2 \rrbracket}, y{:}\&(\llbracket B \rrbracket \otimes ((\overline{\llbracket T \rrbracket \boxplus \llbracket B \rrbracket}) \otimes \mathbf{1})), x{:}\llbracket T \rrbracket \boxplus \llbracket B \rrbracket}$$

$$\boxed{\left\llbracket \dfrac{\mathsf{D}_1 \vdash^{\mathsf{0}} e_1 : A \qquad \mathsf{D}_2, a{:}A \vdash^{T} e_2{:}B}{\mathsf{D}_1, \mathsf{D}_2 \vdash^{T} \mathtt{LET}\, a = e_1\ \mathtt{IN}\, e_2 : B} \right\rrbracket y, x = \dfrac{}{\begin{array}{l}(\nu q)([\![e_1]\!]_q \mid q(a).q.\mathsf{0}; [\![e_2]\!]_{y,x}) \\ \vdash \overline{\llbracket \mathsf{D}_1 \rrbracket}, \overline{\llbracket \mathsf{D}_2 \rrbracket}, y{:}\&\big(\llbracket B \rrbracket \div \overline{\llbracket T \rrbracket}\big), x{:}\llbracket T \rrbracket \boxplus \llbracket B \rrbracket\end{array}}}$$

Fig. 12. Encoding let (Part 1).

Encoding **let** (third case) - D stands for the free variables in e_2, excluding a:

$$\llbracket \mathtt{LET}\, a = e_1\ \mathtt{IN}\, e_2 \rrbracket_{y,x} = (\nu q)(\llbracket e_1 \rrbracket_{q,x} \mid q.\mathtt{some}_D; q(a).q(s).q.\mathtt{close}; \llbracket e_2 \rrbracket_{y,s})$$

Source type judgment for $\mathtt{LET}\, a = e_1\ \mathtt{IN}\, e_2$:

$$\frac{\mathsf{D}_1 \vdash^T e_1 : A \qquad \oplus\mathsf{D}_2, a{:}A \vdash^T e_2 : B}{\mathsf{D}_1, \oplus\mathsf{D}_2 \vdash^T \mathtt{LET}\, a = e_1\ \mathtt{IN}\, e_2 : B}$$

Corresponding typing derivation (we use Proposition A.1 to ensure compatibility between $\llbracket e_1 \rrbracket_{q,x}$ and $\llbracket e_2 \rrbracket_{y,s}$):

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\begin{array}{l}\llbracket e_1 \rrbracket_{q,x} \vdash \overline{\llbracket \mathsf{D}_1 \rrbracket}, q{:}\&(\llbracket A \rrbracket \otimes ((\overline{\llbracket T \rrbracket \boxplus \llbracket A \rrbracket}) \otimes \mathbf{1})),\\ \quad x{:}\llbracket T \rrbracket \boxplus \llbracket A \rrbracket\end{array}}}{\begin{array}{l}\llbracket e_1 \rrbracket_{q,x} \vdash \overline{\llbracket \mathsf{D}_1 \rrbracket}, q{:}\&(\llbracket A \rrbracket \otimes ((\overline{\llbracket T \rrbracket \boxplus \llbracket A \rrbracket \boxplus \llbracket B \rrbracket}) \otimes \mathbf{1})),\\ \quad x{:}\llbracket T \rrbracket \boxplus \llbracket A \rrbracket \boxplus \llbracket B \rrbracket\end{array}} \quad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\begin{array}{l}\llbracket e_2 \rrbracket_{y,s} \vdash \&\overline{\llbracket \mathsf{D}_2 \rrbracket}, a{:}\overline{\llbracket A \rrbracket},\\ y{:}\&(\llbracket B \rrbracket \otimes ((\overline{\llbracket T \rrbracket \boxplus \llbracket B \rrbracket}) \otimes \mathbf{1})), s{:}\llbracket T \rrbracket \boxplus \llbracket B \rrbracket\end{array}}}{\begin{array}{l}\llbracket e_2 \rrbracket_{y,s} \vdash \&\overline{\llbracket \mathsf{D}_2 \rrbracket}, a{:}\overline{\llbracket A \rrbracket},\\ y{:}\&(\llbracket B \rrbracket \otimes ((\overline{\llbracket T \rrbracket \boxplus \llbracket B \rrbracket \boxplus \llbracket A \rrbracket}) \otimes \mathbf{1})), s{:}\llbracket T \rrbracket \boxplus \llbracket A \rrbracket \boxplus \llbracket B \rrbracket\end{array}}}{\begin{array}{l}q.\mathtt{close}; \llbracket e_2 \rrbracket_{y,s} \vdash q{:}\bot, \&\overline{\llbracket \mathsf{D}_2 \rrbracket}, a{:}\overline{\llbracket A \rrbracket},\\ \qquad y{:}\&(\llbracket B \rrbracket \otimes ((\overline{\llbracket T \rrbracket \boxplus \llbracket A \rrbracket \boxplus \llbracket B \rrbracket}) \otimes \mathbf{1})),\\ \qquad s{:}\llbracket T \rrbracket \boxplus \llbracket A \rrbracket \boxplus \llbracket B \rrbracket\end{array}}}{\begin{array}{l}q(s).q.\mathtt{close}; \llbracket e_2 \rrbracket_{y,s} \vdash\\ \ q{:}(\llbracket T \rrbracket \boxplus \llbracket A \rrbracket \boxplus \llbracket B \rrbracket) \parr \bot,\\ \ \&\overline{\llbracket \mathsf{D}_2 \rrbracket}, a{:}\overline{\llbracket A \rrbracket}, y{:}\&(\llbracket B \rrbracket \otimes ((\overline{\llbracket T \rrbracket \boxplus \llbracket A \rrbracket \boxplus \llbracket B \rrbracket}) \otimes \mathbf{1}))\end{array}}}{\begin{array}{l}q(a).q(s).q.\mathtt{close}; \llbracket e_2 \rrbracket_{y,s} \vdash\\ \ q{:}\overline{\llbracket A \rrbracket} \parr ((\llbracket T \rrbracket \boxplus \llbracket A \rrbracket \boxplus \llbracket B \rrbracket) \parr \bot),\\ \ \&\overline{\llbracket \mathsf{D}_2 \rrbracket}, y{:}\&(\llbracket B \rrbracket \otimes ((\overline{\llbracket T \rrbracket \boxplus \llbracket A \rrbracket \boxplus \llbracket B \rrbracket}) \otimes \mathbf{1}))\end{array}}}{\begin{array}{l}q.\mathtt{some}_D; q(a).q(s).q.\mathtt{close}; \llbracket e_2 \rrbracket_{y,s} \vdash\\ \ q{:}\oplus(\overline{\llbracket A \rrbracket} \parr ((\llbracket T \rrbracket \boxplus \llbracket A \rrbracket \boxplus \llbracket B \rrbracket) \parr \bot)),\\ \ \&\overline{\llbracket \mathsf{D}_2 \rrbracket}, y{:}\&(\llbracket B \rrbracket \otimes ((\overline{\llbracket T \rrbracket \boxplus \llbracket A \rrbracket \boxplus \llbracket B \rrbracket}) \otimes \mathbf{1}))\end{array}}}{(\nu q)(\llbracket e_1 \rrbracket_{q,x} \mid q.\mathtt{some}_D; q(a).q(s).q.\mathtt{close}; \llbracket e_2 \rrbracket_{y,s}) \vdash \overline{\llbracket \mathsf{D}_1 \rrbracket}, \&\overline{\llbracket \mathsf{D}_2 \rrbracket}, y{:}\&(\llbracket B \rrbracket \otimes ((\overline{\llbracket T \rrbracket \boxplus \llbracket A \rrbracket \boxplus \llbracket B \rrbracket}) \otimes \mathbf{1})), x{:}\llbracket T \rrbracket \boxplus \llbracket A \rrbracket \boxplus \llbracket B \rrbracket}}{(\nu q)(\llbracket e_1 \rrbracket_{q,x} \mid q.\mathtt{some}_D; q(a).q(s).q.\mathtt{close}; \llbracket e_2 \rrbracket_{y,s}) \vdash \overline{\llbracket \mathsf{D}_1 \rrbracket}, \&\overline{\llbracket \mathsf{D}_2 \rrbracket}, y{:}\&(\llbracket B \rrbracket \otimes ((\overline{\llbracket T \rrbracket \boxplus \llbracket B \rrbracket}) \otimes \mathbf{1})), x{:}\llbracket T \rrbracket \boxplus \llbracket B \rrbracket}$$

$$\boxed{\Bigg\llbracket \dfrac{\mathsf{D}_1 \vdash^T e_1 : A \qquad \oplus\mathsf{D}_2, a{:}A \vdash^T e_2 : B}{\mathsf{D}_1, \oplus\mathsf{D}_2 \vdash^T \mathtt{LET}\, a = e_1\ \mathtt{IN}\, e_2 : B} \Bigg\rrbracket y, x = \dfrac{}{\begin{array}{l}(\nu q)(\llbracket e_1 \rrbracket_{q,x} \mid q.\mathtt{some}_D; q(a).q(s).q.0; \llbracket e_2 \rrbracket_{y,s})\\ \qquad \vdash \overline{\llbracket \mathsf{D}_1 \rrbracket}, \&\overline{\llbracket \mathsf{D}_2 \rrbracket}, y{:}\&\big(\llbracket B \rrbracket \div \overline{\llbracket T \rrbracket}\big), x{:}\llbracket B \rrbracket \boxplus \llbracket T \rrbracket\end{array}}}$$

Fig. 13. Encoding let (Part 2).

A.5 Lift

See Figure 14.

Encoding **lift**:

$$\llbracket \mathtt{LIFT}\, e \rrbracket_{y,x} = (\nu q)(\llbracket e \rrbracket_q \mid q(v).q.\mathtt{close}; y.\overline{\mathtt{some}}; \overline{y}(w).([w \leftrightarrow v] \mid \overline{y}(w).([w \leftrightarrow x] \mid y.\overline{\mathtt{close}})))$$

Source type judgment for $\mathtt{LIFT}\, e$:

$$\frac{\mathsf{D} \vdash^{0} e : B}{\mathsf{D} \vdash^{T} \mathtt{LIFT}\, e : B}$$

Corresponding typing derivation (writing $c.\mathtt{0}$ instead of $c.\mathtt{close}$ and $[w|v]$ instead of $[w \leftrightarrow v]$):

$$\dfrac{\dfrac{}{\llbracket e \rrbracket_q \vdash \overline{\llbracket \mathsf{D} \rrbracket}, q{:}\llbracket B \rrbracket \otimes \mathbf{1}} \qquad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{[w|v] \vdash v{:}\overline{\llbracket B \rrbracket}, w{:}\llbracket B \rrbracket} \qquad \dfrac{\dfrac{}{[w|x] \vdash x{:}\llbracket T \rrbracket \boxplus \llbracket B \rrbracket, w{:}\overline{\llbracket T \rrbracket \boxplus \llbracket B \rrbracket}} \qquad \dfrac{}{y.\overline{\mathtt{0}} \vdash y{:}\mathbf{1}}}{\overline{y}(w).([w|x] \mid y.\overline{\mathtt{0}}) \vdash y{:}(\overline{\llbracket T \rrbracket \boxplus \llbracket B \rrbracket}) \otimes \mathbf{1}, x{:}\llbracket T \rrbracket \boxplus \llbracket B \rrbracket}}{\overline{y}(w).([w|v] \mid \overline{y}(w).([w|x] \mid y.\overline{\mathtt{0}})) \vdash v{:}\overline{\llbracket B \rrbracket}, y{:}\llbracket B \rrbracket \otimes ((\overline{\llbracket T \rrbracket \boxplus \llbracket B \rrbracket}) \otimes \mathbf{1}), x{:}\llbracket T \rrbracket \boxplus \llbracket B \rrbracket}}{y.\overline{\mathtt{some}}; \overline{y}(w).([w|v] \mid \overline{y}(w).([w|x] \mid y.\overline{\mathtt{0}})) \vdash v{:}\overline{\llbracket B \rrbracket}, y{:}\&(\llbracket B \rrbracket \otimes ((\overline{\llbracket T \rrbracket \boxplus \llbracket B \rrbracket}) \otimes \mathbf{1})), x{:}\llbracket T \rrbracket \boxplus \llbracket B \rrbracket}}{q.\mathtt{0}; y.\overline{\mathtt{some}}; \overline{y}(w).([w|v] \mid \overline{y}(w).([w|x] \mid y.\overline{\mathtt{0}})) \vdash q{:}\bot, v{:}\overline{\llbracket B \rrbracket}, y{:}\&(\llbracket B \rrbracket \otimes ((\overline{\llbracket T \rrbracket \boxplus \llbracket B \rrbracket}) \otimes \mathbf{1})), x{:}\llbracket T \rrbracket \boxplus \llbracket B \rrbracket}}{q(v).q.\mathtt{0}; y.\overline{\mathtt{some}}; \overline{y}(w).([w|v] \mid \overline{y}(w).([w|x] \mid y.\overline{\mathtt{0}})) \vdash q{:}\overline{\llbracket B \rrbracket} \parr \bot, y{:}\&(\llbracket B \rrbracket \otimes ((\overline{\llbracket T \rrbracket \boxplus \llbracket B \rrbracket}) \otimes \mathbf{1})), x{:}\llbracket T \rrbracket \boxplus \llbracket B \rrbracket}}{\begin{array}{c}(\nu q)(\llbracket e \rrbracket_q \mid q(v).q.\mathtt{0}; y.\overline{\mathtt{some}}; \overline{y}(w).([w|v] \mid \overline{y}(w).([w|x] \mid y.\overline{\mathtt{0}}))) \\ \vdash \overline{\llbracket \mathsf{D} \rrbracket}, y{:}\&(\llbracket B \rrbracket \otimes ((\overline{\llbracket T \rrbracket \boxplus \llbracket B \rrbracket}) \otimes \mathbf{1})), x{:}\llbracket T \rrbracket \boxplus \llbracket B \rrbracket\end{array}}$$

$$\boxed{\left\llbracket \dfrac{\mathsf{D} \vdash^{0} e : B}{\mathsf{D} \vdash^{T} \mathtt{LIFT}\, e : B} \right\rrbracket y, x = \dfrac{}{\begin{array}{c}(\nu q)(\llbracket e \rrbracket_q \mid q(v).q.\mathtt{0}; y.\overline{\mathtt{some}}; y\langle v \rangle.y\langle x \rangle.y.\overline{\mathtt{0}}) \\ \vdash \overline{\llbracket \mathsf{D} \rrbracket}, y{:}\&\big(\llbracket B \rrbracket \div \overline{\llbracket T \rrbracket}\big), x{:}\llbracket T \rrbracket \boxplus \llbracket B \rrbracket\end{array}}}$$

Fig. 14. Encoding lift.

A.6 Cast

See Figure 15.

A.7 Throw and Try-Catch

See Figures 16 and 17.

A.8 Fork

See Figure 18.

Encoding **casts**:

$$\llbracket \langle\!\langle v \rangle\!\rangle \rrbracket_y = y.\overline{\texttt{some}}; \overline{y}(z).(z.\texttt{some}_D; (\nu q)(\llbracket v \rrbracket_q \mid q(u).q.\texttt{close}; [u \leftrightarrow z]) \mid y.\texttt{close})$$

Source derivations for $\langle\!\langle v \rangle\!\rangle$:

$$\dfrac{\oplus\mathsf{D} \vdash^{\mathsf{0}} v : T}{\oplus\mathsf{D} \vdash^{\mathsf{0}} \langle\!\langle v \rangle\!\rangle : \oplus T}$$

Corresponding typing derivation (below, D stands for the domain of D):

$$\dfrac{\dfrac{\dfrac{}{\llbracket v \rrbracket_q \vdash \&\overline{\llbracket \mathsf{D} \rrbracket}, q : \llbracket T \rrbracket \otimes \mathbf{1}} \qquad \dfrac{\dfrac{\dfrac{}{[u \leftrightarrow z] \vdash z : \llbracket T \rrbracket, u : \overline{\llbracket T \rrbracket}}}{q.\texttt{close}; [u \leftrightarrow z] \vdash z : \llbracket T \rrbracket, u : \overline{\llbracket T \rrbracket}, q : \bot}}{q(u).q.\texttt{close}; [u \leftrightarrow z] \vdash z : \llbracket T \rrbracket, q : \overline{\llbracket T \rrbracket} \parr \bot}}{\dfrac{(\nu q)(\llbracket v \rrbracket_q \mid q(u).q.\texttt{close}; [u \leftrightarrow z]) \vdash \&\overline{\llbracket \mathsf{D} \rrbracket}, z : \llbracket T \rrbracket}{z.\texttt{some}_D; (\nu q)(\llbracket V \rrbracket_q \mid q(u).q.\texttt{close}; [u \leftrightarrow z]) \vdash \&\overline{\llbracket \mathsf{D} \rrbracket}, z : (\oplus\llbracket T \rrbracket)}} \qquad \dfrac{}{y.\overline{\texttt{close}} \vdash y : \mathbf{1}}}{\overline{y}(z).(z.\texttt{some}_D; (\nu q)(\llbracket v \rrbracket_q \mid q(u).q.\texttt{close}; [u \leftrightarrow z]) \mid y.\overline{\texttt{close}}) \vdash \&\overline{\llbracket \mathsf{D} \rrbracket}, y : (\oplus\llbracket T \rrbracket) \otimes \mathbf{1}}$$

$$\boxed{\left\llbracket \dfrac{\oplus\mathsf{D} \vdash^{\mathsf{0}} v : T}{\oplus\mathsf{D} \vdash^{\mathsf{0}} \langle\!\langle v \rangle\!\rangle : \oplus T} \right\rrbracket y = \dfrac{}{\begin{array}{l} \overline{y}(z).(z.\texttt{some}_D; (\nu q)(\llbracket v \rrbracket_q \mid q(u).q.\mathsf{0}; [u \leftrightarrow z]) \mid y.\overline{\mathsf{0}}) \\ \qquad \vdash \&\overline{\llbracket \mathsf{D} \rrbracket}, y : \uparrow(\oplus\llbracket T \rrbracket) \end{array}}}$$

Fig. 15. Encoding casts.

Source typing derivation:

$$\dfrac{\mathsf{D}_1 \vdash^{U} e_1 : T \quad z : U \vdash^{0} e_2 : T}{\mathsf{D}_1 \vdash^{0} \mathtt{TRY}\, e_1\, \mathtt{CATCH}\, z.\, e_2 : T}$$

To obtain the corresponding typing derivation, we set the following abbreviations:

$$\begin{aligned} K_1(j,y) &= j.\mathtt{case}\,\big(j(u).j.\mathtt{close}; \overline{y}(v).([u \leftrightarrow v] \mid y.\overline{\mathtt{close}})\,,\ j(z).j.\mathtt{close}; \llbracket e_2 \rrbracket_y)\big) \\ K_2(k) &= k.\mathtt{some}_{\emptyset}; k(u).k(z).k.\mathtt{close}; z.\mathtt{inl}; \overline{z}(v).([u \leftrightarrow v] \mid z.\overline{\mathtt{close}}) \end{aligned}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{[u \leftrightarrow v] \vdash u{:}\overline{\llbracket T \rrbracket}, v{:}\llbracket T \rrbracket} \quad \dfrac{}{y.\overline{\mathtt{close}} \vdash y{:}\mathbf{1}}}{\overline{y}(v).([u \leftrightarrow v] \mid y.\overline{\mathtt{close}}) \vdash u{:}\overline{\llbracket T \rrbracket}, y{:}\llbracket T \rrbracket \otimes \mathbf{1}}}{j.\mathtt{close}; \overline{y}(v).([u \leftrightarrow v] \mid y.\overline{\mathtt{close}}) \vdash j{:}\bot, u{:}\overline{\llbracket T \rrbracket}, y{:}\llbracket T \rrbracket \otimes \mathbf{1}}}{j(u).j.\mathtt{close}; \overline{y}(v).([u \leftrightarrow v] \mid y.\overline{\mathtt{close}}) \vdash j{:}\overline{\llbracket T \rrbracket} \parr \bot, y{:}\llbracket T \rrbracket \otimes \mathbf{1}} \quad \dfrac{\dfrac{\dfrac{}{\llbracket e_2 \rrbracket_y \vdash z{:}\overline{\llbracket U \rrbracket}, y{:}\llbracket T \rrbracket \otimes \mathbf{1}}}{j.\mathtt{close}; \llbracket e_2 \rrbracket_y \vdash z{:}\overline{\llbracket U \rrbracket}, j : \bot, y{:}\llbracket T \rrbracket \otimes \mathbf{1}}}{j(z).j.\mathtt{close}; \llbracket e_2 \rrbracket_y \vdash j : \overline{\llbracket U \rrbracket} \parr \bot, y{:}\llbracket T \rrbracket \otimes \mathbf{1}}}{K_1(j,y) \vdash j{:}\, \underbrace{(\overline{\llbracket T \rrbracket} \parr \bot) \with (\overline{\llbracket U \rrbracket} \parr \bot)}_{\overline{\llbracket T \rrbracket \boxplus \llbracket U \rrbracket}}, y{:}\llbracket T \rrbracket \otimes \mathbf{1}}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{[u \leftrightarrow v] \vdash u{:}\overline{\llbracket T \rrbracket}, v{:}\llbracket T \rrbracket} \quad \dfrac{}{z.\overline{\mathtt{close}} \vdash z{:}\mathbf{1}}}{\overline{z}(v).([u \leftrightarrow v] \mid z.\overline{\mathtt{close}}) \vdash u{:}\overline{\llbracket T \rrbracket}, z{:}\llbracket T \rrbracket \otimes \mathbf{1}}}{z.\mathtt{inl}; \overline{z}(v).([u \leftrightarrow v] \mid z.\overline{\mathtt{close}}) \vdash u{:}\overline{\llbracket T \rrbracket}, z{:}(\llbracket T \rrbracket \boxplus \llbracket U \rrbracket)}}{k.\mathtt{close}; z.\mathtt{inl}; \overline{z}(v).([u \leftrightarrow v] \mid z.\overline{\mathtt{close}}) \vdash k{:}\bot, u{:}\overline{\llbracket T \rrbracket}, z{:}(\llbracket T \rrbracket \boxplus \llbracket U \rrbracket)}}{k(z).k.\mathtt{close}; z.\mathtt{inl}; \overline{z}(v).([u \leftrightarrow v] \mid z.\overline{\mathtt{close}}) \vdash k{:}(\llbracket T \rrbracket \boxplus \llbracket U \rrbracket) \parr \bot, u{:}\overline{\llbracket T \rrbracket}}}{k(u).k(z).k.\mathtt{close}; z.\mathtt{inl}; \overline{z}(v).([u \leftrightarrow v] \mid z.\overline{\mathtt{close}}) \vdash k{:}\overline{\llbracket T \rrbracket} \parr ((\llbracket T \rrbracket \boxplus \llbracket U \rrbracket) \parr \bot)}}{K_2(k) \vdash k{:}\, \underbrace{\oplus(\overline{\llbracket T \rrbracket} \parr ((\llbracket T \rrbracket \boxplus \llbracket U \rrbracket) \parr \bot))}_{\overline{\with(\llbracket T \rrbracket \div \overline{\llbracket U \rrbracket})}}}$$

$$\dfrac{\dfrac{}{\llbracket e_1 \rrbracket_{k,j} \vdash \overline{\llbracket \mathsf{D}_1 \rrbracket}, k{:}\with\big(\llbracket T \rrbracket \div \overline{\llbracket U \rrbracket}\big), j{:}\llbracket T \rrbracket \boxplus \llbracket U \rrbracket} \quad \dfrac{}{K_2(k) \vdash k{:}\overline{\with\big(\llbracket T \rrbracket \div \overline{\llbracket U \rrbracket}\big)}}}{(\nu k)(\llbracket e \rrbracket_{k,j} \mid K_2(k)) \vdash \overline{\llbracket \mathsf{D}_1 \rrbracket}, j{:}\llbracket T \rrbracket \boxplus \llbracket U \rrbracket}$$

$$\dfrac{\dfrac{}{(\nu k)(\llbracket e_1 \rrbracket_{k,j} \mid K_2(k)) \vdash \overline{\llbracket \mathsf{D}_1 \rrbracket}, j{:}\llbracket T \rrbracket \boxplus \llbracket U \rrbracket} \quad \dfrac{}{K_1(j,y) \vdash j{:}\overline{\llbracket T \rrbracket \boxplus \llbracket U \rrbracket}, y{:}\llbracket T \rrbracket \otimes \mathbf{1}}}{\begin{array}{l} (\nu j)((\nu k)(\llbracket e_1 \rrbracket_{k,j} \mid k.\mathtt{some}_{\emptyset}; k(u).k(z).k.\mathtt{close}; z.\mathtt{inl}; \overline{z}(v).([u \leftrightarrow v] \mid z.\overline{\mathtt{close}})) \mid \\ \quad j.\mathtt{case}\,\big(j(u).j.\mathtt{close}; \overline{y}(v).([u \leftrightarrow v] \mid y.\overline{\mathtt{close}})\,,\ j(z).j.\mathtt{close}; \llbracket e_2 \rrbracket_y)\big) \vdash \overline{\llbracket \mathsf{D}_1 \rrbracket}, y{:}\llbracket T \rrbracket \otimes \mathbf{1} \end{array}}$$

$$\boxed{\left\llbracket \dfrac{\mathsf{D}_1 \vdash^{U} e_1 : T \quad z : U \vdash^{0} e_2 : T}{\mathsf{D}_1 \vdash^{0} \mathtt{TRY}\, e_1\, \mathtt{CATCH}\, z.\, e_2 : T} \right\rrbracket y = \begin{array}{l} \overline{\begin{array}{l} (\nu j)((\nu k)(\llbracket e_1 \rrbracket_{k,j} \mid k.\mathtt{some}_{\emptyset}; k(u).k(z).k.0; z.\mathtt{inl}; z\langle u \rangle.z.\overline{0}) \mid \\ \qquad j.\mathtt{case}\,\big(j(u).j.0; y\langle u \rangle.y.\overline{0}\,,\ j(z).j.0; \llbracket e_2 \rrbracket_y)\big) \end{array}} \\ \vdash \overline{\llbracket \mathsf{D}_1 \rrbracket}, y{:} \uparrow \llbracket T \rrbracket \end{array}}$$

Fig. 16. Encoding try-catch exceptions.

Encoding **throw**:

$$\llbracket \texttt{THROW}\, v \rrbracket_{y,x} = y.\overline{\texttt{none}}; x.\texttt{inr}; \overline{x}(u).([v \leftrightarrow u] \mid x.\overline{\texttt{close}})$$

Source derivations for $\texttt{TRY}\, e$:

$$\frac{\mathsf{D} \vdash^{0} v : T}{\mathsf{D} \vdash^{T} \texttt{THROW}\, v : U}$$

Corresponding typing derivation:

$$\frac{\dfrac{}{y.\overline{\texttt{none}} \vdash y{:}\&\big(\llbracket T \rrbracket \div \overline{\llbracket U \rrbracket}\big)} \qquad \dfrac{\dfrac{}{\overline{x}(w).([w \leftrightarrow v] \mid x.\overline{\texttt{close}}) \vdash v{:}\overline{\llbracket T \rrbracket}, x{:}\llbracket T \rrbracket \otimes \mathbf{1}}}{x.\texttt{inr}; \overline{x}(w).([w \leftrightarrow v] \mid x.\overline{\texttt{close}}) \vdash v{:}\overline{\llbracket T \rrbracket}, x{:}\, \underbrace{(\llbracket U \rrbracket \otimes \mathbf{1}) \oplus (\llbracket T \rrbracket \otimes \mathbf{1})}_{\llbracket U \rrbracket \boxplus \llbracket T \rrbracket}}}{y.\overline{\texttt{none}} \mid x.\texttt{inr}; \overline{x}(w).([w \leftrightarrow v] \mid x.\overline{\texttt{close}}) \vdash v{:}\overline{\llbracket T \rrbracket}, y{:}\&\big(\llbracket T \rrbracket \div \overline{\llbracket U \rrbracket}\big), x{:}\llbracket T \rrbracket \boxplus \llbracket U \rrbracket}$$

$$\boxed{\left\llbracket \frac{\mathsf{D} \vdash^{0} v : T}{\mathsf{D} \vdash^{T} \texttt{THROW}\, v : U} \right\rrbracket y, x = \frac{}{y.\overline{\texttt{none}} \mid x.\texttt{inr}; x\langle v \rangle.x.\overline{\mathsf{0}} \vdash v{:}\overline{\llbracket T \rrbracket}, y{:}\&\big(\llbracket T \rrbracket \div \overline{\llbracket U \rrbracket}\big), x{:}\llbracket T \rrbracket \boxplus \llbracket U \rrbracket}}$$

Fig. 17. Encoding throw.

Encoding **fork**:

$$\llbracket \texttt{FORK}\, x.e \rrbracket_y = \overline{y}(x).((\nu q)(\llbracket e \rrbracket_q \mid q(u).q.\texttt{close}; [u \leftrightarrow v]) \mid y.\overline{\texttt{close}})$$

Source type judgment for $\texttt{FORK}\, x$:

$$\frac{\mathsf{D}, x : S \vdash^{0} e : \mathbf{unit}}{\mathsf{D} \vdash^{0} \texttt{FORK}\, x.e : \overline{S}}$$

Corresponding typing derivation:

$$\frac{\dfrac{\dfrac{}{\llbracket e \rrbracket_q \vdash \overline{\llbracket \mathsf{D} \rrbracket}, x{:}\overline{\llbracket S \rrbracket}, q{:}\mathbf{1} \otimes \mathbf{1}} \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{\mathbf{0} \vdash \cdot}}{u.\texttt{close}; \mathbf{0} \vdash u{:}\bot}}{q.\texttt{close}; u.\texttt{close}; \mathbf{0} \vdash q{:}\bot, u{:}\bot}}{q(u).q.\texttt{close}; u.\texttt{close}; \mathbf{0} \vdash q{:}\bot \parr \bot}}{(\nu q)(\llbracket e \rrbracket_q \mid q(u).q.\texttt{close}; u.\texttt{close}; \mathbf{0}) \vdash \overline{\llbracket \mathsf{D} \rrbracket}, x{:}\overline{\llbracket S \rrbracket}} \qquad \dfrac{}{y.\overline{\texttt{close}} \vdash y{:}\mathbf{1}}}{\overline{y}(x).((\nu q)(\llbracket e \rrbracket_q \mid q(u).q.\texttt{close}; u.\texttt{close}; \mathbf{0}) \mid y.\overline{\texttt{close}}) \vdash \overline{\llbracket \mathsf{D} \rrbracket}, y{:}\overline{\llbracket S \rrbracket} \otimes \mathbf{1}}$$

$$\boxed{\left\llbracket \frac{\mathsf{D}, x : S \vdash^{0} e : \mathbf{unit}}{\mathsf{D} \vdash^{0} \texttt{FORK}\, x.e : \overline{S}} \right\rrbracket y = \frac{}{\overline{y}(x).((\nu q)(\llbracket e \rrbracket_q \mid \mathsf{S}_q) \mid y.\overline{\mathsf{0}}) \vdash \overline{\llbracket \mathsf{D} \rrbracket}, y{:}\uparrow\llbracket \overline{S} \rrbracket}}$$

Fig. 18. Encoding fork.

A.9 Some and None

See Figures 19 and 20.

A.10 Send and Receive

See Figure 21.

A.11 Non Deterministic Choice

See Figure 22.

A.12 Close

See Figure 23.

A.13 Unit

See Figure 24.

Encoding **some** (output):

$$\llbracket \texttt{SOME}\,!\,z; e\rrbracket_{y,x} = z.\overline{\texttt{some}};\llbracket e\rrbracket_{y,x}$$

Source type judgment for $\texttt{SOME}\,!\,z; .e$:

$$\frac{\mathsf{D}, z : S \vdash^{U} e : T}{\mathsf{D}, z : \oplus S \vdash^{U} \texttt{SOME}\,!\,z; .e : T}$$

Corresponding type derivation:

$$\dfrac{\dfrac{}{\llbracket e\rrbracket_{y,x} \vdash \overline{\llbracket \mathsf{D}\rrbracket}, z{:}\overline{\llbracket S\rrbracket}, y{:}\&(\llbracket T\rrbracket \otimes ((\overline{\llbracket U\rrbracket \boxplus \llbracket T\rrbracket}) \otimes \mathbf{1})), x{:}\llbracket U\rrbracket \boxplus \llbracket T\rrbracket}}{z.\overline{\texttt{some}};\llbracket e\rrbracket_{y,x} \vdash \overline{\llbracket \mathsf{D}\rrbracket}, z{:}\&\overline{\llbracket S\rrbracket}, y{:}\&(\llbracket T\rrbracket \otimes ((\overline{\llbracket U\rrbracket \boxplus \llbracket T\rrbracket}) \otimes \mathbf{1})), x{:}\llbracket U\rrbracket \boxplus \llbracket T\rrbracket}$$

$$\boxed{\left\llbracket \frac{\mathsf{D}, z : S \vdash^{U} e : T}{\mathsf{D}, z : \oplus S \vdash^{U} \texttt{SOME}\,!\,z; e : T}\right\rrbracket y, x = \frac{}{z.\overline{\texttt{some}};\llbracket e\rrbracket_{y,x} \vdash \overline{\llbracket \mathsf{D}\rrbracket}, z{:}\&\overline{\llbracket S\rrbracket}, y{:}\&\big(\llbracket T\rrbracket \div \overline{\llbracket U\rrbracket}\big), x{:}\llbracket U\rrbracket \boxplus \llbracket T\rrbracket}}$$

Encoding **some** (input):

$$\llbracket \texttt{SOME}\,?\,z; e\rrbracket_{y} = z.\texttt{some}_D;(\nu q)(\llbracket e\rrbracket_q \mid q?(u).u.\texttt{close};q.\texttt{close};\mathbf{0}) \mid \overline{y}(v).(v.\overline{\texttt{close}} \mid y.\overline{\texttt{close}})$$

Source type judgment for $\texttt{SOME}\,?\,z; .e$:

$$\frac{\oplus\mathsf{D}, z : S \vdash^{0} e : \mathbf{unit}}{\oplus\mathsf{D}, z : \&S \vdash^{0} \texttt{SOME}\,?\,z; e : \mathbf{unit}}$$

Corresponding type derivation (below, D stands for the domain of D):

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\llbracket e\rrbracket_q \vdash \&\overline{\llbracket \mathsf{D}\rrbracket}, q{:}\mathbf{1}\otimes\mathbf{1}, z{:}\overline{\llbracket S\rrbracket}} \qquad \dfrac{\dfrac{\dfrac{}{q.\texttt{close};\mathbf{0} \vdash q{:}\bot}}{u.\texttt{close};q.\texttt{close};\mathbf{0} \vdash q{:}\bot, u{:}\bot}}{q?(u).u.\texttt{close};q.\texttt{close};\mathbf{0} \vdash q{:}\bot \parr \bot}}{(\nu q)(\llbracket e\rrbracket_q \mid q?(u).u.\texttt{close};q.\texttt{close};\mathbf{0}) \vdash \&\overline{\llbracket \mathsf{D}\rrbracket}, z{:}\overline{\llbracket S\rrbracket}}}{z.\texttt{some}_D;(\nu q)(\llbracket e\rrbracket_q \mid q?(u).u.\texttt{close};q.\texttt{close};\mathbf{0}) \vdash \&\overline{\llbracket \mathsf{D}\rrbracket}, z{:}\oplus\overline{\llbracket S\rrbracket}} \qquad \dfrac{\dfrac{}{v.\overline{\texttt{close}} \vdash v{:}\mathbf{1}} \qquad \dfrac{}{y.\overline{\texttt{close}} \vdash y{:}\mathbf{1}}}{\overline{y}(v).(v.\overline{\texttt{close}} \mid y.\overline{\texttt{close}}) \vdash y{:}\mathbf{1}\otimes\mathbf{1}}}{z.\texttt{some}_D;(\nu q)(\llbracket e\rrbracket_q \mid q?(u).u.\texttt{close};q.\texttt{close};\mathbf{0}) \mid \overline{y}(v).(v.\overline{\texttt{close}} \mid y.\overline{\texttt{close}}) \vdash \&\overline{\llbracket \mathsf{D}\rrbracket}, z{:}\oplus\overline{\llbracket S\rrbracket}, y{:}\mathbf{1}\otimes\mathbf{1}}$$

$$\boxed{\left\llbracket \frac{\oplus\mathsf{D}, z : S \vdash^{0} e : \mathbf{unit}}{\oplus\mathsf{D}, z : \&S \vdash^{0} \texttt{SOME}\,?\,z; e : \mathbf{unit}}\right\rrbracket y, x = \frac{}{\begin{array}{c} z.\texttt{some}_D;(\nu q)(\llbracket e\rrbracket_q \mid \mathsf{S}_q) \mid \overline{y}(v).(v.\overline{\mathsf{0}} \mid y.\overline{\mathsf{0}}) \\ \vdash \&\overline{\llbracket \mathsf{D}\rrbracket}, z{:}\oplus\overline{\llbracket S\rrbracket}, y{:}\uparrow\llbracket \mathbf{unit}\rrbracket \end{array}}}$$

Fig. 19. Encoding some.

Encoding **none**:

$$[\mathtt{NONE}\,!\,z; e]_{y,x} = z.\overline{\mathtt{none}}; [e]_{y,x}$$

Source type judgment for $\mathtt{NONE}\,!\,z; e$:

$$\frac{\mathsf{D} \vdash^{U} e : T}{\mathsf{D}, z : \oplus S \vdash^{U} \mathtt{NONE}\,!\,z; e : T}$$

Corresponding type derivation:

$$\frac{\dfrac{}{z.\overline{\mathtt{none}} \vdash z{:}\&\overline{\llbracket S \rrbracket}} \qquad \dfrac{}{[e]_{y,x} \vdash \overline{\llbracket \mathsf{D} \rrbracket}, y{:}\&\big(\llbracket T \rrbracket \div \overline{\llbracket U \rrbracket}\big), x{:}\llbracket T \rrbracket \boxplus \llbracket U \rrbracket}}{z.\overline{\mathtt{none}} \mid [e]_{y,x} \vdash \overline{\llbracket \mathsf{D} \rrbracket}, z{:}\&\overline{\llbracket S \rrbracket}, y{:}\&\big(\llbracket T \rrbracket \div \overline{\llbracket U \rrbracket}\big), x{:}\llbracket T \rrbracket \boxplus \llbracket U \rrbracket}$$

$$\boxed{\left\llbracket \frac{\mathsf{D} \vdash^{U} e : T}{\mathsf{D}, z : \oplus S \vdash^{U} \mathtt{NONE}\,!\,z; e : T} \right\rrbracket y, x = \frac{}{z.\overline{\mathtt{none}}; [e]_{y,x} \vdash \overline{\llbracket \mathsf{D} \rrbracket}, z{:}\&\overline{\llbracket S \rrbracket}, y{:}\&\big(\llbracket T \rrbracket \div \overline{\llbracket U \rrbracket}\big), x{:}\llbracket T \rrbracket \boxplus \llbracket U \rrbracket}}$$

Fig. 20. Encoding none.

Encoding **send**:

$$\llbracket \texttt{SEND}(c,a); e \rrbracket_{y,x} = \overline{c}(a_1).([a \leftrightarrow a_1] \mid \llbracket e \rrbracket_{y,x})$$

Source type judgment for $\texttt{SEND}(c,a); e$:

$$\frac{\mathsf{D} \vdash^{0} a : T_1 \qquad \mathsf{D}', c : T_2 \vdash^{U} e : T}{\mathsf{D}, \mathsf{D}', c : !T_1.T_2 \vdash^{U} \texttt{SEND}(c,a); e : T}$$

Corresponding typing derivation:

$$\frac{\dfrac{}{[a \leftrightarrow a_1] \vdash a{:}\overline{\llbracket T_1 \rrbracket}, a_1{:}\llbracket T_1 \rrbracket} \qquad \dfrac{}{\llbracket e \rrbracket_{y,x} \vdash \overline{\llbracket \mathsf{D}' \rrbracket}, c{:}\overline{\llbracket T_2 \rrbracket}, y{:}\&\big(\llbracket T \rrbracket \div \overline{\llbracket U \rrbracket}\big), x{:}\llbracket T \rrbracket \boxplus \llbracket U \rrbracket}}{\overline{c}(a_1).([a \leftrightarrow a_1] \mid \llbracket e \rrbracket^{U}_{y,x}) \vdash \overline{\llbracket \mathsf{D}' \rrbracket}, c{:}\llbracket T_1 \rrbracket \otimes \overline{\llbracket T_2 \rrbracket}, y{:}\&\big(\llbracket T \rrbracket \div \overline{\llbracket U \rrbracket}\big), x{:}\llbracket T \rrbracket \boxplus \llbracket U \rrbracket, a{:}\overline{\llbracket T_1 \rrbracket}}$$

$$\boxed{\left\llbracket \frac{\mathsf{D} \vdash^{0} a : T_1 \qquad \mathsf{D}', c : T_2 \vdash^{U} e : T}{\mathsf{D}, \mathsf{D}', c : !T_1.T_2 \vdash^{U} \texttt{SEND}(c,a); e : T} \right\rrbracket y, x = \begin{array}{r} \overline{} \\ c\langle a \rangle.\llbracket e \rrbracket_{y,x} \vdash \overline{\llbracket \mathsf{D}' \rrbracket}, c{:}\llbracket T_1 \rrbracket \otimes \overline{\llbracket T_2 \rrbracket}, \\ y{:}\&\big(\llbracket T \rrbracket \div \overline{\llbracket U \rrbracket}\big), x{:}\llbracket T \rrbracket \boxplus \llbracket U \rrbracket, a{:}\overline{\llbracket T_1 \rrbracket} \end{array}}$$

Encoding **receive**:

$$\llbracket \texttt{RECV}(c,z); e \rrbracket_{y,x} = c(z).\llbracket e \rrbracket_{y,x}$$

Source type judgment for $\texttt{RECV}(c,z); e$:

$$\frac{\mathsf{D}, z : T_1, c : T_2 \vdash^{U} e : T}{\mathsf{D}, c : ?T_1.T_2 \vdash^{U} \texttt{RECV}(c,z); e : T}$$

Corresponding typing derivation:

$$\frac{\dfrac{}{\llbracket e \rrbracket_{y,x} \vdash \overline{\llbracket \mathsf{D} \rrbracket}, z{:}\overline{\llbracket T_1 \rrbracket}, c{:}\overline{\llbracket T_2 \rrbracket}, y{:}\&\big(\llbracket T \rrbracket \div \overline{\llbracket U \rrbracket}\big), x{:}\llbracket T \rrbracket \boxplus \llbracket U \rrbracket}}{c(z).\llbracket e \rrbracket_{y,x} \vdash \overline{\llbracket \mathsf{D} \rrbracket}, c{:}\overline{\llbracket T_1 \rrbracket} \parr \overline{\llbracket T_2 \rrbracket}, y{:}\&\big(\llbracket T \rrbracket \div \overline{\llbracket U \rrbracket}\big), x{:}\llbracket T \rrbracket \boxplus \llbracket U \rrbracket}$$

$$\boxed{\left\llbracket \frac{\mathsf{D}, z : T_1, c : T_2 \vdash^{U} e : T}{\mathsf{D}, c : ?T_1.T_2 \vdash^{U} \texttt{RECV}(c,z); e : T} \right\rrbracket y, x = \begin{array}{r} \overline{} \\ c(z).\llbracket e \rrbracket_{y,x} \vdash \overline{\llbracket \mathsf{D} \rrbracket}, c{:}\overline{\llbracket T_1 \rrbracket} \parr \overline{\llbracket T_2 \rrbracket}, \\ y{:}\&\big(\llbracket T \rrbracket \div \overline{\llbracket U \rrbracket}\big), x{:}\llbracket T \rrbracket \boxplus \llbracket U \rrbracket \end{array}}$$

Fig. 21. Encoding send and receive.

Encoding **non determinism**:

$$[\![e_1 \oplus e_2]\!]_{y,x} = [\![e_1]\!]_{y,x} \oplus [\![e_2]\!]_{y,x}$$

Source type judgment for $e_1 \oplus e_2$:

$$\frac{\oplus\mathsf{D} \vdash^{\mathsf{0}} e_1 : \mathbf{unit} \quad \oplus\mathsf{D} \vdash^{\mathsf{0}} e_2 : \mathbf{unit}}{\oplus\mathsf{D} \vdash^{\mathsf{0}} e_1 \oplus e_2 : \mathbf{unit}}$$

Corresponding typing derivation (writing $c.\mathsf{0}$ instead of $c.\texttt{close}$):

$$\dfrac{\dfrac{\dfrac{\dfrac{}{[\![e_1]\!]_z \vdash \& \overline{[\![\mathsf{D}]\!]}, z{:}\mathbf{1} \otimes \mathbf{1}} \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{\mathbf{0} \vdash \cdot}}{u.\mathsf{0}; \mathbf{0} \vdash u{:}\bot}}{z.\mathsf{0}; u.\mathsf{0}; \mathbf{0} \vdash z{:}\bot, u{:}\bot}}{z(u).q.\mathsf{0}; u.\mathsf{0}; \mathbf{0} \vdash z{:}\bot \parr \bot}}{(\nu z)([\![e_1]\!]_z \mid z(u).z.\mathsf{0}; u.\mathsf{0}; \mathbf{0}) \vdash \& \overline{[\![\mathsf{D}]\!]}} \qquad \dfrac{\text{(as in the LHS)}}{(\nu z)([\![e_2]\!]_z \mid z(u).z.\mathsf{0}; u.\mathsf{0}; \mathbf{0}) \vdash \& \overline{[\![\mathsf{D}]\!]}}}{(\nu z)([\![e_1]\!]_z \mid z(u).z.\mathsf{0}; u.\mathsf{0}; \mathbf{0}) \oplus (\nu z)([\![e_2]\!]_z \mid z(u).z.\mathsf{0}; u.\mathsf{0}; \mathbf{0}) \vdash \& \overline{[\![\mathsf{D}]\!]}} \qquad \dfrac{}{[\![*]\!]_y \vdash y{:} \uparrow [\![\mathbf{unit}]\!]}}{((\nu z)([\![e_1]\!]_z \mid z(u).z.\mathsf{0}; u.\mathsf{0}; \mathbf{0}) \oplus (\nu z)([\![e_2]\!]_z \mid z(u).z.\mathsf{0}; u.\mathsf{0}; \mathbf{0})) \mid [\![*]\!]_y \vdash \overline{[\![\mathsf{D}]\!]}, y{:} \uparrow [\![\mathbf{unit}]\!]}$$

$$\boxed{\left[\!\!\left[\frac{\oplus\mathsf{D} \vdash^{\mathsf{0}} e_1 : \mathbf{unit} \quad \oplus\mathsf{D} \vdash^{\mathsf{0}} e_2 : \mathbf{unit}}{\oplus\mathsf{D} \vdash^{\mathsf{0}} e_1 \oplus e_2 : \mathbf{unit}}\right]\!\!\right] y = \frac{}{((\nu z)([\![e_1]\!]_z \mid \mathsf{S}_z) \oplus (\nu z)([\![e_2]\!]_z \mid \mathsf{S}_z)) \mid [\![*]\!]_y \vdash \& \overline{[\![\mathsf{D}]\!]}, y{:} \uparrow [\![\mathbf{unit}]\!]}}$$

Fig. 22. Encoding non-determinism.

Encoding **close** (output):

$$\llbracket \texttt{CLOSE}\,!\,c \rrbracket_{y,x} = x.\overline{\texttt{close}} \mid y.\overline{\texttt{some}}; \overline{y}(r).(r.\overline{\texttt{close}} \mid \overline{y}(c_1).([c \leftrightarrow c_1] \mid y.\overline{\texttt{close}}))$$

Source type judgment for $\texttt{CLOSE}\,!\,c$:

$$\dfrac{\mathsf{D} \vdash^{T} e : \mathbf{unit}}{\mathsf{D}, c : \mathsf{end}_{!} \vdash^{T} \texttt{CLOSE}\,!\,c; e : \mathbf{unit}}$$

Corresponding typing derivation for $\llbracket \texttt{CLOSE}\,!\,c \rrbracket_{y,x}$:

$$\dfrac{\dfrac{}{c.\overline{\texttt{close}} \vdash c{:}\mathbf{1}} \qquad \dfrac{}{\llbracket e \rrbracket_{y,x} \vdash y{:}\mathbin{\&}\big(\llbracket\mathbf{unit}\rrbracket \div \overline{\llbracket T\rrbracket}\big), x{:}\llbracket\mathbf{unit}\rrbracket \boxplus \llbracket T\rrbracket}}{c.\overline{\texttt{close}} \mid \llbracket e \rrbracket_{y,x} \vdash c{:}\mathbf{1}, y{:}\mathbin{\&}\big(\llbracket\mathbf{unit}\rrbracket \div \overline{\llbracket T\rrbracket}\big), x{:}\llbracket\mathbf{unit}\rrbracket \boxplus \llbracket T\rrbracket}$$

$$\boxed{\left\llbracket \dfrac{\mathsf{D} \vdash^{T} e : \mathbf{unit}}{\mathsf{D}, c : \mathsf{end}_{!} \vdash^{T} \texttt{CLOSE}\,!\,c; e : \mathbf{unit}} \right\rrbracket y, x = \dfrac{}{c.\overline{0} \mid \llbracket e \rrbracket_{y,x} \vdash c{:}\llbracket\mathsf{end}_{!}\rrbracket, y{:}\mathbin{\&}\big(\llbracket\mathbf{unit}\rrbracket \div \overline{\llbracket T\rrbracket}\big), x{:}\llbracket\mathbf{unit}\rrbracket \boxplus \llbracket T\rrbracket}}$$

Encoding **close** (input):

$$\llbracket \texttt{CLOSE}\,?\,c \rrbracket_{y,x} = c.\texttt{close}; \mathbf{0} \mid y.\overline{\texttt{some}}; \overline{y}(r).(r.\overline{\texttt{close}} \mid \overline{y}(x_1).([x \leftrightarrow x_1] \mid y.\overline{\texttt{close}}))$$

Source type judgment for $\texttt{CLOSE}\,?\,x$:

$$\dfrac{}{c : \mathsf{end}_{?} \vdash^{T} \texttt{CLOSE}\,?\,c : \mathbf{unit}}$$

Corresponding typing derivation for $\llbracket \texttt{CLOSE}\,?\,x \rrbracket_{y,c}$ (writing $c.\mathsf{0}$ instead of $c.\texttt{close}$):

$$\dfrac{\dfrac{}{c.\mathsf{0}; \mathbf{0} \vdash c{:}\bot} \qquad \dfrac{\dfrac{\dfrac{}{r.\overline{0} \vdash r{:}\mathbf{1}} \qquad \dfrac{\dfrac{}{[x \leftrightarrow x_1] \vdash x{:}\llbracket T\rrbracket \boxplus \llbracket\mathbf{unit}\rrbracket, x_1{:}\overline{\llbracket T\rrbracket \boxplus \llbracket\mathbf{unit}\rrbracket}} \qquad \dfrac{}{y.\overline{0} \vdash y : \mathbf{1}}}{\overline{y}(x_1).([x \leftrightarrow x_1] \mid y.\overline{0}) \vdash y{:}(\overline{\llbracket T\rrbracket \boxplus \llbracket\mathbf{unit}\rrbracket}) \otimes \mathbf{1}, x{:}\llbracket T\rrbracket \boxplus \llbracket\mathbf{unit}\rrbracket}}{\overline{y}(r).(r.\overline{0} \mid \overline{y}(x_1).([x \leftrightarrow x_1] \mid y.\overline{0})) \vdash y{:}\mathbf{1} \otimes ((\overline{\llbracket T\rrbracket \boxplus \llbracket\mathbf{unit}\rrbracket}) \otimes \mathbf{1}), x{:}\llbracket T\rrbracket \boxplus \llbracket\mathbf{unit}\rrbracket}}{y.\texttt{some}; \overline{y}(r).(r.\overline{0} \mid \overline{y}(x_1).([x \leftrightarrow x_1] \mid y.\overline{0})) \vdash y{:}\mathbin{\&}(\mathbf{1} \otimes ((\overline{\llbracket T\rrbracket \boxplus \llbracket\mathbf{unit}\rrbracket}) \otimes \mathbf{1})), x{:}\llbracket T\rrbracket \boxplus \llbracket\mathbf{unit}\rrbracket}}{c.\mathsf{0}; \mathbf{0} \mid y.\overline{\texttt{some}}; \overline{y}(r).(r.\overline{0} \mid \overline{y}(x_1).([x \leftrightarrow x_1] \mid y.\overline{0})) \vdash c{:}\bot, y{:}\mathbin{\&}\big(\llbracket\mathbf{unit}\rrbracket \div \overline{\llbracket T\rrbracket}\big), x{:}\llbracket T\rrbracket \boxplus \llbracket\mathbf{unit}\rrbracket}$$

$$\boxed{\left\llbracket \dfrac{}{c : \mathsf{end}_{?} \vdash^{T} \texttt{CLOSE}\,?\,c : \mathbf{unit}} \right\rrbracket y, c = \dfrac{}{\begin{array}{l} c.\mathsf{0}; \mathbf{0} \mid y.\overline{\texttt{some}}; \overline{y}(r).(r.\overline{0} \mid y\langle x\rangle.y.\overline{0}) \\ \quad \vdash c{:}\llbracket\mathsf{end}_{?}\rrbracket, y{:}\mathbin{\&}\big(\llbracket\mathbf{unit}\rrbracket \div \overline{\llbracket T\rrbracket}\big), x{:}\llbracket T\rrbracket \boxplus \llbracket\mathbf{unit}\rrbracket \end{array}}}$$

Fig. 23. Encoding close.

Encoding **unit**:

$$\llbracket * \rrbracket_y = \overline{y}(u).(u.\overline{\texttt{close}} \mid y.\overline{\texttt{close}})$$

Source type judgment for $*$:

$$\frac{}{\cdot \vdash^{0} * : \mathbf{unit}}$$

Corresponding typing derivation:

$$\frac{\dfrac{}{u.\overline{\texttt{close}} \vdash u{:}\mathbf{1}} \qquad \dfrac{}{y.\overline{\texttt{close}} \vdash y{:}\mathbf{1}}}{\overline{y}(u).(u.\overline{\texttt{close}} \mid y.\overline{\texttt{close}}) \vdash y{:}\mathbf{1} \otimes \mathbf{1}}$$

$$\boxed{\left\llbracket \frac{}{\cdot \vdash^{0} * : \mathbf{unit}} \right\rrbracket y = \frac{}{\overline{y}(u).(u.\overline{\mathsf{0}} \mid y.\overline{\mathsf{0}}) \vdash y{:} \uparrow \llbracket \mathbf{unit} \rrbracket}}$$

Fig. 24. Encoding unit.