

Session-typed Staged Metaprogramming

Pedro Ângelo 


LASIGE, Faculdade de Ciências da Universidade de Lisboa, Portugal

Atsushi Igarashi 

Kyoto University, Kyoto, Japan

Yuito Murase 

Kyoto University, Kyoto, Japan

Vasco T. Vasconcelos 

LASIGE, Faculdade de Ciências da Universidade de Lisboa, Portugal

Abstract

We propose the integration of staged metaprogramming into a session-typed message passing functional language. We build on a model of contextual modal type theory with multi-level contexts, where contextual values, closing arbitrary terms over a series of variables, may then be boxed and transmitted in messages. Once received, one such value may then be unboxed and locally applied before being run. To motivate this integration, we present examples of real-world use cases, for which our system would be suitable, such as servers preparing and shipping code on demand via session typed messages. We present a type system that distinguishes linear (used exactly once) from unrestricted (used an unbounded number of times) resources, and further define an algorithmic type checker, suitable for a concrete implementation. We show type preservation, a progress result for sequential computations and absence of runtime errors for the concurrent runtime, as well as correctness properties for the algorithmic type checker hold.

2012 ACM Subject Classification Theory of computation → Type theory; Theory of computation → Process calculi

Keywords and phrases linear types, session types, contextual modal types, metaprogramming, staged computation

Digital Object Identifier 10.4230/LIPIcs.CVIT.2016.23

1 Introduction

Metaprogramming manipulates code in order to generate and evaluate code at runtime, allowing in particular to explore the availability of certain arguments to functions in order to save computational effort. In this paper we are interested in programming languages where the code produced is typed by construction and where code may refer to a context providing types for the free variables, commonly known as contextual typing [27, 32, 33, 34]. On an orthogonal axis, session types have been advocated as a means to discipline concurrent computations, by accurately describing protocols for the channels used to exchange messages between processes [11, 23, 24, 25, 47, 48, 50].

The integration of session types with metaprogramming allows to setup code-producing servers that run in parallel with the rest of the program and provide code on demand, exchanged on typed channels. Linearity is central to session types, but current metaprogramming models lack support for such a feature. We extend a simple model of contextual modal type theory (with monomorphic contexts) with support for session types, to obtain a call-by-value linear lambda calculus with multi-level contexts.

Our development is based on Davies and Pfenning [20], where we use a box modality to distinguish generated code. We further allow code to refer to variables in a context, described by contextual types, along the lines of Nanevski et al. [34]. Mœbius [27] further adds to



© Jane Open Access and Joan R. Public;
licensed under Creative Commons License CC-BY 4.0

42nd Conference on Very Important Topics (CVIT 2016).

Editors: John Q. Open and Joan R. Access; Article No. 23; pp. 23:1–23:29

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

modal contextual type theory the provision for pattern matching on code, for generating polymorphic code, and for generating code that depends on other code fragments. We forgo the first two directions, and base our development on the last. We propose a multilevel contextual modal linear lambda calculus with support for session types, where in particular the composition of code fragments avoids creating extraneous administrative redexes due to boxing and unboxing. An alternative starting point would have been the Fitch- or Kripke-style formulation, providing for the Lisp quote/unquote, where typing contexts are viewed as stacks modeling the different stages of computation [18, 32, 49]. It seemed to us that the let-box approach would simplify the extension to the linear setting and to session types.

The rest of the paper is structured as follows. In Section 2 we motivate our proposal by means of examples. Section 3 introduces terms, processes, term evaluation and process reduction. Section 4 describes types and a type assignment system. Section 5 introduces algorithmic type checking. Section 6 reviews related work. Section 7 concludes the paper and points directions for future work. We leave for the appendix extra examples (Appendix A), extra definitions (Appendix B), as well as the proofs for results (Appendix C).

2 Motivation

To motivate our approach, we present examples of real-world use cases.

Distributed computing. In distributed systems, the computational power of several networked machines, which need not be in the same geographical location or network, is harnessed towards the completion of a shared goal. To distribute tasks between nodes, messages, containing code to be executed, are dispatched across the network. Commercial implementations such as Apache Spark [56] and Hadoop [54], follow the *MapReduce* programming model [21], inspired by functional programming. A related concept is that of *volunteer computing*, where volunteers (e.g., personal laptops) donate their idle processing power to solve large-scale problems. Boinc [1], targeting scientific projects with intensive computational needs, is one such example: volunteers subscribe to receive tasks as code, compute it and then return the result.

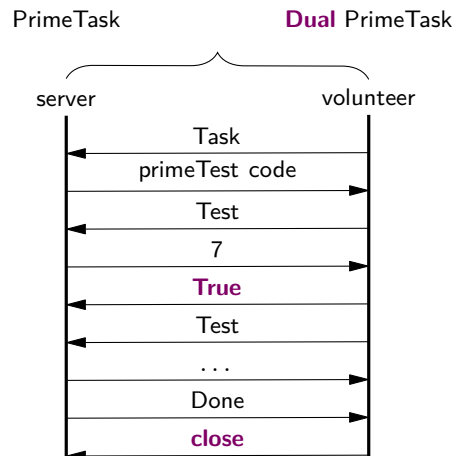
Consider the following protocol, structuring the communication between the central server and a volunteer, in the context of a prime search computational effort, such as PrimeGrid [5].

```
type PrimeTask = &{ Task: ![Int ⊢ Bool].PrimeTask,
                    Test: !Int.?Bool.PrimeTask,
                    Done: Wait }
```

The server offers three alternatives to subscribed volunteers. The first alternative is **Task**, which will provide the volunteer with a task to be performed. The task itself is specified as code, which we denote as the *box type* $[Int ⊢ Bool]$, i.e., code of type **Bool**, parameterized by a variable of type **Int**. The task is then sent to the volunteer ($![Int ⊢ Bool]$), and then the protocol continues from the beginning (**PrimeTask**). Afterwards, the volunteer can donate its idle computational resources by selecting **Test**. The server sends a number ($!Int$), which we'll call the *prime candidate*, to be tested, and then expects a result, i.e., either **True** or **False** in return ($?Bool$), which it stores. The interaction again goes back to the beginning. The server provides a third alternative, **Done**, allowing the volunteer to stop contributing, subsequently waiting for the volunteer to close the connection (**Wait**).

Assuming a `testPrime` function of type $[Int ⊢ Bool]$, i.e., code that accepts the splicing in of a number and returns either **True** or **False**, we write the server as follows:

```
server : [Int] → [Bool] → PrimeTask → [Int]
```



■ **Figure 1** Message sequence chart for the PrimeTask example

```

93 server cs rs (Task s) =
94   let s = send testPrime in server cs rs s
95 server (c:cs) res (Test s) =
96   let s = send c in
97   let (result, s) = receive s in
98   server cs (res ++ [result]) s
99 server _ res (Done s) = let s = wait s in res
100

```

A volunteer, when connected to the server, selects between these three options. By selecting **Task** and then received the code, the volunteer stores it for later use. By selecting **Test**, the volunteer receives the prime candidate and runs the test by splicing the candidate into the code. The result is then sent to the server. The volunteer may repeat this action as many times as he wishes, thus donating more processing time. Finally, when the volunteer no longer wishes to contribute, he can just select **Done** and close the connection.

```

107 volunteer : Dual PrimeTask → Unit
108 volunteer v =
109   let v = select Task v in
110   let (primeTest, v) = receive v in
111   let v = select Test v in
112   let (cand, v) = receive v in
113   let box t = primeTest in send (t[cand]) v in
114   let v = select Test v in
115   let (cand, v) = receive v in
116   let box t = primeTest in send (t[cand]) v in
117   ...
118   let v = select Done v in close v
119
120

```

When running our main program, the interaction unfolds as depicted in Figure 1.

```

121 main : [Bool]
122 main = let s = forkWith (λ_. volunteer) in — s : Dual PrimeTask
123 server [7..] s
124
125
126

```

However, during the course of the interaction, the primality test algorithm can be improved. Hence, the volunteer can regularly poll for updated code, by selecting **Task** again.

23:4 Session-typed Staged Metaprogramming

```

129
130 volunteer : Dual PrimeTask → Unit
131 volunteer v =
132   ...
133   let v = select Task v in
134   let (primeTest, v) = receive v in
135   ...
136

```

In this example, session types are used to govern the communication aspect of task assignment, whereas staged metaprogramming models sending code via messages.

Computational offloading consists in transferring computationally intensive tasks from resource-constrained devices (drones, IoT devices, smartphones) to more powerful remote servers. The main advantage lies in circumventing the shortcomings of these devices, particularly low storage capacity, low processing power and low battery life, enabling greater efficiency and decreased energy consumption on the device, as witnessed by Clonecloud [17]. The advantages of this concept become highly relevant in the context of high throughput networks, since the overhead costs regarding the communication aspect are lessened. Our contribution relates closely with the concept of computation offloading, where session types broker the offloading of tasks, described as code via box types, as in the type below.

```

148
149 type ComputeServer = &{ Compute:?[ ⊢ Int ].! Int . ComputeServer
150                       , Done:Wait }
151

```

Code generation service. Template metaprogramming, as proposed by Sheard and Peyton Jones [42] for Haskell, allows one to write metaprograms that can produce other programs, such as those capable of manipulating arbitrary-sized data structures. For example, in Haskell, it's possible to write metaprograms such as: *fstN*, which produces functions to extract the first element of an arbitrary-sized tuple, or *zipN*, which produces n-ary zip functions. Consider the protocol for a server offering code generating services:

```

158
159 type CodeGenerator = &{
160   Fst3:![ ⊢ (a,b,c) → a ]. CodeGenerator ,
161   Fst4:![ ⊢ (a,b,c,d) → a ]. CodeGenerator ,
162   ...
163   Zip3:![ ⊢ [a] → [b] → [c] → [(a,b,c)] ]. CodeGenerator
164   ...
165   Done:Close }
166

```

The server provides clients with functions to manipulate data structures of arbitrary size. To satisfy a request, the server evaluates the specified template metaprogram, splicing in the size of the data structure, and sending the result to the client. By only storing the function templates, and deriving the instances, i.e., by running the metaprograms, the server's storage requirements remain low as the service is scaled with more function instances it offers. While this example is still not feasible, it serves to illustrate the advantages of our contribution.

Other use cases. Multi-tier programming languages, such as Hop.js [41], Ocsigen/Eliom [13], Meteor.js [3] and Unison [8], allow a single program to describe the functionality of the different tiers of an application, e.g., client, server and database tier, in one unifying language. Session types allow to structure and verify client-server interactions, while providing type-safety properties compatible with multi-tier programming. ScalaLoci [7] and Ur/Web [9] are languages who relate to multi-staged programming, but lack explicit communication models. These would benefit from session types, e.g., to ensure correctness across computation stages.

HTMX [2], Next.js [4] and React Server Components [6] are web frameworks featuring pre-rendering strategies, where the server pre-renders parts of the UI and sends them to the

Constant	c	$::= \text{close} \mid \text{wait} \mid \text{send} \mid \text{receive} \mid \text{select } l \mid \text{new} \mid \text{fork} \mid \text{fix} \mid \text{unit}$
Contextual value	ρ, σ	$::= \bar{x}.M$
Value	v, u	$::= c \mid x[\varepsilon] \mid \lambda x.M \mid (v, v) \mid \text{box } \sigma \mid \text{send } v$
Term	M, N	$::= v \mid x[\bar{\sigma}] \mid M M \mid (M, M) \mid \text{let } (x, x) = M \text{ in } M \mid$ $\text{let box } x = M \text{ in } M \mid \text{match } M \text{ with } \{l \rightarrow M_l\}^{l \in L}$
Process	P, Q	$::= \langle M \rangle \mid P \mid Q \mid (\nu x x)P$
Evaluation context	E	$::= [] \mid E M \mid v E \mid (E, M) \mid (v, E) \mid \text{let } (x, x) = E \text{ in } M \mid$ $\text{match } E \text{ with } \{l \rightarrow M_l\}^{l \in L} \mid \text{let box } x = E \text{ in } M$

■ **Figure 2** Terms and processes

182 client, while also allowing server components, i.e., code that only runs in servers. Session
 183 types allow to verify the correctness of UI update flows, while staged metaprogramming
 184 helps establish the theoretical basis behind the pre-rendering strategies.

185 3 Terms and processes

186 We build our language on the syntactic categories of variables, x, y, z and of labels l . We
 187 write \bar{X} for a sequence of objects $X_1 \cdots X_n$ with $n \geq 0$. The empty sequence (when $n = 0$) is
 188 denoted by ε . The syntax of terms is in Figure 2. It includes channel related primitives: **close**
 189 for closing a channel, **wait** for waiting for a channel to be closed, **send** to write a given value
 190 on a channel, **receive** to read a value from a channel, **select** l for sending label l on a given
 191 channel and **new** for creating a channel. **fork** creates a new thread, **fix** is the call-by-value
 192 fixed point combinator. **unit** is the only value of its type, and can be thought as a placeholder
 193 for other primitive values, such as integer or boolean values found in examples.

194 Further terms include constants, modal variables (playing the dual role of term variables
 195 and of channel endpoints), lambda abstraction and application, pair introduction and
 196 elimination, box introduction and elimination, a **match** term to branch according to the label
 197 l selected. The examples in Section 2 use **forkWith**, a convenient function that puts together
 198 channel creation and process spawning:

```
199 forkWith f = let (x, y) = new () in fork (lambda_. f y) ; x
200
201
```

202 To objects of the form $\bar{x}.M$ we call *contextual values*. They denote code fragments M
 203 parameterized by the variables in sequence \bar{x} . A *contextual term variable* $x[\bar{\sigma}]$ applies the
 204 code fragment described by x to contextual values $\bar{\sigma}$. The **box** σ term turns a code fragment
 205 denoted by a contextual value σ into a term; the **let box** $x = M$ in N term eliminates the
 206 box in M and binds x to it, to be used in N . The **let-box** term allows code to be spliced into
 207 another code fragment (N), by eliminating the box in M and binding the result. When \bar{x} is
 208 the empty sequence we sometimes write M in place of the contextual value $\varepsilon.M$. Similarly,
 209 when $\bar{\sigma}$ is the empty sequence we sometimes write x instead of the contextual term variable
 210 $x[\varepsilon]$. Channel endpoints are always expressed by $x[\varepsilon]$, which is included in values.

211 The *bindings* in the language are the following. Variable x is bound in M in terms $\lambda x.M$
 212 and **let box** $x = N$ in M ; variables x and y are bound in M in term **let** $(x, y) = N$ in M ; the
 213 variables in \bar{x} are bound in M in contextual value $\bar{x}.M$. The set of bound and free variables

R-BETA	R-SPLIT	
$(\lambda x.M) v \rightarrow \{\varepsilon.v/x\}M$	$\text{let } (x, y) = (u, v) \text{ in } M \rightarrow \{\varepsilon.u/x\}\{\varepsilon.v/y\}M$	
R-LETBOX	R-FIX	R-CTX
$\text{let box } x = \text{box } \sigma \text{ in } M \rightarrow \{\sigma/x\}M$	$\text{fix } v \rightarrow v (\lambda x.(\text{fix } v) x)$	$\frac{M \rightarrow N}{E[M] \rightarrow E[N]}$

■ **Figure 3** Term evaluation $M \rightarrow M$

in terms (free M) are defined accordingly. We follow the variable convention whereby all bound variables are chosen to be different from the free variables, in all contexts [14, 37].

Substitution is defined accordingly, from the bindings in the language, while taking advantage of the variable convention. Substitution is rather conventional except for the fact that the term language includes applied modal variables $x[\bar{\sigma}]$ rather than ordinary variables x . We denote by $\{\sigma/x\}M$ the term obtained by replacing the free occurrences of variable x by the contextual value σ in term M , and similarly for $\{\sigma/x\}\rho$. We detail the novelties.

$$\begin{aligned} \{\bar{z}.M/x\}(x[\bar{\rho}]) &= \{\{\bar{z}.M/x\}\bar{\rho}/\bar{z}\}M & \{\sigma/x\}(y[\bar{\rho}]) &= y[\{\sigma/x\}\bar{\rho}] \quad \text{if } x \neq y \\ \{\sigma/x\}(\bar{y}.M) &= \bar{y}.\{\sigma/x\}M \end{aligned}$$

Multiple substitution $\{\bar{\rho}/\bar{z}\}M$ is defined only when $\bar{\rho}$ and \bar{z} are sequences of the same length. The variable convention ensures that the variables in \bar{z} are pairwise distinct and not free outside M , hence the substitution of the various variables in \bar{z} can be performed in sequence. Substitution on applied variables, $\{\bar{z}.M/x\}(x[\bar{\rho}])$, triggers further substitution for \bar{z} in M . For example, $\{z_1, z_2.\text{send } z_1 z_2/x\}(x[y, 42]) = \{y/z_1\}\{42/z_2\}(\text{send } z_1 z_2) = \text{send } y 42$. Substitution on the remaining term constructors is a homomorphism; for example $\{\sigma/x\}(\text{box } \rho) = \text{box } (\{\sigma/x\}\rho)$ and $\{\sigma/x\}(\text{let box } y = M \text{ in } N) = \text{let box } y = \{\sigma/x\}M \text{ in } \{\sigma/x\}N$. Since substitution for an applied variable triggers another, well-definedness is not trivial. We show that substitution is well defined for typable terms (provided that a few additional side conditions are met) in Appendix C.

If terms provide for the sequential part of the language, *processes* deal with concurrency and are used as a runtime only. Programmers write terms M that are run on an initial thread $\langle M \rangle$ that may eventually fork new threads and create new communication channels. Process $P \mid Q$ denotes the parallel composition of two processes. Process $(\nu xy)P$ introduces in process P a communication channel described by its two endpoints x and y . Our development is quite standard in the literature [23]; the details are in Appendix B.

Evaluation on terms is given by the relation $M \rightarrow N$, defined by the rules in Figure 3. It includes function application (R-BETA), the elimination of a pair of values (u, v) and their binding to variables x and y in M (R-SPLIT). Notice that substitution is defined for contextual values only, hence we lift value u to $\varepsilon.u$, and similarly for v . Rule R-LETBOX unboxes a boxed code fragment σ , binds it to x to be used in M . Rule R-FIX unfolds the call by value fixed point constructor fix [31]. Finally, rule R-CTX evaluates terms under evaluation contexts, the syntax of which is in Figure 2. These are the standard call-by-value evaluation contexts. Notice we do not allow reduction inside boxes just as we do not allow under λ -abs.

Multiplicity	$m ::= 1 \mid \omega$
Type	$T, U ::= \text{Unit} \mid T \rightarrow_m T \mid T \times T \mid \Box \tau^{n+1} \mid S$
Session type	$R, S ::= \text{Close} \mid \text{Wait} \mid !T.S \mid ?T.S \mid \oplus \{l : S_l\}^{l \in L} \mid \& \{l : S_l\}^{l \in L} \mid a \mid \mu a.S$
Contextual type τ^n	$::= \bar{\tau} \vdash^n T$
Typing context	$\Gamma, \Delta ::= \varepsilon \mid \Gamma, x : \tau$

■ **Figure 4** Types

$$\begin{array}{c}
\text{Close} \perp \text{Wait} \quad \text{Wait} \perp \text{Close} \quad \frac{S \perp R}{!T.S \perp ?T.R} \quad \frac{S \perp R}{?T.S \perp !T.R} \quad \frac{R \perp S\{\mu a.S/a\}}{R \perp \mu a.S} \\
\\
\frac{R\{\mu a.R/a\} \perp S}{\mu a.R \perp S} \quad \frac{S_l \perp R_l \quad \forall l \in L}{\oplus \{l : S_l\}^{l \in L} \perp \& \{l : R_l\}^{l \in L}} \quad \frac{S_l \perp R_l \quad \forall l \in L}{\& \{l : S_l\}^{l \in L} \perp \oplus \{l : R_l\}^{l \in L}}
\end{array}$$

■ **Figure 5** Type duality (coinductive) $\boxed{S \perp S}$

4 Types

248

249 We build the language of types on the syntactic category of *type references*, a, b . We further
 250 use letters k, n to denote natural numbers, and may call them *levels*. The syntax of *types*
 251 is in Figure 4. *Multiplicities* m , include 1 denoting a *linear* resource, i.e. that must be
 252 used exactly once, and ω describing an *unrestricted* resource, i.e. that may be used an
 253 unbounded number of times, including zero. Type **Unit** (for constant **unit**) can be thought as
 254 a placeholder for other base types, such as those for integer or boolean values. Further types
 255 include those for linear and unrestricted functions, $T \rightarrow_1 U$ and $T \rightarrow_\omega U$, that for linear
 256 pairs, $T \times U$ (for simplicity we do not consider unrestricted pairs), and for box types $\Box \tau$,
 257 that is, code fragments characterised by contextual types τ . The grammar for contextual
 258 types is indexed by levels; when levels are not important we omit them and use τ .

259 *Session types* describe communication channel endpoints. Types **Close** and **Wait** are for
 260 channels ready, or waiting, to be closed, respectively. Type $!T.S$ and $?T.S$ are for channels
 261 that send, or receive, values of type T and continue as S , respectively. Type $\oplus \{l : S_l\}^{l \in L}$
 262 is for channels that may select one branch $k \in L$ and proceed as S_k (internal choice). Its dual,
 263 $\& \{l : S_l\}^{l \in L}$, is for channels that offer a menu of labelled choices (external choice). Recursive
 264 session types are built from type references a and recursion $\mu a.S$. We take an *equi-recursive*
 265 approach to types, not distinguishing a recursive type from its unfolding [10, 37].

266 The notion of *duality* in session types is captured by relation $R \perp S$, in Figure 6. Types
 267 **Wait** and **Close**; input and output; and internal and external choice are dual to each other.
 268 Recursive types are unfolded. The definition is coinductive, so that we accept possible infinite
 269 derivations. We refer to Gay et al. for details [22].

Box types are of the form $\bar{\tau} \vdash^n T$, where $\bar{\tau}$ denotes a possible empty sequence of contextual
 types. Such a type represents a code fragment M of type T , parameterised by a sequence
 of variables \bar{x} of contextual types $\bar{\tau}$, called contexts [27, 32]. Ordinary (modal) variables
 are typed at level 0. The context of an ordinary variable is always empty, which justifies
 writing T for the contextual type $\varepsilon \vdash^0 T$. Furthermore, in a type $\bar{\tau} \vdash^{n+1} T$, we require all

$$\frac{T \omega}{(\varepsilon \vdash^0 T) \omega} \quad (\bar{\tau} \vdash^{n+1} T) \omega \quad \text{Unit } \omega \quad (T \rightarrow_\omega U) \omega \quad (\Box \tau) \omega$$

■ **Figure 6** Unrestricted types $\boxed{T \omega}$ $\boxed{\tau \omega}$

contextual types in $\bar{\tau}$ to be of levels smaller than or equal to n . The types in $\bar{\tau}$ characterise the parameters \bar{x} of the contextual value $\bar{x}.M$ and thus must be typed at levels smaller than that of the value itself. We capture these intuitions with a family of type formation predicates, $T \text{ ctype}^n$, defined by the rules below.

$$\begin{array}{c} (\varepsilon \vdash^0 T) \text{ ctype}^0 \\ \hline \frac{\tau \text{ ctype}^n}{(\bar{\tau} \vdash^{n+1} T) \text{ ctype}^{n+1}} \quad \frac{\tau \text{ ctype}^n}{\tau \text{ ctype}^{n+1}} \end{array}$$

Henceforth we assume that all contextual types τ are well formed, that is, that $\tau \text{ ctype}^n$ holds for some n . For example, object $(\bar{\tau} \vdash^1 U) \vdash^1 T$ cannot be deemed a well formed type. This formulation adheres to the development of Møebius and its kinding system [27].

Finally, the *box type* $\Box \tau^{n+1}$ represents a code fragment of contextual type τ . Boxed code fragments are typed at levels starting at 1, hence the syntax $\bar{\tau} \vdash^{n+1} T$ (emphasis on $n+1$).

Predicates on types and on typing contexts Types denote resources that may be used exactly once or else an unbounded number of times. The ω predicate is true of types and contextual types whose values can be used an unbounded number of times. The definition is in Figure 6. Contextual types of level 1 or above are all unrestricted, whereas those of level 0 are unrestricted only when the type T of the code is unrestricted. Level 0 represents ordinary code (of type T) and hence $\varepsilon \vdash^0 T$ is unrestricted if T is. On the other hand, levels higher than 0 represent boxed code fragments that can be duplicated or discarded at will, hence they are all of an unrestricted nature. For types, **Unit**, unrestricted arrows $T \rightarrow_\omega U$ and box types $\Box \tau$ are the only unrestricted types. The rules in Figure 6 are algorithmic. The ω predicate is thus decidable, what allows us to talk of its negation. A contextual type τ that is not unrestricted is called *linear*. The corresponding predicate is denoted $\tau \text{ l}$. Two further predicates describe upper and lower bounds to the levels of contextual types. The predicate $< k$ is true of contextual types $\bar{\tau} \vdash^n T$ such that $n < k$. The predicate $\geq k$ is true of contextual types $\bar{\tau} \vdash^n T$ such that $n \geq k$.

Typing contexts Γ, Δ bind variables x to contextual types τ . We assume no variable is bound twice in the same context and take typing contexts up to the *exchange* of individual entries. The remaining two substructural rules—contraction and weakening—are handled by context split (described below) and by allowing contexts with unrestricted types in axioms. The four predicates are then lifted to typing contexts in the standard way. For example $\Gamma \omega$ is true of contexts $x_1: \tau_1, \dots, x_k: \tau_k$ such that $\tau_1 \omega, \dots, \tau_k \omega$. Intuitively, $\Gamma < n$ is true of a context Γ suitable for the *local variables* in a code fragment of level n , whereas $\Gamma \geq n$ is true of a context suitable for the *outer variables*. Notice that there is no context Γ for which $\Gamma < 0$ holds, with the exception of the empty context ε ; $\Gamma \geq 0$ is a tautology.

We use notation τ^ω to denote a contextual type τ for which $\tau \omega$ holds, and similarly for τ^1 . For contexts, we use notation Γ^ω to denote a context Γ for which $\Gamma \omega$ holds, and similarly for $\Gamma^{<n}$ and $\Gamma^{\geq n}$. Let $\mathcal{U}(\Gamma)$ denote the unrestricted part of Γ , that is, the typing context containing exactly those bindings $x: \tau^\omega$ in Γ . We define $\mathcal{L}(\Gamma)$ in a corresponding manner.

Typing terms The *types for constants* are described by the type schemes in Figure 7. Those for session types are taken mostly from Gay and Vasconcelos [23]. Constants close

$$\begin{aligned}
\text{typeof}(\text{close}) &= \text{Close} \rightarrow_{\omega} \text{Unit} & \text{typeof}(\text{wait}) &= \text{Wait} \rightarrow_{\omega} \text{Unit} \\
\text{typeof}(\text{send}) &= T \rightarrow_{\omega} !T.S \rightarrow_1 S & \text{typeof}(\text{receive}) &= ?T.S \rightarrow_{\omega} T \times S \\
\text{typeof}(\text{select } k) &= \oplus\{l : S_l\}^{l \in L} \rightarrow_{\omega} S_k, \text{ with } k \in L \\
\text{typeof}(\text{new}) &= \text{Unit} \rightarrow_{\omega} R \times S, \text{ with } R \perp S & \text{typeof}(\text{fork}) &= (\text{Unit} \rightarrow_1 \text{Unit}) \rightarrow_{\omega} \text{Unit} \\
\text{typeof}(\text{fix}) &= ((T \rightarrow_{\omega} T) \rightarrow_{\omega} (T \rightarrow_{\omega} T)) \rightarrow_{\omega} (T \rightarrow_{\omega} T) & \text{typeof}(\text{unit}) &= \text{Unit}
\end{aligned}$$

■ **Figure 7** Type schemes for constants $\boxed{\text{typeof}(c) = T}$

$$\begin{aligned}
\varepsilon &= \varepsilon \circ \varepsilon & \frac{\Gamma = \Gamma_1 \circ \Gamma_2}{\Gamma, x : \tau^1 = (\Gamma_1, x : \tau^1) \circ \Gamma_2} & \frac{\Gamma = \Gamma_1 \circ \Gamma_2}{\Gamma, x : \tau^1 = \Gamma_1 \circ (\Gamma_2, x : \tau^1)} \\
& & \frac{\Gamma = \Gamma_1 \circ \Gamma_2}{\Gamma, x : \tau^{\omega} = (\Gamma_1, x : \tau^{\omega}) \circ (\Gamma_2, x : \tau^{\omega})}
\end{aligned}$$

■ **Figure 8** Context split $\boxed{\Gamma = \Gamma \circ \Gamma}$

and **wait** close different ends of a channel and return **Unit**. Constant **send** accepts a value and a channel on which to send the value and returns the continuation channel; the second arrow is linear for T may denote a linear value. Constant **receive** accepts a channel and returns a pair composed of the value read and the continuation channel. Constant **select** k accepts an external choice type selects branch k , returning the appropriate continuation. Constant **new** denotes a suspended computation that, when invoked (with **unit**), returns a pair of channel endpoints of dual types. Constant **fork** receives a suspended computation, spawns a new thread to run the computation and returns **Unit**. Finally the type of **fix** is that of the call-by-value lambda calculus fixed-point combinator; all arrows are unrestricted for the argument function may be used an unbounded number of times (in each recursive call), including zero (in the base case), and **fix** itself can be used an unbounded number of times.

There will be times when the type system must merge typing contexts coming from different subderivations into a single context. *Context split* (read bottom-up) does the job. The rules are in Figure 8 and taken verbatim from Walker [53]: linear types go left or right (but not in both directions); unrestricted types are copied into both contexts.

Typing judgements are of the form $\Gamma \vdash M : T$, stating term M has type T under typing context Γ . The rules for the judgements, which we describe bellow, are in Figure 9.

Contextual terms are closures of the form $\bar{x}.M$, closing term M over a sequence of variables \bar{x} . Such a contextual term is given a contextual type $\bar{\tau} \vdash^n T$ when T is of type M under the hypothesis that the variables in \bar{x} are of the types in $\bar{\tau}$ (rule T-Ctx). Intuitively, the free variables of M are split in two groups: local and outer. The local variables, \bar{x} , are distinguished in the contextual term $\bar{x}.M$ and typed under context $\bar{x} : \bar{\tau}^{<n}$ where n is an upper bound of the levels k_i assigned to each local variable x_i in \bar{x} . The level of each local variable is arbitrary, as long as it is smaller than n . Outer variables are typed under context $\Gamma^{\geq n}$. Natural number n thus denotes the level at which the code fragment $\bar{x}.M$ is typed. In general, a contextual term can be typed at different levels. All resources (variables) in the context in the conclusion, as well as resources \bar{x} are consumed in the derivation of M .

$$\begin{array}{c}
\text{T-CONST} \\
\frac{}{\Gamma^\omega \vdash c : \text{typeof}(c)}
\end{array}
\quad
\begin{array}{c}
\text{T-VAR} \\
\frac{\bar{\Gamma} \vdash \bar{\sigma} : \bar{\tau}}{(\Delta^\omega, x : (\bar{\tau} \vdash^n T)) \circ \bar{\Gamma} \vdash x[\bar{\sigma}] : T}
\end{array}
\quad
\begin{array}{c}
\text{T-LINARRI} \\
\frac{\Gamma, x : (\varepsilon \vdash^0 T) \vdash M : U}{\Gamma \vdash \lambda x. M : T \rightarrow_1 U}
\end{array}$$

$$\begin{array}{c}
\text{T-UNARRI} \\
\frac{\Gamma^\omega, x : (\varepsilon \vdash^0 T) \vdash M : U}{\Gamma^\omega \vdash \lambda x. M : T \rightarrow_\omega U}
\end{array}
\quad
\begin{array}{c}
\text{T-ARRE} \\
\frac{\Gamma \vdash M : T \rightarrow_m U \quad \Delta \vdash N : T}{\Gamma \circ \Delta \vdash M N : U}
\end{array}
\quad
\begin{array}{c}
\text{T-PAIRI} \\
\frac{\Gamma \vdash M : T \quad \Delta \vdash N : U}{\Gamma \circ \Delta \vdash (M, N) : T \times U}
\end{array}$$

$$\begin{array}{c}
\text{T-PAIRE} \\
\frac{\Gamma \vdash M : T \times U \quad \Delta, x : (\varepsilon \vdash^0 T), y : (\varepsilon \vdash^0 U) \vdash N : V}{\Gamma \circ \Delta \vdash \text{let } (x, y) = M \text{ in } N : V}
\end{array}
\quad
\begin{array}{c}
\text{T-CTX} \\
\frac{\Gamma^{\geq n}, \bar{x} : \bar{\tau}^{<n} \vdash M : T}{\Gamma^{\geq n} \vdash \bar{x}. M : (\bar{\tau} \vdash^n T)}
\end{array}
\quad
\begin{array}{c}
\text{T-BOXI} \\
\frac{\Gamma^\omega \vdash \sigma : \tau}{\Gamma^\omega, \Delta^\omega \vdash \text{box } \sigma : \Box \tau}
\end{array}$$

$$\begin{array}{c}
\text{T-BOXE} \\
\frac{\Gamma \vdash M : \Box \tau \quad \Delta, x : \tau \vdash N : T}{\Gamma \circ \Delta \vdash \text{let box } x = M \text{ in } N : T}
\end{array}
\quad
\begin{array}{c}
\text{T-MATCH} \\
\frac{\Gamma \vdash M : \&\{l : S_l\}^{l \in L} \quad \Delta \vdash N_l : S_l \rightarrow_1 T \quad (\forall l \in L)}{\Gamma \circ \Delta \vdash \text{match } M \text{ with } \{l : N_l\}^{l \in L} : T}
\end{array}$$

■ **Figure 9** Term formation $\boxed{\Gamma \vdash M : T}$

331 *Modal variables* are of the form $x[\sigma_1 \cdots \sigma_k]$, denoting a contextual term applied to
 332 contextual values $\sigma_1 \cdots \sigma_k$. To type each σ_i we need a separate context Γ_i . The type of
 333 x must be a contextual type of the form $\tau_1 \cdots \tau_k \vdash^n T$ and each contextual term σ_i must
 334 be of type τ_i . Since x may occur free in any of σ_i , we use context split to allow the entry
 335 $x : (\tau_1 \cdots \tau_k \vdash^n T)$ to occur in each Γ_i . In this case, the contextual type must be unrestricted.
 336 The lengths of all sequences— $\bar{\Gamma}$, $\bar{\sigma}$ and $\bar{\tau}$ —must coincide. When the sequences are empty,
 337 the rule becomes the conventional axiom for variables. In fact rule T-VAR is the natural
 338 generalization of the axiom in Walker’s linear λ -calculus [53], obtained by writing type $\varepsilon \vdash^n T$
 339 in the context as T , and writing modal variable $x[\varepsilon]$ as x , thus $\Delta^\omega, x : (\varepsilon \vdash^n T) \vdash x[\varepsilon] : T$
 340 abbreviates to $\Delta^\omega, x : T \vdash x : T$. The axiom justifies the presence of the unrestricted context
 341 Δ in the conclusion of T-VAR.

We briefly pause the presentation of the type system to address substitution. The *substitution principle* in the presence of linear typing is tricky because substitution can duplicate or discard terms [23, 53]. In addition, we have to take levels of types into account. The resulting substitution principle can be stated as follows.

$$\begin{array}{c}
\text{SUBS} \\
\frac{\Gamma \vdash \sigma : \tau^n \quad \Delta, x : \tau^n \vdash M : T \quad (\tau \text{ 1 or } \Gamma \omega) \quad (\Gamma \geq n)}{\Gamma \circ \Delta \vdash \{\sigma/x\}M : T}
\end{array}$$

342 We write $\Gamma \circ \Delta$ to denote the context Θ such that $\Theta = \Gamma \circ \Delta$. When we do so, we assume
 343 that the context split operation is defined. The first side condition $\tau^n \text{ 1 or } \Gamma \omega$ prevents free
 344 variables of linear types in σ from being duplicated or discarded by substitution $\{\sigma/x\}M$: if
 345 τ^n is linear, then no duplication or discarding occurs and, if $\Gamma \omega$, then free variables are safe
 346 to be duplicated or discarded. The second side condition, $\Gamma \geq n$, means that the levels of
 347 free variables in σ have to be equal to or greater than the level of σ .

If $n = 0$, i.e. x is an ordinary variable, the condition $\Gamma \geq n$ trivially holds. Although the other side condition does not hold in general, we can show that it holds for values (that is, if

$\sigma = \varepsilon.v$ for some v). Thus, we can obtain the following (call-by-value) substitution property:

$$\frac{\Gamma \vdash \varepsilon.v : (\varepsilon \vdash^0 U) \quad \Delta, x : (\varepsilon \vdash^0 U) \vdash M : T}{\Gamma \circ \Delta \vdash \{\varepsilon.v/x\}M : T} \quad \text{abbreviated to} \quad \frac{\Gamma \vdash v : U \quad \Delta, x : U \vdash M : T}{\Gamma \circ \Delta \vdash \{v/x\}M : T}$$

Returning to the type system, we address the conventional typing rules for λ -abstraction and application in the linear lambda calculus. The rules are those of Walker [53] with the necessary adaptation of the introduction rules to account for contexts featuring contextual types. There are two introduction rules, one for the linear, the other for the unrestricted arrow. In either case, the λ -bound variable x is local to M , hence of level 0. We record this fact by assigning x a type of the form $\varepsilon \vdash^0 T$. For the unrestricted arrow we require an unrestricted context, given that the context contains entries for the free variables in term $\lambda x.M$ and this, being unrestricted, may be duplicated or discarded.

The rules enjoy local soundness and local completeness [36]. We discuss these in turn. For *local soundness* we have:

$$\frac{\frac{\Gamma, x : (\varepsilon \vdash^0 T) \vdash M : U}{\Gamma \vdash \lambda x.M : T \rightarrow_\omega U} \text{T-LINARRI} \quad \Delta \vdash v : T}{\Gamma \circ \Delta \vdash (\lambda x.M) v : U} \text{T-ARRE} \Rightarrow$$

$$\frac{\frac{\Delta \vdash v : T}{\Delta \vdash \varepsilon.v : (\varepsilon \vdash^0 T)} \text{T-CTX} \quad \Gamma, x : (\varepsilon \vdash^0 T) \vdash M : U}{\Gamma \circ \Delta \vdash \{\varepsilon.v/x\}M : U} \text{SUBS}$$

The case for \rightarrow_ω is similar and can be obtained by adding the ω restriction to context Γ .

For *local completeness*, assume that x is not free in M and let τ be the contextual type $\varepsilon \vdash^0 T$. We must distinguish four cases, according to when Γ and/or T are unrestricted or linear. We start with the case when both Γ and T are linear. Then, τ is linear and $\Gamma, x : \tau = \Gamma \circ x : \tau$, which justifies the instance of rule T-ARRE below.

$$\frac{\Gamma \vdash M : T \rightarrow_1 U \quad \frac{}{x : \tau \vdash x[\varepsilon] : T} \text{T-VAR}}{\Gamma, x : \tau \vdash M x[\varepsilon] : U} \text{T-ARRE} \Rightarrow \frac{}{\Gamma \vdash M : T \rightarrow_1 U} \text{T-LINARRI}$$

When T is unrestricted, we have $\Gamma, x : \tau = (\Gamma, x : \tau) \circ x : \tau$ and we must use weakening (Lemma 8) to add $x : \tau$ to the context of M . We show the case when Γ is also unrestricted.

$$\frac{\Gamma^\omega \vdash M : T \rightarrow_\omega U \quad \frac{}{x : \tau \vdash x[\varepsilon] : T} \text{T-VAR}}{\Gamma^\omega, x : \tau \vdash M : T \rightarrow_\omega U} \text{WEAK} \quad \frac{}{\Gamma^\omega, x : \tau \vdash M x[\varepsilon] : U} \text{T-ARRE} \Rightarrow \frac{}{\Gamma^\omega \vdash \lambda x.(M x[\varepsilon]) : T \rightarrow_\omega U} \text{T-UNARRI}$$

The two remaining cases are similar.

The *box introduction and elimination* rules are as follows. In rule T-BoxI, the type of the box is the type $\Box\tau$ if the contextual term σ has contextual type τ . The context for the box elimination rule, T-BoxE, is split into two: one part (Γ) to type M , the other (Δ) to type N .

Term M must denote a boxed code fragment, hence the type of M must be $\Box\tau$. Term N is typed under context Δ extended with an entry for x .

Linear type systems with a context split operator perform all the required weakening at the leaves of derivations. Hence all axioms expect a Γ^ω context to discard unrestricted variables. This is the case of rules T-CONST and T-VAR (with $\bar{\sigma}$ empty), and is all we need in most cases. But, when typing a box, we need weakening for a different reason: to discard unrestricted resources of small level (of levels smaller than that necessary to type the code fragment inside the box). Context Δ^ω plays this role; we expect this context to contain all unrestricted entries of levels below n . Linear resources cannot be discarded in any case.

Rule T-BoxI is perfectly aligned with preceding work [20, 27, 33], where level control is checked at box introduction. A derived rule, composed of T-CTX followed by T-BoxI, coincides with the box introduction rule of Mœbius [27]:

$$\frac{\Gamma^\omega, \bar{x} : \bar{\tau}^{<n} \vdash M : T \quad (\Gamma^\omega \geq n)}{\Gamma^\omega, \Delta^\omega \vdash \text{box } (\bar{x}.M) : \Box(\bar{\tau} \vdash^n T)}$$

For *local soundness* we have:

$$\frac{\frac{\Gamma^\omega \vdash \sigma : \tau^n \quad (\Gamma^\omega \geq n)}{\Gamma^\omega, \Delta^\omega \vdash \text{box } \sigma : \Box\tau^n} \text{ T-BoxI} \quad \Theta, x : \tau^n \vdash M : T}{(\Gamma^\omega, \Delta^\omega) \circ \Theta \vdash \text{let box } x = \text{box } \sigma \text{ in } M : T} \text{ T-BoxE} \Rightarrow$$

$$\frac{\frac{\Gamma^\omega \vdash \sigma : \tau^n}{\Gamma^\omega, \Delta^\omega \vdash \sigma : \tau^n} \text{ WEAK} \quad \Theta, x : \tau^n \vdash M : T}{(\Gamma^\omega, \Delta^\omega) \circ \Theta \vdash \{\sigma/x\}M : T} \text{ SUBS}$$

For *local completeness* we first define eta expansion for contextual types, written as $\eta(x, \tau)$ by the following rules: $\eta(x, \varepsilon \vdash^0 T) = \varepsilon.x[\varepsilon]$ and $\eta(x, \bar{\tau} \vdash^{n+1} T) = \bar{y}.x[\eta(y, \tau)]$. Eta expansion $\eta(x, \tau)$ is defined at any contextual type τ because the level of τ strictly decreases at each step. We can confirm that the following lemma holds.

► **Lemma 1.** $x : \tau \vdash \eta(x, \tau) : \tau$.

Proof. By induction on the level of τ . ◀

We now witness local completeness. Let Δ be the unrestricted part of Γ . Then $\Gamma = \Gamma \circ \Delta$. Let $\bar{\rho}$ be the sequence of contextual types, whose level is lower than n , and τ be the contextual type $\bar{\rho} \vdash^n T$. We have $\bar{x} : \bar{\rho}^{<n}$ and $(u : \tau)^{\geq n}$. Furthermore, we can see the context $(u : \tau)^{\geq n}$ is unrestricted because $n > 0$ is required by the well-formedness condition of $\Box\tau$.

$$\Gamma \vdash M : \Box\tau \Rightarrow$$

$$\frac{\frac{\frac{\frac{\overline{\text{Lemma 1}}}{\bar{x} : \bar{\rho} \vdash \eta(x, \bar{\rho}) : \bar{\rho}}}{\bar{x} : \bar{\rho} \circ u : \tau \vdash u[\eta(x, \bar{\rho})] : T} \text{ T-VAR}}{u : \tau \vdash \bar{x}.u[\eta(x, \bar{\rho})] : \tau} \text{ T-CTX}}{\Gamma \vdash M : \Box\tau \quad \Delta, u : \tau \vdash \text{box } (\bar{x}.u[\eta(x, \bar{\rho})]) : \Box\tau} \text{ T-BoxI} \text{ T-BoxE}$$

$$\Gamma \circ \Delta \vdash \text{let box } u = M \text{ in box } (\bar{x}.u[\eta(x, \bar{\rho})]) : \Box\tau$$

Finally, the rules for pair introduction and elimination are standard and can be found, e.g., in Almeida et al. in Gay and Vasconcelos or in Walker [11, 23, 53].

We show a conventional safety property for the term language based on *preservation* and *progress*—Theorems 2 and 3—following Pierce [37]. In the case of progress we take

into consideration terms that may be (temporarily) stuck waiting for a communication on a channel. The process language does not enjoy progress due to possible occurrences of deadlocks. Safety for the process language is based on *preservation* and *absence of runtime errors*—Theorems 2 and 4—following Honda et al. [25].

- **Theorem 2** (Preservation). 1. If $M \rightarrow N$ and $\Gamma \vdash M : T$, then $\Gamma \vdash N : T$.
 2. If $\Gamma \vdash P$ and $P \rightarrow Q$, then $\Gamma \vdash Q$.

Progress is usually stated for closed terms. Because channel endpoints are also variables, we must take into account terms with free endpoints, that is variables of session types. We say that a context Γ *contains session types only*, and write Γ^S when Γ is of the form $x_1 : \varepsilon \vdash^0 S_1, \dots, x_n : \varepsilon \vdash^0 S_n$.

- **Theorem 3** (Progress for evaluation). Let $\Gamma^S \vdash M : T$. Then,
 1. M is a value, or
 2. $M \rightarrow N$, for some N , or
 3. M is of the form $E[N]$, for some E , with N of one of the following forms: *close* x , *wait* x , *send* $v x$, *receive* x , *select* $l x$, *match* x with $\{l \rightarrow N_l\}^{l \in L}$, *new* v or *fork* v .

In order to address the absence of runtime errors for the process language, we start with a few definitions. The *subject* of a term M is variable x (denoting a channel endpoint) in the following cases and undefined in all other cases.

close x *wait* x *send* $v x$ *receive* x *select* $k x$ *match* x with $\{l \rightarrow N_l\}^{l \in L}$

A process $(\nu xy)(\langle E[M] \rangle \mid \langle F[N] \rangle)$ is a *redex* if

- M is *close* x and N is *wait* y ,
- M is *send* $v x$ and N is *receive* y ,
- M is *select* $l' x$ and N is *match* y with $\{l \rightarrow N_l\}^{l \in L}$ and $l' \in L$.

or conversely for processes $(\nu xy)(\langle F[N] \rangle \mid \langle E[M] \rangle)$. This definition corresponds to the axioms R-CLOSE, R-COM and R-BRANCH in the operational semantics (Figure 13). Saying that P is a redex is equivalent to say that P reduces by one of these rules.

Finally a process P is a *runtime error* if is structural congruent to a process of the form $(\nu x_1 y_1) \cdots (\nu x_n y_n)(P_1 \mid \cdots \mid P_m)$ and there are i, j, k such that the subject of P_i is x_k , the subject of P_j is y_k but $(\nu x_k y_k)(P_i \mid P_j)$ is not a redex.

- **Theorem 4** (Absence of immediate errors). If $\vdash P$ then P is not a runtime error.

5 Algorithmic Type Checking

The typing rules of Figure 9 are not algorithmic. Two problems arise: context split (a non-deterministic procedure) and guessing the types (and levels) of λ -abstractions and contextual values' bound variables. For the former, we refactor the typing rules so they return the unused part of the incoming context [53]. For the latter, we work with an explicitly typed term language: we now write c^τ to resolve the polymorphism of constants, $\lambda_m x : T.M$ to introduce the type of bound variable x , and $\overline{x} : \tau^n.M$ to introduce the contextual types and the level of the contextual value.

We introduce a new type synthesis judgement $\Gamma_{\text{in}} \vdash M_{\text{in}} \Rightarrow T_{\text{out}} \mid \Delta_{\text{out}}$, where context Γ and term M are considered inputs and type T and context Δ are considered outputs. The rules are in Figure 10. In both A-CONST, A-UNARRI and A-BOXI, the input and output contexts are the same, meaning no linear variables may be consumed. Some rules make use of context difference, which we introduce in Definition 5.

$$\begin{array}{c}
\text{A-CONST} \\
\frac{}{\Gamma \vdash c^\tau \Rightarrow \tau \mid \Gamma}
\end{array}
\quad
\begin{array}{c}
\text{A-UNVAR} \\
\frac{\Gamma_1, x: (\bar{\tau} \vdash^n T) \vdash \sigma_1 \Rightarrow \tau_1 \mid \Gamma_2 \quad \Gamma_2 \vdash \sigma_2 \Rightarrow \tau_2 \mid \Gamma_3 \dots \Gamma_k \vdash \sigma_n \Rightarrow \tau_n \mid \Gamma_{k+1}}{\Gamma_1, x: (\bar{\tau} \vdash^n T)^\omega \vdash x[\sigma_1, \dots, \sigma_k] \Rightarrow T \mid \Gamma_{k+1}}
\end{array}$$

$$\begin{array}{c}
\text{A-LINVAR} \\
\frac{\Gamma_1 \vdash \sigma_1 \Rightarrow \tau_1 \mid \Gamma_2 \dots \Gamma_k \vdash \sigma_k \Rightarrow \tau_k \mid \Gamma_{k+1}}{\Gamma_1, x: (\bar{\tau} \vdash^n T) \vdash x[\sigma_1, \dots, \sigma_k] \Rightarrow T \mid \Gamma_{k+1}}
\end{array}
\quad
\begin{array}{c}
\text{A-LINARRI} \\
\frac{\Gamma, x: (\varepsilon \vdash^0 T) \vdash M \Rightarrow U \mid \Delta}{\Gamma \vdash \lambda_1 x: T. M \Rightarrow T \rightarrow_1 U \mid \Delta \div x}
\end{array}$$

$$\begin{array}{c}
\text{A-UNARRI} \\
\frac{\Gamma, x: (\varepsilon \vdash^0 T) \vdash M \Rightarrow U \mid \Delta \quad \Delta \div x = \Gamma}{\Gamma \vdash \lambda_\omega x: T. M \Rightarrow T \rightarrow_\omega U \mid \Gamma}
\end{array}
\quad
\begin{array}{c}
\text{A-ARRE} \\
\frac{\Gamma_1 \vdash M \Rightarrow T \rightarrow_m U \mid \Gamma_2 \quad \Gamma_2 \vdash N \Rightarrow T \mid \Gamma_3}{\Gamma_1 \vdash M N \Rightarrow U \mid \Gamma_3}
\end{array}$$

$$\begin{array}{c}
\text{A-PAIRI} \\
\frac{\Gamma_1 \vdash M \Rightarrow T \mid \Gamma_2 \quad \Gamma_2 \vdash N \Rightarrow U \mid \Gamma_3}{\Gamma_1 \vdash (M, N) \Rightarrow T \times U \mid \Gamma_3}
\end{array}
\quad
\begin{array}{c}
\text{A-PAIRE} \\
\frac{\Gamma_1 \vdash M \Rightarrow T \times U \mid \Gamma_2 \quad \Gamma_2, x: (\varepsilon \vdash^0 T), y: (\varepsilon \vdash^0 U) \vdash N \Rightarrow V \mid \Gamma_3}{\Gamma_1 \vdash \text{let } (x, y) = M \text{ in } N \Rightarrow V \mid \Gamma_3 \div x \div y}
\end{array}$$

$$\begin{array}{c}
\text{A-BOXI} \\
\frac{\Gamma \vdash \sigma \Rightarrow \tau \mid \Gamma}{\Gamma \vdash \text{box } \sigma \Rightarrow \Box \tau \mid \Gamma}
\end{array}
\quad
\begin{array}{c}
\text{A-BOXE} \\
\frac{\Gamma_1 \vdash M \Rightarrow \Box \tau \mid \Gamma_2 \quad \Gamma_2, x: \tau \vdash N \Rightarrow T \mid \Gamma_3}{\Gamma_1 \vdash \text{let box } x = M \text{ in } N \Rightarrow T \mid \Gamma_3 \div x}
\end{array}$$

$$\begin{array}{c}
\text{A-MATCH} \\
\frac{\Gamma_1 \vdash M \Rightarrow \&\{l: S_l\}^{l \in L} \mid \Gamma_2 \quad \Gamma_2 \vdash N_l \Rightarrow S_l \rightarrow_1 T \mid \Gamma_3 \quad (\forall l \in L)}{\Gamma_1 \vdash \text{match } M \text{ with } \{l: N_l\}^{l \in L} \Rightarrow T \mid \Gamma_3}
\end{array}
\quad
\begin{array}{c}
\text{A-CTX} \\
\frac{\Gamma_1 \xrightarrow{n} \Gamma_2^{\geq n}, \Gamma_3^{\leq n} \quad \bar{\tau} < n \quad \Gamma_2, \bar{x}: \bar{\tau} \vdash M \Rightarrow T \mid \Delta}{\Gamma_1 \vdash \bar{x}: \bar{\tau}^n. M \Rightarrow (\bar{\tau} \vdash^n T) \mid \Delta \div \bar{x}, \Gamma_3}
\end{array}$$

■ **Figure 10** Algorithmic type checking $\boxed{\Gamma \vdash M \Rightarrow T \mid \Delta}$

► **Definition 5** (Context difference). *Context difference, written as $\Gamma \div x = \Delta$, ensures the binding for x is removed from Γ , in case the type bound is unrestricted, or it doesn't even occur, in case the type is linear. Note that the operator \div is left-associative.*

$$\frac{x: \tau \notin \Gamma}{\Gamma \div x = \Gamma} \quad \frac{}{(\Gamma, x: \tau^\omega, \Delta) \div x = \Gamma, \Delta}$$

Rules A-LINARRI, A-UNARRI, A-PAIRE, A-BOXE and A-CTX use it to ensure the bound variables do not escape their scope. In rule A-CTX, we must split the context $(\Gamma \xrightarrow{n} \Delta)$ according to the level n , using only the outer fragment to type the subexpression, mimicking rule T-CTX. The remaining rules are easy to understand.

If we denote by M^\bullet an annotated term (as those in Figure 10), we can easily define an erasure function, $\text{erase}(M^\bullet)$ by induction on the structure of M^\bullet . Then we have:

► **Theorem 6** (Algorithmic correctness).

1. If $\Gamma \vdash M^\bullet \Rightarrow T \mid \Delta$ and Δ^ω , then $\Gamma \vdash \text{erase}(M^\bullet) : T$.
2. If $\Gamma \vdash M : T$, then there is a M^\bullet s.t. $M = \text{erase}(M^\bullet)$, and $\Gamma \vdash M^\bullet \Rightarrow T \mid \Delta$ and Δ^ω .

6 Related work

Our approach to staged metaprogramming is based on contextual modal types. The logical basis for this approach goes back to modal logic S4 [39, 20], where a type $\Box T$ stands for a closed code fragment that evaluates to a value with type T . Box types are then generalized by introducing contexts, such as $\Box(\overline{T_1} \vdash T_2)$, allowing free variables in a code fragment by listing their types in the box type. These types are called contextual modal types [33, 34]. To avoid confusion with contextual types, we call them box types in this paper. Möbius [27] further generalizes contextual modal types to hold contexts with different levels, allowing wider variety of metaprogramming. There seems to be two styles when it comes to formalizing contextual modal types: one is a dual-context style [20, 33, 34] where box values are deconstructed with let-box syntax and meta-variables, and the other is Kripke-style or Fitch-style [18, 20, 32, 49] where box values are deconstructed with an unquote term. In this classification, Möbius [27] can be considered a calculi in dual-context style, but further generalized to allow multi-level contexts. As discussed in the introduction, we chose Möbius as a basis because its syntax is more tractable than the Kripke/Fitch-style formulation.

Intensional analysis [26, 35], i.e. the ability to perform pattern matching on code fragments, is a desirable feature with which to improve our work. Despite the original formulation of Möbius also supporting intensional analysis, extending it to support linear types can be a non-trivial problem. Supporting polymorphism is another potential extension. Möbius supports generating code that depends on polymorphic types, which should be feasible in our type system. Murase et al. proposed polymorphic contexts [32] that can abstract several parts of a context by context variables. Such extension could also be useful in our type system, but it is not clear if we can extend our Möbius-based type system, given that polymorphic contexts as defined by Murase et al. [32] are designed for a Fitch-style formulation.

Type systems for multi-stage programming based on linear-time temporal types [19, 46] represent open code fragments as open terms using quasi-quotations, and have been adopted by recent practical/theoretical type systems [29, 44, 45, 55]. However, it is known that building a sound type system with extensions to run-time evaluation and effectful computation upon linear-time temporal type is non-trivial [28, 40, 46]. As shown in Section 2, we want run-time evaluation of code fragments, and effectful computation sending code fragments via channels. Hence, linear-time temporal types are not a suitable solution towards our goal.

Session types were introduced by Honda et al. [24, 25, 47] in the decade of 1990. The types we employ—input/output and internal/external choice—are from these early works. The types for channels ready to be closed are from Caires et al. [15, 16]. The early works on session types were built on the π -calculus; session types for functional programming were later suggested by Gay and Vasconcelos [23]. The idea of using different identifiers to denote the two ends of a same communication channel is by Vasconcelos [50]. Our semantics is mostly taken from Thiemann et al. [11, 48].

The process and channel creation operators in our language (`fork` and `new`) allow quite flexible patterns of interaction. If used separately, however, they may lead to process deadlocks. In contrast, the works of Caires, Pfenning, Toninho and Wadler interpret session types on linear logic, leading to languages where channel creation and parallel composition are coupled, thus ensuring absence of deadlocks for typable processes [15, 16, 52].

Sessions types are built on top of linear type systems. Wadler proposed a programming language with two classes of types, linear and unrestricted, [51]. Walker studied different substructural types systems, including linear type systems [53]. We use the context split operator and the arrow introduction and elimination from rules from Walker. Unlike the two

475 aforementioned works we segregate unrestricted types by means of a predicate on arbitrary
 476 types, rather than by annotations on types.

477 The only integration of linear type systems and multi-stage programming that we are
 478 aware of is proposed by Ângelo et al. [12]. Here we incorporate unrestricted types and session
 479 types into the type system, and provide an algorithmic type checker and more detailed proofs.

480 7 Conclusion and future work

481 We show the merits of enhancing a session type system with staged meta-programming,
 482 allowing protocols that include code exchange. Such an extension requires the integration
 483 of staged meta-programming into a linear type system (on top of which session types are
 484 defined), accounting for the main technical challenge. We also show our system can be
 485 implemented in a practical way, by providing algorithmic type checking rules, in the style of
 486 [53], as well as correctness results. We further plan on improving the system’s expressivity,
 487 both on the way protocols are defined and on how code is created and used. One could aim
 488 at two orthogonal extensions. The first is polymorphism: both for session types [11] and for
 489 staged meta-programming [27]. The second is the incorporation of context-free session types,
 490 allowing programming with the sequential composition of types [38, 43]. These extensions
 491 would lead to more general and reusable code and protocols.

492 — References —

- 493 1 Boinc. <https://boinc.berkeley.edu/>. Accessed: 2025-04-09.
- 494 2 HTMX. <https://htmx.org/>. Accessed: 2025-02-22.
- 495 3 Meteor.js. <https://www.meteor.com/>. Accessed: 2025-04-09.
- 496 4 Next.js. <https://nextjs.org/>. Accessed: 2025-04-09.
- 497 5 Primegrid. <https://www.primegrid.com/>. Accessed: 2025-04-09.
- 498 6 React server components. <https://react.dev/reference/rsc/server-components>. Ac-
 499 cessed: 2025-04-09.
- 500 7 Scala-loci. <https://scala-loci.github.io/>. Accessed: 2025-04-09.
- 501 8 Unison. <https://www.unison-lang.org/>. Accessed: 2025-04-09.
- 502 9 Ur/Web. <https://github.com/urweb/urweb>. Accessed: 2025-04-09.
- 503 10 Martín Abadi and Marcelo P. Fiore. Syntactic considerations on recursive types. In *LICS*,
 504 pages 242–252. IEEE Computer Society, 1996. doi:10.1109/LICS.1996.561324.
- 505 11 Bernardo Almeida, Andreia Mordido, Peter Thiemann, and Vasco T. Vasconcelos. Polymorphic
 506 lambda calculus with context-free session types. *Inf. Comput.*, 289(Part):104948, 2022.
 507 doi:10.1016/J.IC.2022.104948.
- 508 12 Pedro Ângelo, Atsushi Igarashi, and Vasco T. Vasconcelos. Linear contextual metaprogramming
 509 and session types. In *PLACES*, volume 401 of *EPTCS*, pages 1–10, 2024. doi:10.4204/EPTCS.
 510 401.1.
- 511 13 Vincent Balat. Ocsigen: typing web interaction with objective caml. In *Workshop on ML*,
 512 pages 84–94. ACM, 2006. doi:10.1145/1159876.1159889.
- 513 14 Hendrik Pieter Barendregt. *The lambda calculus - its syntax and semantics*, volume 103 of
 514 *Studies in logic and the foundations of mathematics*. North-Holland, 1985.
- 515 15 Luís Caires and Frank Pfenning. Session types as intuitionistic linear propositions. In
 516 *Proceedings of CONCUR*, volume 6269 of *LNCS*, pages 222–236. Springer, 2010. doi:10.1007/
 517 978-3-642-15375-4_16.
- 518 16 Luís Caires, Frank Pfenning, and Bernardo Toninho. Linear logic propositions as session types.
 519 *Math. Struct. Comput. Sci.*, 26(3):367–423, 2016. doi:10.1017/S0960129514000218.

- 520 17 Byung-Gon Chun, Sunghwan Ihm, Petros Maniatis, Mayur Naik, and Ashwin Patti. Clonecloud:
521 elastic execution between mobile device and cloud. In Christoph M. Kirsch and Gernot
522 Heiser, editors, *European Conference on Computer Systems, Proceedings of the Sixth European
523 conference on Computer systems, EuroSys 2011, Salzburg, Austria, April 10-13, 2011*, pages
524 301–314. ACM, 2011. doi:10.1145/1966445.1966473.
- 525 18 Ranald Clouston. Fitch-style modal lambda calculi. In *FOSSACS*, volume 10803 of *Lecture
526 Notes in Computer Science*, pages 258–275. Springer, 2018. doi:10.1007/978-3-319-89366-2\
527 _14.
- 528 19 Rowan Davies. A temporal logic approach to binding-time analysis. *J. ACM*, 64(1):1:1–1:45,
529 2017. doi:10.1145/3011069.
- 530 20 Rowan Davies and Frank Pfenning. A modal analysis of staged computation. *J. ACM*,
531 48(3):555–604, 2001. doi:10.1145/382780.382785.
- 532 21 Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters.
533 *Commun. ACM*, 51(1):107–113, 2008. doi:10.1145/1327452.1327492.
- 534 22 Simon J. Gay, Peter Thiemann, and Vasco T. Vasconcelos. Duality of session types: The final
535 cut. In *PLACES*, volume 314 of *EPTCS*, pages 23–33, 2020. doi:10.4204/EPTCS.314.3.
- 536 23 Simon J. Gay and Vasco Thudichum Vasconcelos. Linear type theory for asynchronous session
537 types. *J. Funct. Program.*, 20(1):19–50, 2010. doi:10.1017/S0956796809990268.
- 538 24 Kohei Honda. Types for dyadic interaction. In *CONCUR*, volume 715 of *LNCS*, pages 509–523.
539 Springer, 1993. doi:10.1007/3-540-57208-2_35.
- 540 25 Kohei Honda, Vasco Thudichum Vasconcelos, and Makoto Kubo. Language primitives and
541 type discipline for structured communication-based programming. In *ESOP*, volume 1381 of
542 *LNCS*, pages 122–138. Springer, 1998. doi:10.1007/BFb0053567.
- 543 26 Jason Z. S. Hu and Brigitte Pientka. Layered modal type theory - where meta-programming
544 meets intensional analysis. In *ESOP*, volume 14576 of *LNCS*, pages 52–82. Springer, 2024.
545 doi:10.1007/978-3-031-57262-3_3.
- 546 27 Junyoung Jang, Samuel Gélneau, Stefan Monnier, and Brigitte Pientka. Möbius: metapro-
547 gramming using contextual types: the stage where system F can pattern match on itself. *Proc.
548 ACM Program. Lang.*, 6(POPL):1–27, 2022. doi:10.1145/3498700.
- 549 28 Yuki Yoshi Kameyama, Oleg Kiselyov, and Chung-chieh Shan. Shifting the stage - staging with
550 delimited control. *J. Funct. Program.*, 21(6):617–662, 2011. doi:10.1017/S0956796811000256.
- 551 29 Oleg Kiselyov. Metaocaml theory and implementation. *CoRR*, abs/2309.08207, 2023.
552 URL: <https://doi.org/10.48550/arXiv.2309.08207>, arXiv:2309.08207, doi:10.48550/
553 ARXIV.2309.08207.
- 554 30 Robin Milner. Functions as processes. *Math. Struct. Comput. Sci.*, 2(2):119–141, 1992.
555 doi:10.1017/S0960129500001407.
- 556 31 John C. Mitchell. *Foundations for programming languages*. Foundation of computing series.
557 MIT Press, 1996.
- 558 32 Yuito Murase, Yuichi Nishiwaki, and Atsushi Igarashi. Contextual modal type theory with
559 polymorphic contexts. In *ESOP*, volume 13990 of *LNCS*, pages 281–308. Springer, 2023.
560 doi:10.1007/978-3-031-30044-8_11.
- 561 33 Aleksandar Nanevski and Frank Pfenning. Staged computation with names and necessity. *J.
562 Funct. Program.*, 15(5):893–939, 2005. doi:10.1017/S095679680500568X.
- 563 34 Aleksandar Nanevski, Frank Pfenning, and Brigitte Pientka. Contextual modal type theory.
564 *ACM Trans. Comput. Log.*, 9(3):23:1–23:49, 2008. doi:10.1145/1352582.1352591.
- 565 35 Lionel Parreaux, Antoine Voizard, Amir Shaikhha, and Christoph E. Koch. Unifying analytic
566 and statically-typed quasiquotes. *Proc. ACM Program. Lang.*, 2(POPL):13:1–13:33, 2018.
567 doi:10.1145/3158101.
- 568 36 Frank Pfenning and Rowan Davies. A judgmental reconstruction of modal logic. *Math. Struct.
569 Comput. Sci.*, 11(4):511–540, 2001. doi:10.1017/S0960129501003322.
- 570 37 Benjamin C. Pierce. *Types and programming languages*. MIT Press, 2002.

- 38 Diogo Poças, Diana Costa, Andreia Mordido, and Vasco T. Vasconcelos. System f^{μ}_{ω} with context-free session types. In *ESOP*, volume 13990 of *Lecture Notes in Computer Science*, pages 392–420. Springer, 2023. doi:10.1007/978-3-031-30044-8_15.
- 39 Dag Prawitz. Natural deduction. *Almqvist & Wiksell, Stockholm, Sweden*, 1965.
- 40 Morten Rhiger. Staged computation with staged lexical scope. In Helmut Seidl, editor, *Programming Languages and Systems - 21st European Symposium on Programming, ESOP 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings*, volume 7211 of *Lecture Notes in Computer Science*, pages 559–578. Springer, 2012. doi:10.1007/978-3-642-28869-2_28.
- 41 Manuel Serrano and Vincent Prunet. A glimpse of hopjs. In *ICPF*, pages 180–192. ACM, 2016. doi:10.1145/2951913.2951916.
- 42 Tim Sheard and Simon Peyton Jones. Template meta-programming for haskell. In *Workshop on Haskell*, pages 1–16. ACM, 2002. doi:10.1145/581690.581691.
- 43 Gil Silva, Andreia Mordido, and Vasco T. Vasconcelos. Subtyping context-free session types. In *CONCUR*, volume 279 of *LIPIcs*, pages 11:1–11:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023. doi:10.4230/LIPICS.CONCUR.2023.11.
- 44 Nicolas Stucki, Jonathan Immanuel Brachthäuser, and Martin Odersky. Multi-stage programming with generative and analytical macros. In Eli Tilevich and Coen De Roover, editors, *GPCE '21: Concepts and Experiences, Chicago, IL, USA, October 17 - 18, 2021*, pages 110–122. ACM, 2021. doi:10.1145/3486609.3487203.
- 45 Takashi Suwa and Atsushi Igarashi. An ml-style module system for cross-stage type abstraction in multi-stage programming. In Jeremy Gibbons and Dale Miller, editors, *Functional and Logic Programming - 17th International Symposium, FLOPS 2024, Kumamoto, Japan, May 15-17, 2024, Proceedings*, volume 14659 of *Lecture Notes in Computer Science*, pages 237–272. Springer, 2024. doi:10.1007/978-981-97-2300-3_13.
- 46 Walid Taha and Michael Florentin Nielsen. Environment classifiers. In *POPL*, pages 26–37. ACM, 2003. doi:10.1145/604131.604134.
- 47 Kaku Takeuchi, Kohei Honda, and Makoto Kubo. An interaction-based language and its typing system. In *PARLE*, volume 817 of *LNCS*, pages 398–413. Springer, 1994. doi:10.1007/3-540-58184-7_118.
- 48 Peter Thiemann and Vasco T. Vasconcelos. Context-free session types. In *ICFP*, pages 462–475. ACM, 2016. doi:10.1145/2951913.2951926.
- 49 Nachiappan Valliappan, Fabian Ruch, and Carlos Tom'e Corti nas. Normalization for fitch-style modal calculi. *Proc. ACM Program. Lang.*, 6(ICFP):772–798, 2022. doi:10.1145/3547649.
- 50 Vasco T. Vasconcelos. Fundamentals of session types. *Inf. Comput.*, 217:52–70, 2012. doi:10.1016/J.IC.2012.05.002.
- 51 Philip Wadler. Linear types can change the world! In *Programming concepts and methods*, page 561. North-Holland, 1990.
- 52 Philip Wadler. Propositions as sessions. *J. Funct. Program.*, 24(2-3):384–418, 2014.
- 53 David Walker. *Advanced Topics in Types and Programming Languages*, chapter Substructural Type Systems, pages 3–44. The MIT Press, 2005.
- 54 Tom White. *Hadoop - The Definitive Guide: Storage and Analysis at Internet Scale (4. ed., revised & updated)*. O'Reilly, 2015.
- 55 Ningning Xie, Leo White, Olivier Nicole, and Jeremy Yallop. Macocaml: Staging composable and compilable macros. *Proc. ACM Program. Lang.*, 7(ICFP):604–648, 2023. doi:10.1145/3607851.
- 56 Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In *HotCloud'10*. USENIX Association, 2010.

A Extra example: Code splicing

We now present the generation of a code fragment to send a fixed number of integers on a stream. The type of streams, as seen from the side of processes writing on the stream, is as follows.

```
type Stream =  $\oplus$ {More: !Int.Stream, Done: Close}
```

The writer chooses between selecting More values or selecting Done. In the former case, the writer sends an integer value (!Int) and “goes back to the beginning” (Stream); in the latter case the writer must close the channel (Close). Function sendFives accepts an integer value and returns a code fragment. The code fragment requires a channel endpoint (of type Stream) and, when executed, produces a unit value; its type is written [Stream \vdash Unit], which is called a *box type*. We proceed by pattern-matching on the parameter.

```
sendFives : Int  $\rightarrow$  [Stream  $\vdash$  Unit]
sendFives 0 = box (y. close (select Done y))
sendFives n = let box u = sendFives (n - 1)
              in box (x. u[send 5 (select More x)])
```

When all values have been sent on the stream (when n is 0), all it remains is to select Done and then close the channel. The box term generates code under a variable environment (an evaluation context), in this case containing variable y alone, denoting the channel endpoint. The code fragment $y. \text{close} (\text{select Done } y)$ enclosed inside box is called a *contextual value*.

When there are values left to be sent (when n is different from 0), we recursively compute code to send $n-1$ values, unbox it storing the resulting contextual value (say $z. \text{close} (\text{select Done } z)$) in u , and then prepare code to send the n -th value. The use of the let-box bound variable u , which is given a *contextual type* Stream \vdash Unit (without square brackets []), always takes the form $u[e]$, where e is a term of type Stream, and results in a term of type Unit obtained by substituting e (without being evaluated) for z . The part $[e]$ is called an *explicit substitution*. In this example, term $u[\text{send } 5 (\text{select More } x)]$ applies u of type Stream \vdash Unit to term $\text{send } 5 (\text{select More } x)$ of type Stream. If u is the contextual value $z. \text{close} (\text{select Done } z)$, then $\text{box } x. u[\text{send } 5 (\text{select More } x)]$ evaluates to $\text{box } x. \text{close} (\text{select Done } (\text{send } 5 (\text{select More } x)))$, a piece of code that sends number 5 on channel x and then closes the channel.

The main difference between a contextual type such as Stream \vdash Unit and the corresponding box type ([Stream \vdash Unit]) is that the latter kind of types represents *first-class* code values—one can pass it to another function or store into a data structure—whereas the former kind are second-class—a variable of type Stream \vdash Unit may only be used with an explicit substitution to compose another piece of code. See work on contextual modal type theory (e.g., [34]) for more details.

We may now compute and run code to send a fixed number of integer values.

```
send4Fives : Stream  $\rightarrow$  Unit
send4Fives c = let box u = sendFives 4 in u[c]
```

Term sendFives 4 is a boxed code fragment (a term) of type [Stream \vdash Unit]. Then, u is an unboxed code fragment (a contextual value) of contextual type Stream \vdash Unit. We provide the contextual value with an explicit substitution $[c]$. The whole let term then amounts to running the code

```
close (select Done (send 5 (select More (... send 5 (select More c)
...))))
```

23:20 Session-typed Staged Metaprogramming

without calling function `sendFives` or using recursion in any other form.

The next example transmits code on channels. Imagine a server preparing code on behalf of clients. The server uses a channel to interact with its clients: it first receives a number n , then replies with code to send n fives, and finally waits for the channel to be closed. The type of the communication channel is as follows.

```
678 type Builder = ?Int .![ Stream ⊢ Unit ].Wait
679
680
```

The server receives n on a given channel and computes the code using a call to `sendFives`. It then waits for the channel to be closed.

```
683 serveFives : Builder → Unit
684 serveFives c =
685   let (n, c) = receive c in wait (send (sendFives n) c)
686
687
```

On the other end of the channel sits a client: it sends a number (4 in this case), receives the code (of type $[Stream ⊢ Unit]$), closes the channel and evaluates the code received.

```
690 sendFives' : Dual Builder → Stream → Unit
691 sendFives' c d =
692   let (code, c) = receive (send 4 c) in close c ;
693   let box u = code in u[d]
694
695
```

The **Dual** operator on session types provides a view of the other end of the channel. In this case, **Dual** Builder is the type $!Int .?[Stream ⊢ Unit].Close$, where $!$ is turned into $?$ and **Wait** is turned into **Close** (and conversely in both cases). Notice that `code` is a boxed code fragment of type $[Stream ⊢ Unit]$, hence u is the corresponding code fragment (of type $Stream ⊢ Unit$) and $u[d]$ runs the code on channel d .

To complete the example we need a function for reading streams, that is, a consumer of type **Dual** Stream $→$ Unit. Function `readInts` reads and discards all integer values on the stream and then waits for the stream to be closed. Here we proceed by pattern matching on the label received on the channel.

```
705 readInts : Dual Stream → Unit
706 readInts (Done c) = wait c
707 readInts (More c) = let (_, c) = receive c in readInts c
708
709
```

Finally, the main thread forks two threads—one running `serveFives`, the other to collect the integer values (`readInts`)—and continues with `sendFives'`. We take advantage of a primitive function, `forkWith` that expects a suspended computation (a thunk), creates a new channel, forks the thunk on one end of the channel, and returns the other end of the channel for further interaction.

```
715 main : Unit
716 main =
717   let c = forkWith (λ_. serveFives) in — c : Dual Builder
718   let d = forkWith (λ_. readInts) in — d : Stream
719   sendFives' c d
720
721
```

The interaction among the three processes is depicted in Figure 11, where the code fragment produced by function `serveFives` and transmitted to `sendFives'` is

```
724 box (c.close (select Done (send 5 (select More (...(select More c)
725   ...))))))
726
```

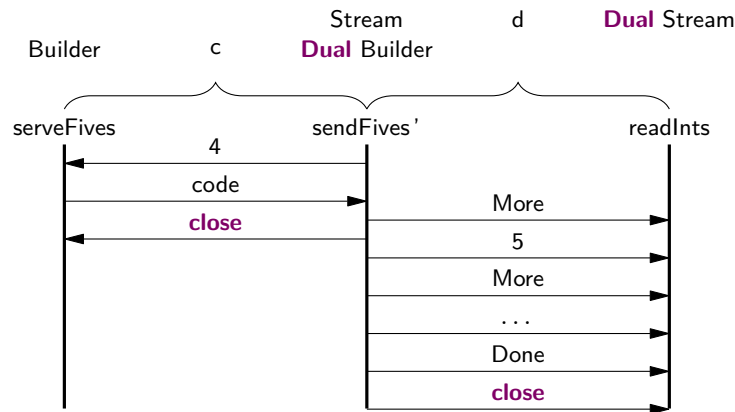


Figure 11 Message sequence chart for the SendFives example

Contextual metaprogramming for session types runs on the top of a linear type system: each resource is classified as *linear* (used exactly once) or *unrestricted* (used an unbounded number of times, including zero). Channel endpoints are always linear so that protocols may not encounter unexpected interactions. Functions may be linear or unrestricted. All the functions we have seen so far are unrestricted because they do not capture free linear values. The recursive functions (`sendFives` for example) are necessarily unrestricted, for they are discarded at the end of recursion. Some others could be classified as linear if so desired. One such example is function `serveFives` which is used exactly once in function `main`. We annotate arrows with ω for unrestricted and 1 for linear. In examples, we often omit the ω label. Function `serveFives` could as well be of type $\text{Builder} \rightarrow_1 \text{Unit}$.

Boxed code fragments are always unrestricted, so that they may be used as many times as needed. Below are examples where a code fragment (denoted by u and of type $\text{Stream} \vdash \text{Stream}$) that sends 5 on a given channel is duplicated (first example) or discarded (second example).

```

742 sendTwice : Stream → Unit
743 sendTwice = let box u = box (y. send 5 (select More y))
744             in λx. close (select Done (u[u[x]]))
745
746 sendNone : Stream → Unit
747 sendNone = let box u = box (y. send 5 (select More y))
748             in λx. close (select Done x)

```

In the first case, two 5 messages are sent on channel x and then the channel is closed; in the second, the only interaction on the channel is to close it.

Contextual types are indexed by *levels*. We have been writing $\text{Stream} \vdash \text{Unit}$ as an abbreviation for $\text{Stream} \vdash_1 \text{Unit}$, a level-1 contextual type. One may distinguish two kinds of code fragments: those with box types (like $[\text{Stream} \vdash \text{Stream}]$) stored in level-0 variables, and those with contextual types (like $\text{Stream} \vdash \text{Stream}$) stored in level-1 or higher variables. The example above demonstrates that level-1 contextual values can be duplicated and discarded. It is worth noting that level-0 resources with box types can be unrestricted as well. The following variant of `sendTwice` exemplifies the unrestricted nature of box types, where the level-0 variable z with a box type $[\text{Stream} \vdash_1 \text{Stream}]$ is used twice.

```

760 sendTwice' : Stream → Unit
761 sendTwice' = let z = box (y. send 5 (select More y))
762

```


$$\begin{aligned}
P &\equiv \langle \text{unit} \rangle \mid P & P \mid Q &\equiv Q \mid P & (P_1 \mid P_2) \mid P_3 &\equiv P_1 \mid (P_2 \mid P_3) \\
(\nu xy)P \mid Q &\equiv (\nu xy)(P \mid Q) & (\nu xy)P &\equiv (\nu yx)P \\
(\nu xy)(\nu zw)P &\equiv (\nu zw)(\nu xy)P
\end{aligned}$$

■ **Figure 12** Structural congruence $P \equiv Q$

```

763   let box u = z in
764   let box v = z in
765   in λx. close (select Done u[v[x]])
766

```

Code fragments are unrestricted resources in our type system because boxes are designed to be independent of linear resources. Level- n boxes abstract all variables at levels lower than n , and this means that level-0 variables are always abstracted. Our type system only allows level-0 variables to be linear. Hence, boxes will not depend on linear resources, and it is safe to duplicate or discard code fragments even when they are code fragments that compute linear resources.

So far, all our examples showcase contextual types of up to level 1. Here is an example featuring level 2 types, where code to send an arbitrary number of integer values is sandwiched between code to send integer 100.

```

776 sandwich : [Stream, (Stream ⊢ 1 Stream) ⊢ 2 Unit]
777 sandwich = box (x, u.
778   let x1 = send 100 (select More x) in
779   let x2 = u[x1] in
780   let x3 = send 100 (select More x2) in
781   close x3)
782

```

The communication template `sandwich` is code parameterised on two arguments. The first argument is a channel endpoint of type `Stream`, which is a level 0 type. The second is code parameterised on a channel endpoint that evaluates to another channel endpoint, both of type `Stream`. Then, the second argument is of level 1. Hence, the type of the expression `sandwich` is a level 2 box type.

Finally we splice in the code of `sandwich` into function `sandwichMe`, allowing both a channel endpoint and a supplied value, in the form of the box, to be spliced into the template code, filling the value to be sent in second. The code `sandwichMe box(z. send 5 (select More (send 5 (select More z))))` results in sending four integer values (100, 5, 5 and 100) before closing the channel.

```

794 sandwichMe : [Stream ⊢ 1 Stream] → [Stream ⊢ 1 Unit]
795 sandwichMe y = let box u = y in
796   let box v = sandwich in
797   box (x. v[x, u])
798

```

800 B Processes

801 *Reduction* on processes is given by the relation $P \rightarrow Q$, defined by the rules in Figure 13.
 802 As customary in the π -calculus, reduction builds on a further relation—structural congruence

$$\begin{array}{c}
\text{R-EXP} \\
\frac{M \rightarrow N}{\langle M \rangle \rightarrow \langle N \rangle} \\
\\
\text{R-FORK} \quad \frac{}{\langle E[\text{fork } v] \rangle \rightarrow \langle E[\text{unit}] \rangle \mid \langle v \text{ unit} \rangle} \quad \text{R-NEW} \quad \frac{}{\langle E[\text{new}] \rangle \rightarrow (\nu xy) \langle E[(x, y)] \rangle} \\
\\
\text{R-CLOSE} \quad \frac{}{(\nu xy) (\langle E[\text{close } x] \rangle \mid \langle F[\text{wait } y] \rangle) \rightarrow \langle E[\text{unit}] \rangle \mid \langle F[\text{unit}] \rangle} \\
\\
\text{R-COM} \quad \frac{}{(\nu xy) (\langle E[\text{send } v \ x] \rangle \mid \langle F[\text{receive } y] \rangle) \rightarrow (\nu xy) \langle E[x] \rangle \mid \langle F[(v, y)] \rangle} \\
\\
\text{R-BRANCH} \quad \frac{}{(\nu xy) (\langle E[\text{select } l' \ x] \rangle \mid \langle F[\text{match } y \text{ with } \{l \rightarrow M_l\}^{l \in L}] \rangle) \rightarrow (\nu xy) (\langle E[x] \rangle \mid \langle F[M_{l'} y] \rangle)} \\
\\
\text{R-PAR} \quad \frac{P \rightarrow P'}{P \mid Q \rightarrow P' \mid Q} \quad \text{R-RES} \quad \frac{P \rightarrow Q}{(\nu xy) P \rightarrow (\nu xy) Q} \quad \text{R-STRUCT} \quad \frac{P \equiv P' \quad P' \rightarrow Q' \quad Q' \equiv Q}{P \rightarrow Q}
\end{array}$$

■ **Figure 13** Process reduction $\boxed{P \rightarrow P}$

803 $P \equiv Q$ —that provides for the syntactic rearrangement of processes, while preparing these
 804 for reduction [30].

805 *Structural congruence* is the least congruence relation generated by the axioms in Figure 12.
 806 The first axiom posits thread $\langle \text{unit} \rangle$ as the neutral element of parallel composition. The next
 807 two axioms tell that parallel composition is commutative and associative. Law $(\nu xy)P \mid Q \equiv$
 808 $(\nu xy)(P \mid Q)$ is called scope extrusion and allows the scope of a ν -binder to expand to a new
 809 process Q or to retract from it, as needed. Further axioms allow exchanging the order of
 810 the two endpoints of a channel, and exchanging the order of ν -binders. In scope extrusion,
 811 because of the variable convention, the condition ‘ x, y not free in Q ’ is redundant: x and y
 812 occur bound in $(\nu xy)P$ and therefore cannot occur free in Q .

813 We now describe the reduction rules. Rule R-EXP lifts term evaluation to process reduction.
 814 Rule R-FORK creates a new thread. Value v is supposed to be a suspended computation (a
 815 thunk), so that $v \text{ unit}$ runs the computation. The hole in the context in the original thread
 816 is filled with unit : all communication between the two threads must be accomplished via
 817 message passing on channels free in v . Rule R-NEW creates a new channel denoted by its two
 818 endpoints x and y . The hole in the thread is filled with a pair (x, y) so that the thread may
 819 manipulate the channel.

820 The next three rules manipulate channels via their endpoints x, y . For ease of reading
 821 we abbreviate contextual term value $x[\varepsilon]$ to x , and similarly for y . Rule R-CLOSE closes a
 822 channel: one thread must be closing one endpoint, the other waiting for the channel to be
 823 closed. The ν -binder is eliminated since the channel cannot be further used. Rule R-COM
 824 exchanges a value between a **send** and a **receive** thread. The result of sending is the endpoint
 825 itself, x ; the result of receiving is a pair composed of the value exchanged and the endpoint y .
 826 The two threads may then continue exchanging values on these endpoints. Rule R-BRANCH
 827 exercises the choice of a branch in a **match** process. The label selected on a thread (l') chooses
 828 branch $M_{l'}$ on the other thread. The result of selection is the endpoint itself, as in **send**; the
 829 result of match is $M_{l'}$, which is supposed to be a function, applied to endpoint y . Similarly
 830 to R-COM, the two threads may continue interacting on channel xy . Finally, the last three
 831 rules allow reduction under parallel composition and ν -binders, and incorporate structural

$$\begin{array}{c}
\text{R-EXP} \\
\frac{\Gamma \vdash M : \text{Unit}}{\Gamma \vdash \langle M \rangle} \\
\\
\text{P-PAR} \\
\frac{\Gamma \vdash P \quad \Delta \vdash Q}{\Gamma \circ \Delta \vdash P \mid Q} \\
\\
\text{P-RES} \\
\frac{\Gamma, x : (\varepsilon \vdash^0 R), y : (\varepsilon \vdash^0 S) \vdash P \quad R \perp S}{\Gamma \vdash (\nu xy)P}
\end{array}$$

■ **Figure 14** Process formation $\boxed{\Gamma \vdash P}$

congruence in reduction.

Even if programmers are not supposed to write processes directly, we still need to type them in order to state and prove the type safety result for the process language. The rules, in Figure 14, are straightforward and taken from Vasconcelos [50] and Thiemann et al. [11, 48]. In a thread $\langle M \rangle$, term M must be of Unit type. Given that the result of the evaluation is discarded, any unrestricted type would do. The parallel composition splits the context in the conclusion and uses one part for each process. Scope restriction introduces entries for each of the two channel ends. These must be of dual types.

C Proofs of the main results

► **Lemma 7** (Basic properties of context split [50]). *Let $\Gamma = \Gamma_1 \circ \Gamma_2$.*

1. $\mathcal{U}(\Gamma) = \mathcal{U}(\Gamma_1) = \mathcal{U}(\Gamma_2)$.
2. *If $x : \tau^1 \in \Gamma$ then either $x : \tau^1 \in \Gamma_1$ and $x : \tau' \notin \Gamma_2$; or $x : \tau^1 \in \Gamma_2$ and $x : \tau' \notin \Gamma_1$, for any τ' .*
3. $\Gamma = \Gamma_2 \circ \Gamma_1$.
4. *If $\Gamma_1 = \Delta_1 \circ \Delta_2$ then $\Delta = \Delta_2 \circ \Gamma_2$ and $\Gamma = \Delta_1 \circ \Delta$.*

Proof. A straightforward induction on the structure of context Γ (taken from [?], Lemma 7.1). ◀

C.1 Soundness

► **Lemma 8** (Unrestricted weakening). *If $\Gamma \vdash M : T$ and $\tau \omega$, then $\Gamma, x : \tau \vdash M : T$.*

Proof. By rule induction on $\Gamma \vdash M : T$. ◀

► **Lemma 9** (Unrestricted strengthening). *If $\Gamma, x : \tau \vdash M : T$ and $x \notin \text{free } M$, then $\tau \omega$ and $\Gamma \vdash M : T$.*

Proof. By rule induction on $\Gamma, x : \tau \vdash M : T$. ◀

► **Lemma 10** (Substitution). *Let $\Gamma \vdash \sigma : \tau$ and $\Gamma \geq n$ where n is the level of τ and suppose $\tau \perp$ or $\Gamma \omega$.*

1. *If $\Delta, x : \tau \vdash M : T$ and $\Gamma \circ \Delta$ is well defined, then $\Gamma \circ \Delta \vdash \{\sigma/x\}M : T$.*
2. *If $\Delta, x : \tau \vdash \rho : \tau'$ and $\Gamma \circ \Delta$ is well defined, then $\Gamma \circ \Delta \vdash \{\sigma/x\}\rho : \tau'$.*

Proof. We prove the two statements simultaneously by lexicographic induction on the level n of τ and the size of the derivation of $\Delta, x : \tau \vdash M : T$ or $\Delta, x : \tau \vdash \rho : \tau'$ with case analysis on the last typing rule used. In this proof, we write $\Gamma \setminus x$ for the typing environment obtained by the binding of x from Γ (if any).

We outline the most interesting cases.

Case rule T-VAR: We have $M = y[\bar{\rho}]$ and $\Delta, x : \tau = \Gamma_1 \circ \dots \circ \Gamma_m \circ (\Delta_0^\omega, y : (\bar{\tau} \vdash^{n'} T))$ and $\bar{\Gamma} \vdash \bar{\rho} : \bar{\tau}$ for some $y, \bar{\rho}, \Delta_0, \bar{\Gamma}, \bar{\tau}, n'$. There are two possible sub-cases to consider:

- 866 ■ Subcase $x \neq y$: We will show that $\Gamma \circ \Delta \vdash y[\{\sigma/x\}\bar{\rho}] : T$ since $\{\sigma/x\}(y[\bar{\rho}]) = y[\{\sigma/x\}\bar{\rho}]$.
- 867 ■ Subsubcase $\tau 1$: It must be the case that there exists i such that $\Gamma_i = \Gamma'_i, x : \tau$ for some
- 868 Γ'_i and for $j \neq i, x \notin \text{dom}(\Gamma_j)$. It is easy to show that $\Gamma \circ \Gamma'_j$ is well defined, because we
- 869 assumed that $\Gamma \circ \Delta$ is well defined. By the induction hypothesis, $\Gamma \circ \Gamma'_i \vdash \{\sigma/x\}\rho_i : \tau_i$.
- 870 For $j \neq i, \{\sigma/x\}\rho_j = \rho_j$ as $x \notin \text{dom}(\Gamma_j)$ and thus x is not a free variable of ρ_j . By
- 871 T-VAR, we have $(\Gamma_1 \circ \dots \circ (\Gamma \circ \Gamma'_i) \circ \dots \circ \Gamma_m) \circ (\Delta_0^\omega, y : (\bar{\tau} \vdash^{n'} T)) \vdash y[\{\sigma/x\}\bar{\rho}] : T$. It is
- 872 easy to show $(\Gamma_1 \circ \dots \circ (\Gamma \circ \Gamma'_j) \circ \dots \circ \Gamma_m) \circ \Gamma \circ (\Delta_0^\omega, y : (\bar{\tau} \vdash^{n'} T)) = \Gamma \circ (\Delta, x : \tau) \setminus x = \Delta$.
- 873 ■ Subsubcase $\tau \omega$. It must be the case that, for any $1 \leq i \leq m, \Gamma_i = \Gamma'_i, x : \tau$
- 874 for some Γ'_i . We also have $\Gamma \omega$ by assumption. Similarly to the above case, by
- 875 the induction hypothesis, for any $i, \Gamma \circ \Gamma'_i \vdash \{\sigma/x\}\rho_i : \tau_i$. By T-VAR, we have
- 876 $((\Gamma \circ \Gamma'_1) \circ \dots \circ (\Gamma \circ \Gamma'_m)) \circ (\Delta_0^\omega, y : (\bar{\tau} \vdash^{n'} T)) \vdash y[\{\sigma/x\}\bar{\rho}] : T$. It is easy to show
- 877 $((\Gamma \circ \Gamma'_1) \circ \dots \circ (\Gamma \circ \Gamma'_m)) \circ (\Delta_0^\omega, y : (\bar{\tau} \vdash^{n'} T)) = \Gamma \circ (\Gamma'_1 \circ \dots \circ \Gamma'_m) \circ (\Delta_0^\omega, y : (\bar{\tau} \vdash^{n'}$
- 878 $T)) = \Gamma \circ (\Delta, x : \tau) \setminus x = \Gamma \circ \Delta$.
- 879 ■ Subcase $x = y$: We have $\tau = \bar{\tau} \vdash^{n'} T$ and $\sigma = \bar{z}.N$ for some \bar{z} and N and $\{\sigma/x\}M =$
- 880 $\{\{\sigma/x\}\bar{\rho}/\bar{z}\}N$. We will show that $\Gamma \circ \Delta \vdash \{\{\sigma/x\}\bar{\rho}/\bar{z}\}N : T$. By T-CTX, we have
- 881 $\Gamma, \bar{z} : \bar{\tau} \vdash N : T$. Note that $\Gamma \geq^n$ and $(\bar{z} : \bar{\tau}) <^n$. We have further case analysis on whether
- 882 $n = 0$ or not and the multiplicity of τ .
- 883 ■ Subsubcase $n = 0$. In this case, \bar{z} must be empty. It suffices to show $\Gamma \circ \Delta \vdash N : T$
- 884 but it follows from Weakening. (It is easy to show $\Delta \omega$ because $\bar{\Gamma}$ is also empty and
- 885 $\Delta = \Delta_0$.)
- 886 ■ Subsubcase $n > 0$ and $\tau 1$. Similarly to the case for $x \neq y$ and $\tau 1$, there exists i
- 887 such that $\Gamma_i = \Gamma'_i, x : \tau$ and $\Gamma \circ \Gamma'_i \vdash \{\sigma/x\}\rho_i : \tau_i$ and, for $j \neq i, \Gamma'_j \vdash \{\sigma/x\}\rho_j : \tau_j$.
- 888 By the induction hypothesis (note that the levels of τ_j is strictly lower than n),
- 889 $\Gamma_1 \circ \dots \circ (\Gamma \circ \Gamma'_i) \circ \dots \circ \Gamma_m \vdash \{\{\sigma/x\}\bar{\rho}/\bar{z}\}N : T$. It is easy to show that $\Delta =$
- 890 $\Gamma_1 \circ \dots \circ \Gamma'_i \circ \dots \circ \Gamma_m \circ \Delta_0^\omega$. Weakening finishes the case.
- 891 ■ Subsubcase $n > 0$ and $\tau \omega$ is similar.

892 **Case rule T-UNARRI.** Let that $M = \lambda y.M_0$ and $T = T' \rightarrow_\omega U'$. We then have that $\Gamma \vdash \sigma : \tau$

893 and $\Delta, x : \tau \vdash \lambda y.M_0 : T' \rightarrow_\omega U'$. We want to show $\Gamma \circ \Delta \vdash \{\sigma/x\}(\lambda y.M_0) : T' \rightarrow_\omega U'$.

894

895 We have that $\{\sigma/x\}(\lambda y.M_0) = \lambda y.\{\sigma/x\}M_0$. By T-UNARRI, we have that $\Delta \omega$ and $\tau \omega$ and

896 $\Delta, x : \tau, y : (\varepsilon \vdash^0 T') \vdash M_0 : U'$. Without loss of generality, we can assume $y \notin \text{dom}(\Gamma)$. Now

897 we have two subcases depending on the multiplicity of the type of y .

- 898 ■ Subcase $(\varepsilon \vdash^0 T') 1$: $\Gamma \circ (\Delta, y : (\varepsilon \vdash^0 T'))$ is well defined. By the induction hypothesis,
- 899 $\Gamma \circ (\Delta, y : (\varepsilon \vdash^0 T')) \vdash \{\sigma/x\}M_0 : U'$, which is equivalent to $(\Gamma \circ \Delta), y : (\varepsilon \vdash^0 T') \vdash$
- 900 $\{\sigma/x\}M_0 : U'$.
- 901 ■ Subcase $(\varepsilon \vdash^0 T') \omega$: $(\Gamma, y : (\varepsilon \vdash^0 T')) \circ (\Delta, y : (\varepsilon \vdash^0 T'))$ is well defined and equal to
- 902 $(\Gamma \circ \Delta), y : (\varepsilon \vdash^0 T')$. By Weakening and the induction hypothesis, $(\Gamma \circ \Delta), y : (\varepsilon \vdash^0 T') \vdash$
- 903 $\{\sigma/x\}M_0 : U'$.

904 Since $\tau \omega$, we have $\Gamma \omega$ by assumption. Thus, $(\Gamma \circ \Delta) \omega$. By T-UNARRI, $\Gamma \circ \Delta \vdash \{\sigma/x\}(\lambda y.M_0) :$

905 $T' \rightarrow_\omega U'$.

906 **Case rule T-BoxI.** We have $\Delta, x : \tau = \Delta_1, \Delta_2$ and $\Delta_1 \omega$ and $\Delta_2 \omega$ (hence $\tau \omega$ and $\Gamma \omega$) and

907 $M = \text{box } \rho$ and $T = \Box(\bar{\tau}' \vdash^n U)$ and $\Delta_1 \vdash \rho : (\bar{\tau}' \vdash^n U)$. Since $\Gamma \circ \Delta$ is well defined by

908 assumption and $\Gamma \omega$ and $\Delta \omega$, it must be the case that $\Gamma = \Delta$ and $\Gamma \circ \Delta = \Gamma$. We want to

909 show $\Delta \vdash \{\sigma/x\}\text{box } \rho : \Box(\bar{\tau}' \vdash^k U)$.

910 By Weakening, $\Delta, x : \tau \vdash \rho : (\bar{\tau}' \vdash^n U)$. By the induction hypothesis, $\Delta \vdash \{\sigma/x\}\rho :$

911 $(\bar{\tau}' \vdash^n U)$. Rule T-BoxI finishes the case.

912 **Case rule T-BoxE.** We have $M = \text{let box } y = M' \text{ in } N'$ and $\Delta, x : \tau = \Delta_1 \circ \Delta_2$ and

913 $\Delta_1 \vdash M' : \Box\tau'$ and $\Delta_2, y : \tau' \vdash N' : T$ for some $y, M', N', \Delta_1, \Delta_2$. We want to show

914 $\Gamma \circ \Delta \vdash \{\sigma/x\}(\text{let box } y = M' \text{ in } N') : T$. We have that $\{\sigma/x\}(\text{let box } y = M' \text{ in } N') =$
 915 $\text{let box } y = \{\sigma/x\}M' \text{ in } \{\sigma/x\}N'$. There are two possible sub-cases to consider:

- 916 ■ Subcase $\tau\omega$: By T-BoxE, $\Delta_i = \Delta'_i, x: \tau$ (for $i = 1, 2$) for some Δ'_1, Δ'_2 and $\Delta'_1, x: \tau \vdash M' :$
 917 $\Box\tau'$ and $\Delta'_2, x: \tau', y: \tau' \vdash N' : T$. By the induction hypothesis, $\Gamma \circ \Delta'_1 \vdash \{\sigma/x\}M' : \Box\tau'$
 918 and $\Gamma \circ (\Delta'_2, y: \tau') \vdash \{\sigma/x\}N' : T$. We have $\Gamma \omega$ since $\tau \omega$. By T-BoxE and by Lemma 7
 919 , $\Gamma \circ (\Delta'_1 \circ \Delta'_2) \vdash \text{let box } y = \{\sigma/x\}M' \text{ in } \{\sigma/x\}N' : T$. It is easy to see $\Delta = \Delta'_1 \circ \Delta'_2$.
- 920 ■ Subcase $\tau 1$ and x occurs free in M' : By T-BoxE, $\Delta_1 = \Delta'_1, x: \tau$ for some Δ'_1 and
 921 $\Delta'_1, x: \tau \vdash M' : \Box\tau'$ and $\Delta_2, y: \tau' \vdash N' : T$. By the induction hypothesis, $\Gamma \circ \Delta'_1 \vdash$
 922 $\{\sigma/x\}M' : \Box\tau'$. Since $\Delta_2, y: \tau' \vdash N' : T$ and $x: \tau \notin \Delta_2, y: \tau'$, then x does not occur free
 923 in N' and hence, $\{\sigma/x\}N' = N'$. Therefore, $\Delta_2, y: \tau' \vdash \{\sigma/x\}N' : T$. By T-BoxE and
 924 by Lemma 7 , $\Gamma \circ (\Delta'_1 \circ \Delta_2) \vdash \text{let box } y = \{\sigma/x\}M' \text{ in } \{\sigma/x\}N' : T$. It is easy to see
 925 $\Delta = \Delta'_1 \circ \Delta_2$. If instead x occurs free in N' , the proof is similar.

926 **Case rule T-CTX**: We have $\rho = \bar{y}.M$ and $\tau' = \bar{\tau'} \vdash^k T'$ for some \bar{y}, M and $(\Delta, x: \tau) \geq k$ and
 927 $\Delta, x: \tau, \bar{y}: \bar{\tau'} \vdash M : T'$. We want to show $\Gamma \circ \Delta \vdash \{\sigma/x\}\bar{y}.M : \bar{\tau'} \vdash^k T'$.

928 Let $\bar{z}: \bar{\tau''}$ be a subsequence of $\bar{y}: \bar{\tau'}$ with all $\tau''_i \omega$. Then, $(\Gamma, \bar{z}: \bar{\tau''}) \circ (\Delta, \bar{y}: \bar{\tau'}) =$
 929 $(\Gamma \circ \Delta), \bar{y}: \bar{\tau'}$. By weakening, we have $\Gamma, \bar{z}: \bar{\tau''} \vdash \sigma : \tau$. By the induction hypothesis,
 930 $(\Gamma, \bar{z}: \bar{\tau''}) \circ (\Delta, \bar{y}: \bar{\tau'}) \vdash \{\sigma/x\}M : T'$, which is equivalent to $(\Gamma \circ \Delta), \bar{y}: \bar{\tau'} \vdash \{\sigma/x\}M : T'$.
 931 Since $(\Delta, x: \tau) \geq k$, in particular $\tau \geq k$, we have $\Gamma \geq k$. By T-CTX, $\Gamma \circ \Delta \vdash \bar{y}. \{\sigma/x\}M :$
 932 $\bar{\tau'} \vdash^k T'$. ◀

933 ► **Lemma 11** (Context typing). $\Gamma \vdash E[M] : T$ if and only if $\Gamma = \Gamma_1 \circ \Gamma_2$ and $\Gamma_1 \vdash M : U$ and
 934 $\Gamma_2, x: U \vdash E[x] : T$.

935 **Proof.** By structural induction on E . ◀

936 **Proof of Theorem 2.** By rule induction on $M \rightarrow N$. The cases for R-BETA and R-LETBOX
 937 were discussed before. The case for R-SPLIT is similar to that of R-BETA, using the fact that
 938 context split is commutative and associative (Lemma 7). The case for R-FIX is straightforward.
 939 The case for R-CTX follows from Lemma 11. ◀

940 ► **Lemma 12** (Unrestricted values are typed under unrestricted typing environment). If $\Gamma \vdash v : T$
 941 and $T \omega$, then $\Gamma \omega$.

942 **Proof.** By case analysis on the last typing rule used for $\Gamma \vdash v : T$. ◀

943 ► **Lemma 13** (Value substitution). If $\Gamma \vdash v : U$ and $\Delta, x: U \vdash M : T$ and $\Gamma \circ \Delta$ is defined,
 944 then $\Gamma \circ \Delta \vdash \{v/x\}M : T$.

945 **Proof.** By Lemmas 10 and 12. ◀

946 ► **Lemma 14** (Soundness for structural congruence). Let $P \equiv Q$. Then $\Gamma \vdash P$ if and only if
 947 $\Gamma \vdash Q$.

948 **Proof.** By rule induction on $\Gamma \vdash P$ and on $\Gamma \vdash Q$. ◀

949 C.2 Progress for the functional language

950 **Proof of Lemma 15.** The proof is by case analysis on the last rule used in the derivation of
 951 $\Gamma^S \vdash v : T$.

- 952 1. The only rule that applies is T-CONST. Then, we have that $v = c$ and $\text{Unit} = \text{typeof}(c)$.
 953 According to Figure 7, the only constant c such that $\text{typeof}(c) = \text{Unit}$ is `unit`. Hence,
 954 $v = \text{unit}$.

- 955 2. The only rule that applies is T-VAR. We then have that $v = x[\bar{\sigma}]$ and $(\Delta^\omega, \bar{\Gamma} \circ x : (\bar{\tau} \vdash^n$
 956 $T))^S \vdash x[\bar{\sigma}] : T$. Due to the restriction on contexts, we have that $\bar{\tau} \vdash^n T = \varepsilon \vdash^0 T$, and
 957 hence $\bar{\sigma} = \varepsilon$. Therefore, $v = x$, which is the abbreviated form of $x[\varepsilon]$.
- 958 3. The only rules that apply are:
- 959 ■ T-CONST. Therefore, $v = \text{close, wait, send, send } v, \text{ receive, select } l, \text{ new, fork or fix}$.
 - 960 ■ T-LINARRI or T-UNARRI. Therefore, $v = \lambda x.M$.
- 961 4. The only rule that applies is T-PAIRI. Hence, $v = (u, w)$.
- 962 5. The only rule that applies is T-BOXI. Hence, $v = \text{box } \sigma$.

963

964 **Proof of Theorem 3.** By rule induction on the hypothesis, using canonical forms (Lemma 15).
 965 Cases for rules T-CONST, T-VAR, T-LINARRI, T-UNARRI, T-PAIRI and T-BOXI are trivial.

966 Case rule T-ARRE. Then $M = M_1 M_2$ and $\Gamma_1^S \circ \Gamma_2^S \vdash M_1 M_2 : T$, with $\Gamma^S = \Gamma_1^S \circ \Gamma_2^S$.
 967 By T-ARRE, $\Gamma_1^S \vdash M_1 : U \rightarrow_m T$ and $\Gamma_2^S \vdash M_2 : U$. By the induction hypothesis, we have
 968 that, for both M_1 and M_2 , either 1., 2., or 3. apply. There are several cases:

- 969 ■ both M_1 and M_2 are values. Then, by Lemma 15, M_1 is $\lambda x.M'_1$, $\text{close, wait, send, send } v,$
 970 $\text{receive, select } l, \text{ new, fork or fix}$. If $M_1 = \lambda x.M'_1$ then by R-BETA, $M \rightarrow N$, for some N .
 971 If M_1 is $\text{close, wait, send } v, \text{ receive or select } l$, then by T-CONST, U is $\text{Close, Wait, !}T.S,$
 972 $?T.S$ or $\oplus\{l : S_l\}^{l \in L}$, respectively. By Lemma 15, $M_2 = x$, and 3. applies. If M_1 is send,
 973 then M is a value. If M_1 is new , then 3. applies. If M_1 is fork , then 3. applies. If M_1 is
 974 fix , then by R-FIX, $M \rightarrow N$ for some N .
- 975 ■ M_1 is a value and $M_2 \rightarrow N_2$, for some N_2 . Then, by R-CTX, $M \rightarrow N$ for some N .
- 976 ■ M_1 is a value and M_2 fulfils 3. Then 3. applies.
- 977 ■ $M_1 \rightarrow N_1$, for some N_1 . Then, by R-CTX, $M \rightarrow N$ for some N .
- 978 ■ M_1 fulfils 3. Then 3. applies.

979 Case rule T-PAIRE. Then $M = \text{let } (x, y) = M_1 \text{ in } M_2$ and $\Gamma_1^S \circ \Gamma_2^S \vdash M : T$. By T-PAIRE,
 980 we have $\Gamma_1^S \vdash M_1 : U_1 \times U_2$ and $\Delta^S, x : (\varepsilon \vdash^0 S_1), y : (\varepsilon \vdash^0 S_2) \vdash M_2 : T$. By the induction
 981 hypothesis, for both M_1 and M_2 , either 1., 2., or 3. apply. There are several cases:

- 982 ■ M_1 is a value. Then, by Lemma 15, $M_1 = (u, w)$ and $M \rightarrow N$ by R-SPLIT.
- 983 ■ $M_1 \rightarrow N_1$, for some N_1 . Then, by R-CTX, $M \rightarrow N$, for some N .
- 984 ■ M_1 fulfils 3. Then we have that 3. applies.

985 Case rule T-BOXE. Proof similar to the previous case.

986 Case rule T-MATCH. Proof similar to the previous case.

987 C.3 Absence of runtime errors for the process language

988 ► **Lemma 15** (Canonical forms). *Let $\Gamma^S \vdash v : T$.*

- 989 1. *If $T = \text{Unit}$ then $v = \text{unit}$.*
- 990 2. *If $T = \text{Close, Wait, !}T.S, ?T.S, \oplus\{l : S_l\}^{l \in L}$ or $\&\{l : S_l\}^{l \in L}$, then $v = x[\varepsilon]$.*
- 991 3. *If $T = U \rightarrow_m V$ then v is $\lambda x.M$, $\text{close, wait, send, send } v, \text{ receive, select } l, \text{ new, fork or}$
 992 fix .*
- 993 4. *If $T = U \times V$ then $v = (u, w)$.*
- 994 5. *If $T = \Box \tau$ then $v = \text{box } \sigma$.*

995 **Proof.** By case analysis on the last rule used in the derivation of $\Gamma^S \vdash v : T$.

996 **Proof for Theorem 3.** By rule induction on the hypothesis, using canonical forms (Lemma 15).

Proof for Theorem 4. Take $P \equiv (\nu x_1 y_1) \cdots (\nu x_n y_n)(P_1 \mid \cdots \mid P_m)$. Soundness for structural congruence (Lemma 14) guarantees that the latter process is also typable under the empty context. Now take i, j, k such that the subject of P_i is x_k , the subject of P_j is y_k . We know that there is a Γ^S such that $\Gamma^S \vdash P_i \mid P_j$ since all entries in Γ are introduced by rule P-RES. The same rule also ensures that Γ^S contains entries of the form $x_k : R, y_k : S$ with $R \perp S$. Duality guarantees that $(\nu x_k y_k)(P_i \mid P_j)$ is a redex, hence P is not a runtime error. \blacktriangleleft

1005 C.4 Proofs for algorithmic type checking (Section 5)

1006 **► Lemma 16** (Properties of context difference). *Let $\Gamma \div x = \Delta$.*

- 1007 1. $\Delta = \Gamma \setminus \{x : \tau\}$
- 1008 2. $\mathcal{L}(\Gamma) = \mathcal{L}(\Delta)$
- 1009 3. *If $x : \tau \in \Gamma$, then τ^ω and $x : \tau' \notin \Delta$, for some τ' .*

1010 **Proof.** The proof follows by case analysis. \blacktriangleleft

1011 **► Lemma 17** (Algorithmic monotonicity). *If $\Gamma \vdash M^\bullet \Rightarrow T \mid \Delta$ then $\mathcal{U}(\Gamma) = \mathcal{U}(\Delta)$ and*
 1012 *$\mathcal{L}(\Delta) \subseteq \mathcal{L}(\Gamma)$.*

1013 **Proof.** The proof follows by induction on the rules of $\Gamma \vdash M^\bullet \Rightarrow T \mid \Delta$, using Lemma 16. \blacktriangleleft

1014 **► Lemma 18** (Algorithmic linear strengthening). *Assume τ 1. If $\Gamma, x : \tau \vdash M^\bullet \Rightarrow T \mid \Delta, x : \tau$*
 1015 *then $\Gamma \vdash M^\bullet \Rightarrow T \mid \Delta$.*

1016 **Proof.** The proof follows by induction on the rules of $\Gamma \vdash M^\bullet \Rightarrow T \mid \Delta$, using Lemma 16
 1017 and Lemma 17. \blacktriangleleft

1018 **► Lemma 19** (Algorithmic weakening). *If $\Gamma \vdash M^\bullet \Rightarrow T \mid \Delta$ then $\Gamma, x : \tau \vdash M^\bullet \Rightarrow T \mid \Delta, x : \tau$.*

1019 **Proof.** The proof follows by induction on the rules of $\Gamma \vdash M^\bullet \Rightarrow T \mid \Delta$, using Lemma 16. \blacktriangleleft

1020 **Proof of Theorem 6, Item 1.** The proof follows by induction on the rules of $\Gamma \vdash M^\bullet \Rightarrow T \mid$
 1021 Δ , using Lemma 16, Lemma 17 and Lemma 18. The proof for rule A-CONST is trivial. The
 1022 proofs for rules A-UNVAR, A-LINVAR, A-ARRE, A-PAIRI, A-PAIRE and A-MATCH follow the
 1023 same strategy as the proof for rule A-BOXE. We detail the following cases:

- 1024 ■ Rule A-LINARRI. Assume $(\Delta \div x)^\omega$ and $\Gamma \vdash \lambda_1 x : T.M^\bullet \Rightarrow T \rightarrow_1 U \mid \Delta \div x$. By Lemma 16,
 1025 we that $\mathcal{L}(\Delta) = \mathcal{L}(\Delta \div x)$, hence Δ^ω . By rule A-LINARRI, $\Gamma, x : (\varepsilon \vdash^0 T) \vdash M^\bullet \Rightarrow U \mid \Delta$.
 1026 By the induction hypothesis, we have $\Gamma, x : (\varepsilon \vdash^0 T) \vdash \text{erase}(M^\bullet) : U$. By rule T-LINARRI,
 1027 $\Gamma \vdash \lambda x. \text{erase}(M^\bullet) : T \rightarrow_1 U$. By erasure, $\Gamma \vdash \text{erase}(\lambda_1 x : T.M^\bullet) : T \rightarrow_1 U$.
- 1028 ■ Rule A-UNARRI. Assume Γ^ω and $\Gamma \vdash \lambda_\omega x : T.M^\bullet \Rightarrow T \rightarrow_\omega U \mid \Gamma$. By rule A-UNARRI,
 1029 $\Gamma, x : (\varepsilon \vdash^0 T) \vdash M^\bullet \Rightarrow U \mid \Delta$ and $\Delta \div x = \Gamma$. By Lemma 16, we that $\mathcal{L}(\Delta) = \mathcal{L}(\Gamma)$, hence
 1030 Δ^ω . By the induction hypothesis, we have $\Gamma^\omega, x : (\varepsilon \vdash^0 T) \vdash \text{erase}(M^\bullet) : U$. By rule
 1031 T-UNARRI, $\Gamma^\omega \vdash \lambda x. \text{erase}(M^\bullet) : T \rightarrow_\omega U$. By erasure, $\Gamma \vdash \text{erase}(\lambda_\omega x : T.M^\bullet) : T \rightarrow_\omega U$.
- 1032 ■ Rule A-BOXI. Assume Γ^ω and $\Gamma \vdash \text{box}(\bar{x} : \bar{\tau}. M^\bullet) \Rightarrow \square(\bar{\tau} \vdash^n T) \mid \Gamma$. By A-BOXI,
 1033 $\Gamma \vdash \bar{x} : \bar{\tau}. M^\bullet \Rightarrow (\bar{\tau} \vdash^n T) \mid \Gamma$. The only rule that applies is A-CTX, hence $\Gamma_1, \bar{x} : \bar{\tau} \vdash$
 1034 $M^\bullet \Rightarrow T \mid \Delta$ with $\Gamma \xrightarrow{n} \Gamma_1^{\geq n}, \Gamma_2^{\leq n}$ and $\Gamma = \Delta \div \bar{x}, \Gamma_2$. It follows naturally that
 1035 $(\Delta \div \bar{x})^\omega$ and $(\Gamma_2)^\omega$. Furthermore, by Lemma 16, we have that $\mathcal{L}(\Delta) = \mathcal{L}(\Delta \div \bar{x})$, and
 1036 thus Δ^ω . By the induction hypothesis, $\Gamma_1^{\geq n}, \bar{x} : \bar{\tau}^{\leq n} \vdash \text{erase}(M^\bullet) : T$. By rule T-CTX,
 1037 $\Gamma_1^{\geq n} \vdash \bar{x}. \text{erase}(M^\bullet) : (\bar{\tau} \vdash^n T)$. By rule T-BoxI, $\Gamma_1, \Gamma_2 \vdash \text{box}(\bar{x}. \text{erase}(M^\bullet)) : \square(\bar{\tau} \vdash^n T)$.
 1038 By erasure, $\Gamma_1, \Gamma_2 \vdash \text{erase}(\text{box}(\bar{x} : \bar{\tau}. M^\bullet)) : \square(\bar{\tau} \vdash^n T)$.

1039 ■ Rule A-BoxE. Assume $(\Gamma_3 \div x)^\omega$ and $\Gamma_1 \vdash \text{let box } x = M^\bullet \text{ in } N^\bullet \Rightarrow T \mid \Gamma_3 \div x$. By
 1040 Lemma 16, we have that $\mathcal{L}(\Gamma_3) = \mathcal{L}(\Gamma_3 \div x)$, and thus $(\Gamma_3)^\omega$. By A-BoxE, we have
 1041 $\Gamma_1 \vdash M^\bullet \Rightarrow \Box\tau \mid \Gamma_2$ and $\Gamma_2, x: \tau \vdash N^\bullet \Rightarrow T \mid \Gamma_3$. By Lemma 18, we have that
 1042 $\Gamma_1 \setminus \mathcal{L}(\Gamma_2) \vdash M^\bullet \Rightarrow \Box\tau \mid \Gamma_2 \setminus \mathcal{L}(\Gamma_2)$. By the induction hypothesis, $\Gamma_1 \setminus \mathcal{L}(\Gamma_2) \vdash$
 1043 $\text{erase}(M^\bullet) : \Box\tau$ and $\Gamma_2, x: \tau \vdash \text{erase}(N^\bullet) : T$. By T-BoxE, $\Gamma_1 \setminus \mathcal{L}(\Gamma_2) \circ \Gamma_2 \vdash \text{let box } x =$
 1044 $\text{erase}(M^\bullet) \text{ in } \text{erase}(N^\bullet) : T$, and erasure, $\text{let box } x = \text{erase}(M^\bullet) \text{ in } \text{erase}(N^\bullet) =$
 1045 $\text{erase}(\text{let box } x = M^\bullet \text{ in } N^\bullet)$. By Lemma 17, $\mathcal{U}(\Gamma_1) = \mathcal{U}(\Gamma_2)$ and $\mathcal{L}(\Gamma_2) \subseteq \mathcal{L}(\Gamma_1)$.
 1046 By Lemma 7, we have that $\Gamma_1 \setminus \mathcal{L}(\Gamma_2) \circ \Gamma_2 = \Gamma_1$.
 1047

1048 **Proof of Theorem 6, Item 2.** The proof follows by induction on the rules of $\Gamma \vdash M : T$,
 1049 using Lemma 17 and Lemma 19. The proof for rule T-CONST, T-LINARRI and T-UNARRI
 1050 are trivial. The proofs for rules T-VAR, T-ARRE, T-PAIRI, T-PAIRE and T-MATCH follow the
 1051 same strategy as the proof for rule T-BoxE. For rule T-VAR, we must weaken the judgments
 1052 obtained from induction hypothesis from left to right. We detail the following cases:

1053 ■ Rule A-BoxI. Assume $\Gamma^\omega, \Gamma'^\omega \vdash \text{box } (\bar{x}.M) : \Box(\bar{\tau} \vdash^n T)$. By T-BoxI and T-CTX, we
 1054 have that $\Gamma^{\geq n}, \bar{x}: \bar{\tau}^{<n} \vdash M : T$. By the induction hypothesis, there is a M^\bullet such that
 1055 $\text{erase}(M^\bullet) = M$, and $\Gamma^{\geq n}, \bar{x}: \bar{\tau}^{<n} \vdash M^\bullet \Rightarrow T \mid \Delta$ and Δ^ω . Lets split the unused context
 1056 Γ' in the following way: $\Gamma' \xrightarrow{n} (\Gamma'_1)^{\geq n}, (\Gamma'_2)^{<n}$. Then we can apply A-CTX followed
 1057 by A-BoxI, to reach $\Gamma^{\geq n}, \Gamma'^{<n} \vdash \text{box } (\bar{x}: \bar{\tau}. M^\bullet) \Rightarrow \Box(\bar{\tau} \vdash^n T) \mid \Delta \div \bar{x}, (\Gamma'_2)^{<n}$. By
 1058 Lemma 19, we weaken the algorithmic typing judgment with $(\Gamma'_1)^{\geq n}$. Thus, we arrive at
 1059 $\Gamma, \Gamma' \vdash \text{box } (\bar{x}. M^\bullet) \Rightarrow \Box(\bar{\tau} \vdash^n T) \mid (\Delta \div \bar{x}), \Gamma'$. It's easy to check that $((\Delta \div \bar{x}), \Gamma')^\omega$.
 1060 ■ Rule T-BoxE. Assume $\Gamma \circ \Delta \vdash \text{let box } x = M \text{ in } N : T$. Induction on the premises gives
 1061 us $\Gamma \vdash M^\bullet \Rightarrow \Box\tau \mid \Gamma'$ with $(\Gamma')^\omega$ and $\text{erase}(M^\bullet) = M$, and $\Delta, x: \tau \vdash N^\bullet \Rightarrow T \mid \Delta'$
 1062 with $(\Delta')^\omega$ and $\text{erase}(N^\bullet) = N$. Since $\Gamma \cup \mathcal{L}(\Delta) = \Gamma \circ \Delta$, we weaken (Lemma 19)
 1063 $\Gamma \vdash M^\bullet \Rightarrow \Box\tau \mid \Gamma'$, resulting in $\Gamma \circ \Delta \vdash M^\bullet \Rightarrow \Box\tau \mid \Gamma' \cup \mathcal{L}(\Delta)$. We have to show that
 1064 $\Gamma' \cup \mathcal{L}(\Delta) = \Delta$. Since $(\Gamma')^\omega$, then $\mathcal{L}(\Gamma' \cup \mathcal{L}(\Delta)) = \mathcal{L}(\Delta)$. By Lemma 17, we have that
 1065 $\mathcal{U}(\Gamma) = \mathcal{U}(\Gamma')$. Therefore, $\mathcal{U}(\Gamma' \cup \mathcal{L}(\Delta)) = \mathcal{U}(\Gamma') = \mathcal{U}(\Gamma) = \mathcal{U}(\Gamma \circ \Delta) = \mathcal{U}(\Delta)$. Therefore,
 1066 $\Gamma' \cup \mathcal{L}(\Delta) = \Delta$. By A-BoxE, we have $\Gamma \circ \Delta \vdash \text{let box } x = M^\bullet \text{ in } N^\bullet \Rightarrow T \mid \Delta' \div x$. Since
 1067 $(\Delta')^\omega$, by Lemma 16, $(\Delta' \div x)^\omega$.
 1068