Gradual Session Types*

ATSUSHI IGARASHI, Kyoto University, Japan
PETER THIEMANN, University of Freiburg, Germany
VASCO T. VASCONCELOS, LaSIGE, Faculty of Sciences, University of Lisbon, Portugal
PHILIP WADLER, University of Edinburgh, Scotland

Session types are a rich type discipline, based on linear types, that lifts the sort of safety claims that come with type systems to communications. However, web-based applications and microservices are often written in a mix of languages, with type disciplines in a spectrum between static and dynamic typing. Gradual session types address this mixed setting by providing a framework which grants seamless transition between statically typed handling of sessions and any required degree of dynamic typing.

We propose Gradual GV as an extension of the functional session type system GV with dynamic types and casts. We demonstrate type and communication safety as well as blame safety, thus extending previous results to functional languages with session-based communication. The interplay of linearity and dynamic types requires a novel approach to specifying the dynamics of the language.

CCS Concepts: • **Theory of computation → Type structures**; *Operational semantics*; • **Software and its engineering → Functional languages**; **Concurrent programming structures**;

Additional Key Words and Phrases: session types, gradual types, semantics, type checking

ACM Reference Format:
Atsushi Igarashi, Peter Thiemann, Vasco T. Vasconcelos, and Philip Wadler. 2017. Gradual Session Types. *Proc. ACM Program. Lang.* 1, ICFP, Article 38 (September 2017), 28 pages.
https://doi.org/10.1145/3110282

1 INTRODUCTION

It was the best of types, it was the worst of types.

Type systems are retreating from areas they used to dominate, with statically-typed languages such as Java seen as old hat while dynamically-typed languages such as Javascript, Python, and R are on the rise. Simultaneously, type systems are expanding into new domains, such as dependent types in Agda, Coq, and Idris, effect types in Eff, Frank, and Koka, and session types in Links, Scribble, and Singularity OS.

Gradually-typed languages reconcile these two trends. They permit one to assemble programs with some components written in a statically-typed language and some in a dynamically-typed language. Gradually-typed languages have been widely explored in both theory and practice, beginning with contracts in Racket and continuing with Microsoft's TypeScript and Facebook's Hack and Flow. Many systems, such as Racket and TypeScript TPD, rely on contracts or similar constructs to ensure that dynamically-typed values adhere to statically-typed constraints when values pass from one world to the other.

*Artifact available at https://doi.org/10.6094/UNIFR/12757

2475-1421/2017/9-ART38
https://doi.org/10.1145/3110282

At first blush, one might consider gradual types as largely a response to the former trend: they provide a way for those poor schmucks using dynamically-typed languages to evolve their code toward statically-typed languages that are easier to maintain. Exactly such a view underlies the STOP (Scripts to Programs) series of workshops. But on second thought, one might consider gradual types as even more helpful in light of the latter trend. When it comes to dependent types, effect types, or session types the fraction of programmers working in languages that offer static checking for such types is essentially zero. We are all poor schmucks who can benefit from a way to evolve toward languages with more precise type systems.

Hence, an important line of research is to extend gradual typing so that it not only relates statically-typed and dynamically-typed languages, but also relates less-precisely-typed and more-precisely-typed languages. There is a small body of research on doing so for dependent types and effect types, which we review in the section on related work. This paper presents what we believe is the first system that extends gradual typing to session types.

Session types were introduced by Honda [1993], drawing on the π-calculus of Milner et al. [1992] and the linear logic of Girard [1987], and further developed by Honda et al. [1998] and Yoshida and Vasconcelos [2007], among many others. Subtyping for session types was introduced by Gay and Hole [2005], and session types were embedded into a functional language with linear types, similar to the one used in this paper, by Gay and Vasconcelos [2010]. Propositions-as-types interpretations of session types in linear logic were introduced by Caires and Pfenning [2010]; Caires et al. [2014] and Wadler [2012, 2014]. One important line of research is multiparty session types, introduced by Honda et al. [2008, 2016], but we confine our attention here to dyadic session types.

Session types have been adapted to a variety of languages, either statically or dynamically checked, and using either libraries or additions to the toolchain; implementations include C, Erlang, Go, Haskell, Java, Python, Rust, and Scala. New languages incorporating session types include C0, Links, SePi, SILL, and Singularity. Industrial uses of session types include: Red Hat's support of the Scribble specification language, which has been used as a common interface for several systems based on session types; Thoughtworks's use of session types to manage microservices; and the Ocean Observatories Initiative's use of dynamically-checked session types in Python. Session types inspired an entire line of research on what has come to be called behavioral types, the subject of EU COST action BETTY, a recent Shonan meeting, and a recent Dagstuhl seminar.

Here is a simple session type encoding of a protocol to purchase an online video:

$$S_{video} = !\mathsf{string}.\,?\mathsf{int}.\,\oplus\{buy : !\mathsf{CC}.\,?\mathsf{URL}.\,\mathsf{end}_?, quit : \mathsf{end}_!\}.$$

It describes a channel endpoint along which a client sends the name of a video as a string, receives its cost as an integer, and then selects either to buy the video, in which case one sends a credit card number, receives a URL from which the video may be downloaded, and waits for an indication that the channel has been closed, or selects to quit and closes the channel. There is a dual session type for server at the other end of the channel, where ! (write) is swapped with ? (read), $\oplus$ (select from a choice) is swapped with & (offer a choice), and $\mathsf{end}_!$ (close a channel) is swappend with $\mathsf{end}_?$ (wait for a channel to close).

Session types are necessarily linear. Let x be bound to a string and let c be bound to a channel endpoint of type S_{video}. Performing

$$\mathsf{let}\ d = \mathsf{send}\ x\ c\ \mathsf{in}\ \ldots$$

binds d to a channel endpoint of type R, where $S_{video} = !\mathsf{string}.R$. To avoid sending a string to the same channel twice, it is essential that c must be bound to the only reference to the channel endpoint before the operation, and for similar reasons d must be bound to the only reference to the channel endpoint after. Such restrictions can easily be enforced in a statically-typed language with

an affine type discipline. Linearity is required to guarantee that channels are not abandoned before they are closed.

But how is one to ensure linearity in a dynamically-typed language? Following Tov and Pucella [2010], we require that each dynamically-typed reference to a channel endpoint is equipped with a lock, and after the channel is used once the reference is locked to ensure it cannot be used again. To ensure that each channel is appropriately terminated, with either a wait or a close operation, garbage collection flags an error if a dynamically-typed reference to a channel becomes inaccessible.

Our system is the first to integrate static and dynamic session types. It preserves the safety properties of statically-typed sessions, namely progress, preservation, and absence of run-time errors. The latter includes session fidelity: every send is matched with a receive, every select is matched with an offer, and every wait is matched with close. Many, but not all, systems with session types support recursive session types, and many, but not all, systems with session types ensure deadlock freedom; we leave such developments for future work.

We give our system a compact formulation along the lines of the blame calculus of Wadler and Findler [2009], based on the notion of a *cast* to mediate interactions between more-precisely typed (e.g., statically typed) and less-precisely typed (e.g., dynamically typed) components of a program. We define the usual four subtyping relations, ordinary, positive, negative, and naive, and show the usual results, including a tangram theorem relating the four forms of subtyping and blame safety. A corollary of our results is that in any interaction between more-precisely typed and less-precisely typed components of a program, any cast error is due to the less-precisely typed component.

Our paper makes the following contributions.

- Section 2 provides an overview of the novel techniques in our work, and how we dynamically enforce linearity and session types.
- Section 3 describes a complete formal calculus, including syntax, reduction rules, typing rules, and embedding of a dynamically typed language with channel-based communication into our calculus.
- Section 4 presents standard results for our calculus, including progress and preservation, session fidelity, the tangram theorem, and blame safety.

Section 5 describes related work and Section 6 concludes.

2 MOTIVATION

Sy and Rob are programmers, who work together on a project that relies on microservices. Sy is a strong advocate of static typing and relies on an implementation language that supports session types out of the box. Rob, on the other hand, is a strong advocate of dynamically typed languages. One of the credos of microservices architectures is that the implementation of a service endpoint is language-agnostic, which means it can be implemented in any programming language whatsoever as long as it adheres to its protocol. However, Sy does not want to compromise the strong guarantees (e.g., type safety, session fidelity) of the statically typed code by communicating with Rob's client. Rob is also keen on having strong guarantees, but does not mind if they are enforced at run time. Here is the story how they can collaborate safely using Gradual GV, our proposal for a blame calculus underlying a gradually typed functional language with synchronous binary session types.

2.1 A Compute Service

The compute service is one of the "hello world" programs of the session-type community and it is a simplified version of one of the protocols in Sy and Rob's project. This service involves two peers, a server and a client, connected via a communication link. The server runs a protocol that first offers a choice of two arithmetic operations, negation or addition, then reads one or two numbers

depending on the operation, outputs the result of applying the selected operation to its operand(s), and finally closes the connection. The client chooses an operation by sending the server a label, which is either *neg* or *add* indicating the choice of negation or addition, respectively. In session-type notation, the compute protocol from the server's point of view reads as follows.

$$\mathit{Compute} = \&\{\mathit{neg} : ?\mathsf{int}.!\mathsf{int}.\mathsf{end}_!, \mathit{add} : ?\mathsf{int}.?\mathsf{int}.!\mathsf{int}.\mathsf{end}_!\}$$

Sy chose to implement the server in the language GV that is inspired by previous work [Gay and Vasconcelos 2010] and that we will describe formally in Section 3.

```
computeServer :: Compute → unit
computeServer c =
  case c of {
    neg:  c. let (v1, c) = receive c in
             let c = send (-v1) c in
             close c;
    add:  c. let (v1, c) = receive c in
             let (v2, c) = receive c in
             let c = send (v1+v2) c in
             close c
  }
```

The parameter c of type *Compute* is the server's endpoint of the communication link to the client (when unambiguous, we often just say endpoint or channel). The **case** c **of** ... expression receives the client's choice on channel c in the form of a label *neg* or *add* and branches accordingly. The notation "c. " in each branch (re-)binds the variable c to the channel in the state after the transmission has happened and with a session type corresponding to the respective branch in the *Compute* type. The **receive** c operation receives a value on channel c and returns a pair of the received value and the depleted channel. Analogously, the **send** v c operation sends value v on channel c and returns the depleted channel. The final **close** c disconnects the communication link by closing the channel.

2.2 The View from the Client Side

A client of the *Compute* protocol communicates on a channel with the protocol *ComputeD*. This protocol is the dual of *Compute* which (roughly) swaps all sending and receiving operations.

$$\mathit{ComputeD} = \oplus\{\mathit{neg} : !\mathsf{int}.?\mathsf{int}.\mathsf{end}_?, \mathit{add} : !\mathsf{int}.!\mathsf{int}.?\mathsf{int}.\mathsf{end}_?\}$$

Many clients for this simple, finite protocol exhibit a simpler supertype of *ComputeD* because they select a single operation and then proceed linearly along the selected branch of the protocol.

$$\mathit{ComputeDneg} = \oplus\{\mathit{neg} : !\mathsf{int}.?\mathsf{int}.\mathsf{end}_?\}$$

Here is Sy's implementation of a typed client for *ComputeDneg*.

```
negationClient :: int → ComputeDneg → int
negationClient x c =
  let c = select neg c in
  let c = send x c in
  let (y, c) = receive c in
  let _ = wait c in
  y
```

There are two new operations in the client code. The **select** neg c operation selects the *neg* branch in the protocol by sending the *neg* label to the server. It returns a channel to run the selected branch of the protocol with type !int.?int.end$_?$. The **wait** c operation matches the **close** operation on the server and disconnects the client.

2.3 A Unityped Server

To test some new features, Rob also implements the *Compute* protocol, but does so in the unityped language Uni GV, which is strongly typed like Racket or Python but does not impose any static typing discipline. Here is Rob's implementation of the server.

```
-- unityped
dynServer c =
  case c of {
    neg: c. serveOp 1 (λx.-x) c;
    add: c. serveOp 2 (λx.λy.x+y) c
  }

serveOp n op c =
  if n==0 then
    close (send op c)
  else
    let (v,c) = receive c in
    serveOp (n-1) (op v) c
```

The main function dynServer receives the client's selection and delegates to an auxiliary function serveOp that takes the arity of a function, the function itself, and the channel end on which to receive the arguments and to send the result. The serveOp function counts down the number of remaining function applications in the first argument, accumulates partial function applications in the second argument, and propagates the channel end in the third argument.

It is easy to see that the dynServer function implements the *Compute* protocol. However, a dependent type system would be required to describe the type of the serveOp function adequately and we are not aware of an existing session-type system that would be able to give a type to dynServer.

Rob chose this style of implementation because it is amenable to experimentation with protocol extensions: the function dynServer is trivially extensible to new operations and types by adding new lines to the **case** dispatch.

2.4 The Gradual Way

How can we embed Rob's server with other program fragments in the typed language (e.g., Sy's client) while retaining as many typing guarantees as possible? Questions like this have often been answered satisfactorily by resorting to gradual typing and a gradual system works in this case, too.

Let's imagine that the function dynServer is written in a gradually typed surface language analogous to the gradually typed lambda calculus GTLC of Siek et al. [2015], but extended with GV's communication operations. Such a surface language comes with a translation into a blame calculus with explicit casts. This translation makes just those parts of the code dynamic that are necessary to make typing of the code go through. That is, it inserts casts to ⋆, the dynamic type, to resolve potential type clashes and it inserts casts from ⋆ wherever an elimination construct requires a type of a certain form. Here is the output of such a (hypothetical) translation:

```
flexiServer :: Compute → unit
```

```
flexiServer c =
  case c of {
    neg: c. fserve 1 ((λx.−x) : int → int ⇒ ⋆)
                     (c : ?int.!int.end! ⇒ ⊛);
    add: c. fserve 2 ((λx.λy.x+y) : int → int → int ⇒ ⋆)
                     (c : ?int.?int.!int.end! ⇒ ⊛)
  }

fserve :: int → ⋆ → ⊛ → unit
fserve n op c =
  if n==0 then
    close (send op (c : ⊛ ⇒ !⋆.end! ))
  else
    let (v,c) = receive (c : ⊛ ⇒ ?⋆.⊛) in
    fserve (n−1) ((op : ⋆ ⇒ ⋆ → ⋆) v) c
```

The first argument n of fserve is consistently handled as an integer, so its type is **int**. The second argument op is invoked with values of type **int** →**int** →**int**, **int** →**int**, and **int**: the gradual type checker subsumes these types to ⋆ and inserts the respective casts in the code of flexiServer , just like in the translation from GTLC. The third argument c is invoked with channels of different types: ?**int**.?**int**.!**int**.**end**!, ?**int**.!**int**.**end**!, and !**int**.**end**!. These types are subsumed to a type that is novel to this work, the *dynamic session type*, ⊛, a *linear type* which subsumes all session types. The resulting casts in flexiServer look fairly involved, but we should keep in mind that the programmer does **not** have to write them as they result from the translation.

It is important to see that the channel c is handled linearly in functions flexiServer and fserve. For that reason, the role and handling of the linear dynamic session type with respect to the set of session types is analogous to the role and handling of ⋆ with respect to general types, as has been shown in earlier work [Fennell and Thiemann 2012; Thiemann 2014].

2.5 Dynamic Linearity

Siek et al. [2015] postulate that a gradual type system should come with a full embedding of a unityped calculus. This embedding (which we indicate by ceiling brackets ⌈...⌉) extends the embedding given by Wadler and Findler [2009] to handle the operations on sessions (see Figure 8 for its definition).

If Rob's source code for dynServer is not available, then Sy cannot translate it with the gradual type checker as in Section 2.4. Instead, Sy must invoke the fully embedded version of the dynServer function as follows:

```
callDynServer :: Compute → unit
callDynServer c =
  (dynServer : ⋆ ⇒ Compute → unit) c
```

The function callDynServer accepts a channel of type *Compute*, then it casts the embedded dynServer (of type ⋆) to the function type Computer →**unit**, and applies the resulting function to c.

To make the code more perspicuous, Sy could anticipate a couple of standard reductions of the blame calculus [Wadler and Findler 2009], which are also part of Gradual GV, and write

```
callDynServer :: Compute → unit
callDynServer c =
  (dynServer : ⋆ ⇒ ⋆ → ⋆) (c : Compute ⇒ ⋆) : ⋆ ⇒ unit
```

The casts in this code makes it completely obvious that Sy expects dynServer to be a function and further expects c to be used as a channel of type *Compute*. Any misuse will allocate blame to the respective cast in dynServer. (We omit blame labels in the examples for simplicity.)

One kind of misuse that we have not discussed, yet, is compromising linearity: Sy has no guarantee that Rob's code does not accidentally duplicate or drop the communication channel. Gradual GV takes care of linearity by factoring the cast $(\mathsf{c} : \mathsf{Compute} \Rightarrow \star)$ through the dynamic session type $\circledast$:

$$((\mathsf{c} : \mathsf{Compute} \Rightarrow \circledast) : \circledast \Rightarrow \star)$$

The first part is a cast among linear (session) types and it can be handled as outlined in Section 2.4. The second part is a cast from a *linear type* (which could be a session type, a linear function type, or a linear product) to the *unrestricted dynamic type* $\star$.

A cast from a linear type to unrestricted $\star$ is a novelty of our system Gradual GV. Operationally, the cast introduces an indirection through a store: it takes a linear value as an argument, allocates a new cell in the store, moves the linear value along with a representation of its type into the cell, and returns a handle a to the cell as an unrestricted value of type $\star$. Gradual GV represents the cell by a process and creates handles by introducing an appropriate binder so that a process of the form $E[v : \circledast \Rightarrow \star]$ reduces to $(\nu a)(E[a] \mid a \mapsto v : \circledast \Rightarrow \star)$. Linear use of this cell is controlled at run time using ideas from existing techniques for run-time monitoring of affine types [Padovani 2017; Tov and Pucella 2010].

Any access to a cell comes in the guise of a cast $a : \star \Rightarrow T$ from $\star$ to another type applied to a handle a. If the first access to the cell is a cast from $\star$ to a linear type consistent with the type representation stored in the cell, then the cast returns the linear value and empties the cell. Any subsequent access to the same cell results in a linearity violation which allocates blame to the label on the cast from $\star$. If the first cast attempts to convert to an inconsistent linear or non-linear type, then blame is allocated to that cast already. In addition, there is a garbage collection rule that fires when the handle of a full cell is no longer reachable from any process. It allocates blame to the context of the cast to $\star$ because that cast violated the linearity protocol by dismissing the handle.

2.6 End-to-end Dynamicity

The discussion so far ignores the issue of channel creation. In fact, the example code tacitly assumes that channels are created with a fully specified session type that provides a "ground truth". Later on, channels may be cast to $\circledast$ and on to $\star$, but essentially they adhere to the ground truth established at their creation.

Unfortunately, this view cannot be upheld in a calculus that is able to embed a unityped language like Uni GV. When writing **new** in a unityped program to create a channel, Rob (hopefully) has some session type in mind, but it is not manifest in the code.

In the typed setting, **new** returns a linear pair of session endpoints of type $S \times_{\mathsf{lin}} \overline{S}$ where S is the server session type and $\overline{S}$ its dual client counterpart. When embedding the unityped **new**, the session type S is unknown. Hence, the embedding needs to create a channel without an inherent ground truth session type. It does so by assigning both channel ends type $\circledast$, which is considered a self-dual session type, and casting it to $\star$ as in $\mathbf{new} : \circledast \times_{\mathsf{lin}} \circledast \Rightarrow \star$. Gradual GV cannot offer any static guarantees for either end of such a channel.

To see what run-time guarantees Gradual GV can offer for such a channel, let's consider the embedding of the dynamic send and receive operations that may be applied to it. The embedded send operation takes two arguments of type $\star$, for the value and the channel, and returns the updated channel wrapped in type $\star$. The embedded receive operation takes a wrapped channel of

type $\star$ and returns a ($\star$-wrapped) pair of the received value and the updated channel.

$$\lceil \mathsf{send}\, e\, f \rceil = (\mathsf{send}\, \lceil e \rceil\, (\lceil f \rceil : \star \overset{p}{\Rightarrow} !\star.\circledast)) : \circledast \Rightarrow \star$$

$$\lceil \mathsf{receive}\, e \rceil = (\mathsf{receive}\, (\lceil e \rceil : \star \overset{q}{\Rightarrow} ?\star.\circledast)) : \star \times_{\mathsf{lin}} \circledast \Rightarrow \star$$

Now consider running the following unityped program with entry point main.

```
client cc =
  let (v, cc) = receive cc in wait cc
server cs =
  let cs = send 42 cs in close cs
main =
  let (cs,cc) = new in
  let _ = fork (client cc) in
  server cs
```

After a few computation steps, it reaches a configuration where client and server have reduced to

$$E[(\mathsf{receive}\, (cc : \circledast \overset{q}{\Rightarrow} ?\star.\circledast)) : \star \times_{\mathsf{lin}} \circledast \Rightarrow \star] \;\mid\; F[(\mathsf{send}\, (42 : \mathsf{int} \Rightarrow \star)\, (cs : \circledast \overset{p}{\Rightarrow} !\star.\circledast)) : \circledast \Rightarrow \star]$$

for some contexts E and F, where $cc : \circledast$ and $cs : \circledast$ are the two ends of the channel. Our calculus recognizes that the two processes use the channel consistently as the cast target on one end $?\star.\circledast$ is dual to the cast target $!\star.\circledast$ at the other end, drops the casts at both ends, and retypes the ends to $cc : ?\star.\circledast$ and $cs : !\star.\circledast$, respectively.

$$E[(\mathsf{receive}\, cc) : \star \times_{\mathsf{lin}} \circledast \Rightarrow \star] \;\mid\; F[(\mathsf{send}\, (42 : \mathsf{int} \Rightarrow \star)\, cs) : \circledast \Rightarrow \star]$$

Implementing this reduction requires communication between the two processes to check the cast targets for consistency. While our formal presentation abstracts over this implementation issue, we observe that a simple *asynchronous* message exchange is sufficient: Each cast first sends its target type and then receives the target type of the cast at the other end. Then both processes locally check whether the target types are duals of one another. If they are, then both processes continue; otherwise they allocate blame. As both ends compare the same types, the outcome of the check is the same in both processes.

3 GV AND GRADUAL GV

3.1 GV

We begin by discussing a language GV with session types but without gradual types. GV is inspired by both the language introduced by Gay and Vasconcelos [2010] and the 'good variant' of the language described by Wadler [2012, 2014]. Unlike the latter, we do not guarantee deadlock freedom.

3.1.1 Types and subtyping. Figure 1 summarises types of GV. Let m, n range over *multiplicities* for types whose use is either unrestricted, un, or must be linear, lin.

Let T, U range over types, which include: unit type, unit; unrestricted and linear function types, $T \to_m U$; unrestricted and linear product types, $T \times_m U$; and session types. One might also wish to include booleans or base types, but we omit these as they can be dealt with analogously to unit.

Let l range over labels used for selection and case choices. Let S, R range over session types that describe communication protocols for channel endpoints, which include: send $!T.S$, to send a value of type T and then behave as S; receive $?T.S$, to receive a value of type T and then behave as S, select $\oplus\{l_i : S_i\}_{i \in I}$, to send one of the labels l_i and then behave as S_i; case $\&\{l_i : S_i\}_{i \in I}$ to receive any of the labels l_i and then behave as S_i; close $\mathsf{end}_!$, to close a channel endpoint; and wait $\mathsf{end}_?$,

$$
\begin{array}{lll}
\text{Multiplicities} & m, n & ::= \mathsf{lin} \mid \mathsf{un} \\
\text{Types} & T, U & ::= \mathsf{unit} \mid S \mid T \to_m U \mid T \times_m U \\
\text{Session types} & S, R & ::= !T.S \mid ?T.S \mid \oplus\{l_i : S_i\}_{i \in I} \mid \&\{l_i : S_i\}_{i \in I} \mid \mathsf{end}_! \mid \mathsf{end}_?
\end{array}
$$

Duality $\boxed{\overline{S} = R}$

$$
\overline{!T.S} = ?T.\overline{S} \qquad \overline{\oplus\{l_i : S_i\}_{i \in I}} = \&\{l_i : \overline{S}_i\}_{i \in I} \qquad \overline{\mathsf{end}_!} = \mathsf{end}_?
$$

$$
\overline{?T.S} = !T.\overline{S} \qquad \overline{\&\{l_i : S_i\}_{i \in I}} = \oplus\{l_i : \overline{S}_i\}_{i \in I} \qquad \overline{\mathsf{end}_?} = \mathsf{end}_!
$$

Multiplicity ordering $\boxed{m <: n}$

$$
\mathsf{un} <: \mathsf{un} \qquad \mathsf{un} <: \mathsf{lin} \qquad \mathsf{lin} <: \mathsf{lin}
$$

Multiplicity of a type $\boxed{m(T) \quad n^{:>}(T)}$

$$
\mathsf{un}(\mathsf{unit}) \qquad \mathsf{lin}(S) \qquad m(T \times_m U) \qquad m(T \to_m U) \qquad \frac{m(T) \quad m <: n}{n^{:>}(T)}
$$

Subtyping $\boxed{T <: U}$

$$
\mathsf{unit} <: \mathsf{unit} \qquad \frac{T' <: T \quad U <: U' \quad m <: n}{T \to_m U <: T' \to_n U'} \qquad \frac{T <: T' \quad U <: U' \quad m <: n}{T \times_m U <: T' \times_n U'}
$$

$$
\frac{T' <: T \quad S <: S'}{!T.S <: !T'.S'} \qquad \frac{T <: T' \quad S <: S'}{?T.S <: ?T'.S'} \qquad \frac{J \subseteq I \quad (S_j <: R_j)_{j \in J}}{\oplus\{l_i : S_i\}_{i \in I} <: \oplus\{l_j : R_j\}_{j \in J}} \qquad \frac{I \subseteq J \quad (S_i <: R_i)_{i \in I}}{\&\{l_i : S_i\}_{i \in I} <: \&\{l_j : R_j\}_{i \in J}}
$$

$$
\mathsf{end}_! <: \mathsf{end}_! \qquad \mathsf{end}_? <: \mathsf{end}_?
$$

Fig. 1. Types and subtyping in GV

to wait for the other end of the channel to close. We will call the session type that describes the behaviour after send, receive, select, or case the *residual*.

We define the usual notion of the dual of a session type S, written as $\overline{S}$. Send is dual to receive, select is dual to case, and close is dual to wait. Duality is an involution, so that $\overline{\overline{S}} = S$.

Multiplicities are ordered by $\mathsf{un} <: \mathsf{lin}$, indicating that an unrestricted value may be used where a linear value is expected, but not conversely. The unit type is unrestricted, session types are linear, while function types $T \to_m U$ and product types $T \times_m U$ are unrestricted or linear depending on the multiplicity m that decorates the type constructor. We also write $n^{:>}(T)$ if $m(T)$ holds for some m such that $m <: n$, thus $\mathsf{un}^{:>}(T)$ holds only if $\mathsf{un}(T)$, while $\mathsf{lin}^{:>}(T)$ holds if either $\mathsf{lin}(T)$ or $\mathsf{un}(T)$, and hence holds for any type.

We define subtyping as usual for functional-program like systems [Gay and Vasconcelos 2010]. Function types are contravariant in their domain and covariant in their range, and send types are contravariant in the value sent and covariant in the residual session type. All other types and session types are covariant in all components. Width subtyping resembles record subtyping for select, and variant subtyping for case. That is, on an endpoint where one may select among labels with an index in I one may instead select among labels with indexes in J, so long as $J \subseteq I$, while on an endpoint where one must be able to receive any label with an index in I one may instead

$$
\begin{array}{lll}
\text{Names} & & z ::= x \mid c \\
\text{Expressions} & & e, f ::= z \mid () \mid \lambda_m x.e \mid e\ f \mid (e, f)_m \mid \mathsf{let}\, x, y = e\, \mathsf{in}\, f \mid \mathsf{fork}\, e \\
& & \quad \mid \mathsf{new} \mid \mathsf{send}\, e\, f \mid \mathsf{receive}\, e \mid \mathsf{select}\, l\, e \mid \mathsf{case}\, e\, \mathsf{of}\, \{l_i : x_i.e_i\}_{i \in I} \\
& & \quad \mid \mathsf{close}\, e \mid \mathsf{wait}\, e \\
\text{Processes} & & P, Q ::= \langle e \rangle \mid (P \mid Q) \mid (\nu c, d)P \\
\text{Type environments} & & \Gamma, \Delta ::= \cdot \mid \Gamma, z : T
\end{array}
$$

Environment splitting $\boxed{\Gamma = \Gamma_1 \circ \Gamma_2}$

$$
\cdot = \cdot \circ \cdot \qquad \frac{\Gamma = \Gamma_1 \circ \Gamma_2 \quad \mathsf{un}(T)}{\Gamma, z : T = (\Gamma_1, z : T) \circ (\Gamma_2, z : T)} \qquad \frac{\Gamma = \Gamma_1 \circ \Gamma_2 \quad \mathsf{lin}(T)}{\Gamma, z : T = (\Gamma_1, z : T) \circ \Gamma_2} \qquad \frac{\Gamma = \Gamma_1 \circ \Gamma_2 \quad \mathsf{lin}(T)}{\Gamma, z : T = \Gamma_1 \circ (\Gamma_2, z : T)}
$$

Typing expressions $\boxed{\Gamma \vdash e : T}$

$$
\frac{\mathsf{un}(\Gamma)}{\Gamma, z : T \vdash z : T} \qquad \frac{\mathsf{un}(\Gamma)}{\Gamma \vdash () : \mathsf{unit}} \qquad \frac{\Gamma, x : T \vdash e : U \quad m :> (\Gamma)}{\Gamma \vdash \lambda_m x.e : T \to_m U} \qquad \frac{\Gamma \vdash e : T \to_m U \quad \Delta \vdash f : T}{\Gamma \circ \Delta \vdash e\ f : U}
$$

$$
\frac{\Gamma \vdash e : T \quad \Delta \vdash f : U \quad m :> (T) \quad m :> (U)}{\Gamma \circ \Delta \vdash (e, f)_m : T \times_m U} \qquad \frac{\Gamma \vdash e : T \times_m U \quad \Delta, x : T, y : U \vdash f : V}{\Gamma \circ \Delta \vdash \mathsf{let}\, x, y = e\, \mathsf{in}\, f : V}
$$

$$
\frac{\Gamma \vdash e : \mathsf{unit}}{\Gamma \vdash \mathsf{fork}\, e : \mathsf{unit}} \qquad \frac{\mathsf{un}(\Gamma)}{\Gamma \vdash \mathsf{new} : S \times_{\mathsf{lin}} \overline{S}} \qquad \frac{\Gamma \vdash e : T \quad \Delta \vdash f : !T.S}{\Gamma \vdash \mathsf{send}\, e\, f : S} \qquad \frac{\Gamma \vdash e : ?T.S}{\Gamma \vdash \mathsf{receive}\, e : T \times_{\mathsf{lin}} S}
$$

$$
\frac{\Gamma \vdash e : \oplus\{l_i : S_i\}_{i \in I} \quad j \in I}{\Gamma \vdash \mathsf{select}\, l_j\, e : S_j} \qquad \frac{\Gamma \vdash e : \&\{l_i : S_i\}_{i \in I} \quad (\Delta, x_i : S_i \vdash e_i : T)_{i \in I}}{\Gamma \circ \Delta \vdash \mathsf{case}\, e\, \mathsf{of}\, \{l_i : x_i.e_i\}_{i \in I} : T}
$$

$$
\frac{\Gamma \vdash e : \mathsf{end}_!}{\Gamma \vdash \mathsf{close}\, e : \mathsf{unit}} \qquad \frac{\Gamma \vdash e : \mathsf{end}_?}{\Gamma \vdash \mathsf{wait}\, e : \mathsf{unit}} \qquad \frac{\Gamma \vdash e : T \quad T <: U}{\Gamma \vdash e : U}
$$

Typing processes $\boxed{\Gamma \vdash P}$

$$
\frac{\Gamma \vdash e : T \quad \mathsf{un}(T)}{\Gamma \vdash \langle e \rangle} \qquad \frac{\Gamma \vdash P \quad \Delta \vdash Q}{\Gamma \circ \Delta \vdash P \mid Q} \qquad \frac{\Gamma, c : S, d : \overline{S} \vdash P}{\Gamma \vdash (\nu c, d)P}
$$

Fig. 2. Expressions, processes, and typing in GV

receive any label with an index in J, so long as $I \subseteq J$. (Beware that the subtyping on endpoints is exactly the reverse for process-calculus like systems, such as the CP of Wadler [2012, 2014]!)

Subtyping is reflexive, transitive, and antisymmetric. Duality inverts subtyping, in that $S <: R$ if and only if $\overline{R} <: \overline{S}$.

3.1.2 Expressions, processes, and typing. Expressions, processes, and typing for GV are summarised in Figure 2. We let x, y range over variables, c, d range over channel endpoints, and z range over names, which are either variables or channel endpoints.

We let e, f range over expressions, which include names, unit value, function abstraction and application, pair creation and destruction, fork a process, create a new pair of channel endpoints, send, receive, select, case, close, and wait. Function abstraction and pair creation are labeled with the multiplicity of the value created. A GV program is always given as an expression, but as it executes it may fork new processes.

$$
\begin{array}{lll}
\text{Values} & v, w & ::= () \mid \lambda_m x.e \mid (v, w)_m \mid c \\
\text{Eval contexts} & E, F & ::= [\,] \mid E\,e \mid v\,E \mid (E, e)_m \mid (v, E)_m \mid \mathsf{let}\, x, y = E \,\mathsf{in}\, e \mid \mathsf{send}\, E\, e \mid \mathsf{send}\, v\, E \\
& & \quad \mid \mathsf{receive}\, E \mid \mathsf{select}\, l\, E \mid \mathsf{case}\, E \,\mathsf{of}\, \{l_i : x_i.e_i\}_{i \in I} \mid \mathsf{close}\, E \mid \mathsf{wait}\, E
\end{array}
$$

Expression reduction $\boxed{e \longrightarrow f}$

$$
\begin{array}{c}
(\lambda_m x.e)v \longrightarrow e[v/x] \\
\mathsf{let}\, x, y = (u, v)_m \,\mathsf{in}\, e \longrightarrow e[u/x][v/y]
\end{array}
$$

Structural congruence $\boxed{P \equiv Q}$

$$
\begin{array}{ccc}
P \mid Q \equiv Q \mid P & P \mid (Q \mid P') \equiv (P \mid Q) \mid P' & P \mid \langle () \rangle \equiv P \\
((vc, d)P) \mid Q \equiv (vc, d)(P \mid Q) & (vc, d)P \equiv (vd, c)P & (vc, d)(vc', d')P \equiv (vc', d')(vc, d)P
\end{array}
$$

Process reduction $\boxed{P \longrightarrow Q}$

$$
\begin{array}{rl}
\langle E[\mathsf{fork}\, e] \rangle & \longrightarrow \langle E[()] \rangle \mid \langle e \rangle \\
\langle E[\mathsf{new}] \rangle & \longrightarrow (vc, d)\langle E[(c, d)_{\mathsf{lin}}] \rangle \\
(vc, d)(E[\langle \mathsf{send}\, v\, c \rangle] \mid \langle F[\mathsf{receive}\, d] \rangle) & \longrightarrow (vc, d)(\langle E[c] \rangle \mid \langle F[(v, d)_{\mathsf{lin}}] \rangle) \\
(vc, d)(\langle E[\mathsf{select}\, l_j\, c] \rangle \mid \langle F[\mathsf{case}\, d \,\mathsf{of}\, \{l_i : x_i.e_i\}_{i \in I}] \rangle) & \longrightarrow (vc, d)(\langle E[c] \rangle \mid \langle F[e_j[d/x_j]] \rangle) \quad \text{if } j \in I \\
(vc, d)(\langle E[\mathsf{close}\, c] \rangle \mid \langle F[\mathsf{wait}\, d] \rangle) & \longrightarrow \langle E[()] \rangle \mid \langle F[()] \rangle
\end{array}
$$

$$
\frac{P \longrightarrow P'}{P \mid Q \longrightarrow P' \mid Q} \qquad \frac{P \longrightarrow Q}{(vc, d)P \longrightarrow (vc, d)Q} \qquad \frac{P' \equiv P \quad P \longrightarrow Q \quad Q \equiv Q'}{P' \longrightarrow Q'} \qquad \frac{e \longrightarrow f}{\langle E[e] \rangle \longrightarrow \langle E[f] \rangle}
$$

Fig. 3. Reduction in GV

We let P, Q range over processes, which include expressions, parallel composition, and a binder that introduces a pair of channel endpoints. The initial process will consist of a single expression, corresponding to a given GV program.

The *bindings* in the language are as follows: variable x is bound in subexpression e of $\lambda_m x.e$, variables x, y are bound in subexpression f of $\mathsf{let}\, x, y = e \,\mathsf{in}\, f$, variables x_i are are bound in subexpressions e_i of $\mathsf{case}\, e \,\mathsf{of}\, \{l_i : x_i.e_i\}_{i \in I}$, channel endpoints c, d are bound in subprocess P of $(vc, d)P$. The notions of free and bound variables as well that substitution are defined accordingly. We follow Barendregt's variable convention, whereby all names in binding occurrences in any mathematical context are pairwise distinct and distinct from the free names [Barendregt 1984].

We let Γ, Δ range over environments, which are used for typing. An environment consists of zero or more associations of names with types. Environment splitting $\Gamma = \Gamma_1 \circ \Gamma_2$ is standard. It breaks an environment Γ for an expression or process into environments Γ_1 and Γ_2 for its components; a name of unrestricted type may be used in both environments, while a name of linear type must be used in one environment or the other but not both. We write $m(\Gamma)$ if $m(T)$ holds for each T in Γ, and similarly for $m^{:>}(\Gamma)$.

Write $\Gamma \vdash e : T$ if under environment Γ expression e has type T. The typing rules for expressions are standard. In the rules for names, unit, and new the remaining environment must be unrestricted, to enforce the invariant that linear variables are used exactly once. A function abstraction that is

unrestricted must have only unrestricted variables bound in its closure, and a pair that is unrestricted may only contain components that are unrestricted. The rules for send, receive, select, case, close, and wait match the corresponding session types. The typing system supports subsumption: if e has type T and T is a subtype of U then e also has type U.

Write $\Gamma \vdash P$ if under environment Γ process P is well typed. The typing rules for processes are also standard. If expression e has unrestricted type T then process $\langle e \rangle$ is well-typed. If processes P and Q are well-typed, then so is process $P \mid Q$, where the environment of the latter can be split to yield the environments for the former. And if process P is well-typed under an environment that includes channel endpoints c and d with session types S and $\overline{S}$, then process $(\nu c, d)P$ is well-typed under the same environment without c and d.

3.1.3 Reductions. Values, evaluation contexts, reductions for expressions, structural congruences, and reductions for processes for Gradual GV are summarised in Figure 3.

Let v, w range over values, which include unit, function abstractions, pairs of values, and channel endpoints. Let E, F range over evaluation contexts, which are standard.

Write $e \longrightarrow f$ to indicate that expression e reduces to expression f. Reduction is standard, consisting of beta reduction for functions and pairs.

Write $P \equiv Q$ for structural conguence of processes. It is standard, with composition being commutative and associative. A process returning the unit is the identity of parallel composition, so $P \mid \langle () \rangle \equiv P$. The order in which the endpoints are written in a ν-binder is irrelevant. Distinct prefixes commute, and satisfy scope extrusion. The Barendregt convention ensures that c, d are not free in Q in the rule for scope extrusion.

Write $P \longrightarrow Q$ if process P reduces to process Q. Evaluating $\mathsf{fork}\, e$ returns () and creates a new process $\langle e \rangle$. Evaluating new introduces a new binder $(\nu c, d)$ and returns a the pair $(c, d)_{\mathsf{lin}}$ of channel endpoints. Evaluating $\mathsf{send}\, v\, c$ on one endpoint of a channel and $\mathsf{receive}\, d$ on the other, causes the send to return c and the receive to return $(v, d)_{\mathsf{lin}}$. Similarly for select on one endpoint of a channel and case on the other, or close on one endpoint of a channel and wait on the other.

Process reduction is a congruence with regard to parallel composition and binding for channel endpoints, it is closed under structural congruence, and supports expression reduction under evaluation contexts.

3.2 Gradual GV

We now introduce Gradual GV by outlining its differences to GV.

3.2.1 Types and subtyping. Following the usual approach to gradual types, we extend the grammar of types with a *dynamic* type (sometimes also called the *unknown* type), written $\star$. Similarly, we extend session types with the dynamic session type, written $\circledast$. The extended grammar of types is given in Figure 4, where types carried over from Figure 1 are written in gray.

As before, we let T, U range over types and S, R range over session types. We also distinguish a subset of types which we call *ground types*, ranged over by $\mathbf{T}, \mathbf{U}$, and a subset of session types which we call *ground session types*, ranged over by $\mathbf{S}, \mathbf{R}$, consisting of all the type constructors applied only to arguments which are either the dynamic type or the dynamic session type, as appropriate.

We define the multiplicity of the new types by setting $\star$ to be un and $\circledast$ to be lin. The remaining definitions of multiplicity of types carries over unchanged from Figure 1. Type $\star$ is labeled unrestricted although (as we will see below) it corresponds to all possible types, both unrestricted and linear, and therefore we will need to take special care when handling values of type $\star$ that correspond to values of a linear type.

$$
\begin{array}{lrcl}
\text{Types} & T, U & ::= & \mathsf{unit} \mid S \mid T \to_m U \mid T \times_m U \mid \star \\
\text{Session types} & S, R & ::= & !T.S \mid ?T.S \mid \oplus\{l_i : S_i\}_{i \in I} \mid \&\{l_i : S_i\}_{i \in I} \mid \mathsf{end}_! \mid \mathsf{end}_? \mid \circledast \\
\text{Ground types} & \mathbf{T}, \mathbf{U} & ::= & \mathsf{unit} \mid \circledast \mid \star \to_m \star \mid \star \times_m \star \\
\text{Ground session types} & \mathbf{S}, \mathbf{R} & ::= & !\star.\circledast \mid ?\star.\circledast \mid \oplus\{l_i : \circledast\}_{i \in I} \mid \&\{l_i : \circledast\}_{i \in I} \mid \mathsf{end}_! \mid \mathsf{end}_?
\end{array}
$$

Duality $\boxed{\overline{S} = R}$

$$\overline{\circledast} = \circledast$$

Multiplicity of a type $\boxed{m(T)}$

$$\mathsf{un}(\star) \qquad \mathsf{lin}(\circledast)$$

Concretion $\boxed{\gamma(T) \in \mathcal{P}(\dot{\mathcal{T}})}$

$$
\begin{array}{rlrl}
\gamma(\mathsf{unit}) &= \{\mathsf{unit}\} & & \\
\gamma(T \to_m U) &= \{\dot{T} \to_m \dot{U} \mid \dot{T} \in \gamma(T), \dot{U} \in \gamma(U)\} & \gamma(T \times_m U) &= \{\dot{T} \times_m \dot{U} \mid \dot{T} \in \gamma(T), \dot{U} \in \gamma(U)\} \\
\gamma(!T.S) &= \{!\dot{T}.\dot{S} \mid \dot{T} \in \gamma(T), \dot{U} \in \gamma(U)\} & \gamma(?T.S) &= \{?\dot{T}.\dot{S} \mid \dot{T} \in \gamma(T), \dot{U} \in \gamma(U)\} \\
\gamma(\oplus\{l_i : S_i\}_{i \in I}) &= \{\oplus\{l_i : \dot{S}_i\} \mid (\dot{S}_i \in \gamma(S_i))_{i \in I}\} & \gamma(\&\{l_i : S_i\}_{i \in I}) &= \{\&\{l_i : \dot{S}_i\} \mid (\dot{S}_i \in \gamma(S_i))_{i \in I}\} \\
\gamma(\mathsf{end}_!) &= \{\mathsf{end}_!\} & \gamma(\mathsf{end}_?) &= \{\mathsf{end}_?\} \\
\gamma(\star) &= \dot{\mathcal{T}} & \gamma(\circledast) &= \dot{\mathcal{S}}
\end{array}
$$

Consistent subtyping $\boxed{T \lesssim U}$

$$\mathsf{unit} \lesssim \mathsf{unit} \qquad \frac{T' \lesssim T \quad U \lesssim U' \quad m <: n}{T \to_m U \lesssim T' \to_n U'} \qquad \frac{T \lesssim T' \quad U \lesssim U' \quad m <: n}{T \times_m U \lesssim T' \times_n U'}$$

$$\frac{T' \lesssim T \quad S \lesssim S'}{!T.S \lesssim !T'.S'} \qquad \frac{T \lesssim T' \quad S \lesssim S'}{?T.S \lesssim ?T'.S'} \qquad \frac{J \subseteq I \quad (S_j \lesssim S'_j)_{j \in J}}{\oplus\{l_i : S_i\}_{i \in I} \lesssim \oplus\{l_j : S'_j\}_{j \in J}} \qquad \frac{I \subseteq J \quad (S_i \lesssim S'_i)_{i \in I}}{\&\{l_i : S_i\}_{i \in I} \lesssim \&\{l_j : S'_j\}_{j \in J}}$$

$$\mathsf{end}_! \lesssim \mathsf{end}_! \qquad \mathsf{end}_? \lesssim \mathsf{end}_?$$

$$\star \lesssim T \qquad T \lesssim \star \qquad \circledast \lesssim S \qquad S \lesssim \circledast$$

Subtyping $\boxed{T <: U}$

$$\star <: \star \qquad \circledast <: \circledast$$

Fig. 4. Types and subtyping in Gradual GV

Following Garcia et al. [2016], we define the concretion $\gamma(T)$ of a type T of Gradual GV to be a set of types of GV. Let $\dot{\mathcal{T}}$ and $\dot{\mathcal{S}}$ denote the set of types and of session types of GV as defined in Figure 1. The concretion of $\star$ is all types $\dot{\mathcal{T}}$ of GV, and the concretion of $\circledast$ is all session types $\dot{\mathcal{S}}$ of GV. The concretion of each type constructor is formed in the obvious way, e.g., the concretion of the Gradual GV function type is all functions from GV types in the concretion of its domain to GV types in the concretion of its range. Duality in Gradual GV is defined such that $\gamma(\overline{S}) = \{\overline{\dot{S}} \mid \dot{S} \in \gamma(S)\}$. On GV types, this definition coincides with the original one from Figure 1 and we find that $\overline{\circledast} = \circledast$ because their concretions are equal.

$$
\begin{array}{lll}
\text{Names} & z & ::= \cdots \mid a \\
\text{Expressions} & e, f & ::= \cdots \mid e : T \overset{p}{\Rightarrow} U \mid \mathsf{blame}\, p\, q\, X \mid \mathsf{blame}\, p\, X \\
\text{Processes} & P, Q & ::= \cdots \mid (\nu a)P \mid a \mapsto w : \mathbf{T} \overset{p}{\Rightarrow} \star \mid a \mapsto \mathsf{locked}\, p
\end{array}
$$

Typing expressions $\boxed{\Gamma \vdash e : T}$

$$
\frac{\Gamma \vdash e : T \quad T \lesssim U}{\Gamma \vdash (e : T \overset{p}{\Rightarrow} U) : U}
\qquad
\frac{\mathsf{flv}(\Gamma) = X \quad \mathsf{un}(T)}{\Gamma \vdash \mathsf{blame}\, p\, q\, X : T}
\qquad
\frac{\mathsf{flv}(\Gamma) = X \quad \mathsf{un}(T)}{\Gamma \vdash \mathsf{blame}\, p\, X : T}
$$

Typing processes $\boxed{\Gamma \vdash P}$

$$
\frac{\Gamma, a : \star \vdash P}{\Gamma \vdash (\nu a)P}
\qquad
\frac{\Gamma \vdash a : \star \quad \Delta \vdash w : \mathbf{T} \quad \mathsf{lin}(\mathbf{T})}{\Gamma \circ \Delta \vdash a \mapsto w : \mathbf{T} \overset{p}{\Rightarrow} \star}
\qquad
\frac{\Gamma \vdash a : \star}{\Gamma \vdash a \mapsto \mathsf{locked}\, p}
$$

Fig. 5. Expressions, processes, and typing in Gradual GV

Consistent subtyping is defined over types of Gradual GV in terms of subtyping over types of GV. In particular, we define $T \lesssim U$ if there exists $\dot{T}$ in $\gamma(T)$ and $\dot{U}$ in $\gamma(U)$ such that $\dot{T} <: \dot{U}$. Consistent subtyping is written out in full in Figure 4. It is identical to the definition of subtyping from Figure 1, with each occurrence of $<:$ replaced by $\lesssim$, and with the addition of four rules for the new types

$$\star \lesssim T \qquad T \lesssim \star \qquad \circledast \lesssim S \qquad S \lesssim \circledast$$

For example, we have (a) $\oplus\{l_1 : !\star.\, \circledast, l_2 : ?\star.\, \circledast\} \lesssim \oplus\{l_1 : \circledast\}$ and (b) $\&\{l_1 : \circledast\} \lesssim \&\{l_1 : !\star.\, \circledast, l_2 : ?\star.\, \circledast\}$. Consistent subtyping is reflexive, but neither symmetric nor transitive. As with subtyping, we have $\overline{S} \lesssim R$ iff $\overline{R} \lesssim S$. In Gradual GV, we will be permitted to attempt to cast a value of type T to a value of type U exactly when $T \lesssim U$. A cast may fail at run time: while a cast using (a) will not fail, a cast using (b) may fail because the concretion $\gamma(\&\{l_1 : \circledast\})$ contains $\&\{l_1 : \mathsf{end}_!\}$.

Two types are consistent, written $T \sim U$, if $T \lesssim U$ and $U \lesssim T$. Consistency is reflexive and symmetric but not transitive. The standard example of the failure of transitivity is that for any function type we have $T \to_m U \sim \star$ and for any product type we have $\star \sim T' \times_n U'$, but $T \to_m U \not\sim T' \times_n U'$. In the setting of session types one has for example $?T.\, S \sim \circledast$ and $\circledast \sim \mathsf{end}_!$, but $?T.\, S \not\sim \mathsf{end}_!$.

Subtyping $T <: U$ for Gradual GV essentially carries over from GV. Its definition is exactly as in Figure 1, with the addition of two rules that ensure subtyping is reflexive for the dynamic type and the dynamic session type. In contrast to consistent subtyping, subtyping $T <: U$ guarantees that we may always treat a value of the first type as if it belongs to the second type without casting.

3.2.2 Expressions, processes, and typing. Expression, processes, and type rules of Gradual GV are summarised in Figure 5. The expressions of Gradual GV are those of GV, plus additional forms for casts and blame. A cast is written

$$e : T \overset{p}{\Rightarrow} U$$

where e is an expression of type T and $T \lesssim U$, and the entire expression has type U. If a cast in a program fails, the cause of the failure is indicated by evaluating to blame p. We let p, q range over blame labels. If the cast above fails, it may do so in one of two ways: either because the term e contained in the cast, which necessarily returns a value of type T, cannot return a value of type U; or because the context containing the cast, which necessarily accepts a value of type

U, cannot accept a value of type T. We call the first case *positive blame* and indicate it by blaming the label p (on the cast) and we call the second case *negative blame* and indicate it by blaming the complemented blame label $\overline{p}$. Complement is an involution on blame labels, so that $\overline{\overline{p}} = p$.

Blame is indicated by terms of the form

$$\mathsf{blame}\, p\, q\, X \qquad \text{or} \qquad \mathsf{blame}\, p\, X$$

where p and q are blame labels, and X is a set of variables of linear type. As we will see, most instances that yield blame involve two casts, hence the form with two blame labels, although blame can arise for a single cast, hence the form with one blame label. The set X records all linear variables in scope when blame is raised, and is used to maintain the invariant that as a program executes each variable of linear type appears linearly (only once, or once in each branch of a case). Discarding linear variables when raising blame would break the invariant. Blame corresponds to raising an exception, and the list of linear variables corresponds to cleaning up after linear resources when raising an exception (for instance, closing an open file or channel). In the typing rules, the notation $\mathsf{flv}(\Gamma)$ refers set of free variables of linear type that appear in Γ. We also write $\mathsf{flv}(E)$ and $\mathsf{flv}(v)$ for the free linear variables appearing in an evaluation context E or a value v.

The processes of Gradual GV are those of GV, plus three additional forms for linear references. Recall that a value of type $\star$ may in fact be linear, in which case dynamic checking must ensure that it is used exactly once. The mechanism for doing so is to allocate a linear reference. We let a, b range over linear references. A linear reference is of type $\star$, and contains a value w of ground type $\mathbf{T}$, where $\mathbf{T}$ is linear (either $\star \to_{\mathsf{lin}} \star$ or $\star \times_{\mathsf{lin}} \star$ or the dynamic session type $\circledast$). Linear references are allocated by the binding form $(\nu a)P$, and the value contained in store a is indicated by a process which is either of the form

$$a \mapsto w : \mathbf{T} \overset{p}{\Rightarrow} \star \qquad \text{or} \qquad a \mapsto \mathsf{locked}\, p$$

where w is a value of type $\mathbf{T}$ and p is a blame label. Linear references initially take the first form, but change to the second form after the linear reference has been accessed once; any subsequent attempt to access the reference a second time will cause an error.

3.2.3 Reductions. Values, evaluation contexts, reductions for expressions, structural congruences, and reductions for processes for Gradual GV are summarised in Figures 6 and 7.

The values of Gradual GV are those of GV, plus five additional forms. Values of dynamic type either have the form $v : \mathbf{T} \overset{p}{\Rightarrow} \star$ as in other blame calculi, if $\mathbf{T}$ is unrestricted, or a linear reference a, if the dynamic type wraps a linear value. Additionally, there are values of dynamic session type which take the form $v : \mathbf{S} \overset{p}{\Rightarrow} \circledast$.

Following standard practice for blame calculus, we take a cast of a value between function types to be a value, and for similar reasons a cast from a session type to a session type is a value unless one ends of the cast is the dynamic session type:

$$v : T \to_m U \overset{p}{\Rightarrow} T' \to_n U' \qquad \text{or} \qquad v : S \overset{p}{\Rightarrow} R$$

where $T \to_m U \lesssim T' \to_n U'$ and $S \lesssim R$ with $S, R \neq \circledast$.

Additional reductions for expressions appear in Figure 6. Typical of blame calculus is the reduction for a cast between function types, often called the *wrap* rule:

$$(v : T \to_m U \overset{p}{\Rightarrow} T' \to_n U')\, w \longrightarrow (v\, (w : T' \overset{\overline{p}}{\Rightarrow} T)) : U \overset{p}{\Rightarrow} U'$$

The cast on the function decomposes into two casts, one on the domain and one on the range. The fact that function subtyping (and consistent subtyping) is contravariant on the domain and

Values $v, w ::= \cdots \mid v : \mathbf{T} \overset{p}{\Rightarrow} \star \mid v : \mathbf{S} \overset{p}{\Rightarrow} \circledast \mid v : T \to_m U \overset{p}{\Rightarrow} T' \to_n U' \mid v : S \overset{p}{\Rightarrow} R \mid a$
where $\mathsf{un}(\mathbf{T}), S \neq \circledast, R \neq \circledast$

Eval contexts $E, F ::= \cdots \mid E : T \overset{p}{\Rightarrow} U$

Expression reduction $\boxed{e \longrightarrow f}$

$$v : \star \overset{p}{\Rightarrow} \star \longrightarrow v$$

$$v : \circledast \overset{p}{\Rightarrow} \circledast \longrightarrow v$$

$$v : \mathsf{unit} \overset{p}{\Rightarrow} \mathsf{unit} \longrightarrow v$$

$$(v : T \to_m U \overset{p}{\Rightarrow} T' \to_n U')\, w \longrightarrow (v\,(w : T' \overset{\overline{p}}{\Rightarrow} T)) : U \overset{p}{\Rightarrow} U'$$

$$(v, w)_m : T \times_m U \overset{p}{\Rightarrow} T' \times_n U' \longrightarrow (v : T \overset{p}{\Rightarrow} T', w : U \overset{p}{\Rightarrow} U')_n$$

$$\mathsf{send}\, v\, (w : !T.S \overset{p}{\Rightarrow} !T'.S') \longrightarrow (\mathsf{send}\, (v : T' \overset{\overline{p}}{\Rightarrow} T)\, w) : S \overset{p}{\Rightarrow} S'$$

$$\mathsf{receive}\, (w : ?T.S \overset{p}{\Rightarrow} ?T'.S') \longrightarrow \mathsf{let}\, x, y = \mathsf{receive}\, w\, \mathsf{in}\, (x : T \overset{p}{\Rightarrow} T', y : S \overset{p}{\Rightarrow} S')_{\mathsf{lin}}$$

$$\mathsf{select}\, l_k\, (w : \oplus\{l_i : R_i\}_{i \in I} \overset{p}{\Rightarrow} \oplus\{l_j : S_j\}_{j \in J}) \longrightarrow (\mathsf{select}\, l_k\, w) : R_k \overset{p}{\Rightarrow} S_k \qquad \text{if } k \in J,\ J \subseteq I$$

$$\mathsf{case}\, (w : \&\{l_i : R_i\}_{i \in I} \overset{p}{\Rightarrow} \&\{l_i : S_i\}_{i \in J})\, \mathsf{of}\, \{l_j : x_j.e_j\}_{j \in J} \longrightarrow$$
$$\mathsf{case}\, w\, \mathsf{of}\, \{l_i : x_i.\mathsf{let}\, x_i = (x_i : R_i \overset{p}{\Rightarrow} S_i)\, \mathsf{in}\, e_i\}_{i \in I} \qquad \text{if } I \subseteq J$$

$$\mathsf{close}\, (v : \mathsf{end}_! \overset{p}{\Rightarrow} \mathsf{end}_!) \longrightarrow \mathsf{close}\, v$$

$$\mathsf{wait}\, (v : \mathsf{end}_? \overset{p}{\Rightarrow} \mathsf{end}_?) \longrightarrow \mathsf{wait}\, v$$

$$v : T \overset{p}{\Rightarrow} \star \longrightarrow (v : T \overset{p}{\Rightarrow} \mathbf{T}) : \mathbf{T} \overset{p}{\Rightarrow} \star \qquad \text{if } T \neq \star, T \neq \mathbf{T}, T \sim \mathbf{T}$$

$$v : \star \overset{p}{\Rightarrow} T \longrightarrow (v : \star \overset{p}{\Rightarrow} \mathbf{T}) : \mathbf{T} \overset{p}{\Rightarrow} T \qquad \text{if } T \neq \star, T \neq \mathbf{T}, T \sim \mathbf{T}$$

$$v : S \overset{p}{\Rightarrow} \circledast \longrightarrow (v : S \overset{p}{\Rightarrow} \mathbf{S}) : \mathbf{S} \overset{p}{\Rightarrow} \circledast \qquad \text{if } S \neq \circledast, S \neq \mathbf{S}, S \sim \mathbf{S}$$

$$v : \circledast \overset{p}{\Rightarrow} S \longrightarrow (v : \circledast \overset{p}{\Rightarrow} \mathbf{S}) : \mathbf{S} \overset{p}{\Rightarrow} S \qquad \text{if } S \neq \circledast, S \neq \mathbf{S}, S \sim \mathbf{S}$$

Fig. 6. Reduction in Gradual GV, expressions

covariant on the range is reflected in the fact that the cast on the domain is from T' to T and complements the blame label $\overline{p}$, while the cast on the range is form U to U' and leaves the blame label p unchanged. Casts for products follow a similar pattern, though covariant on all components.

Reductions on session types follow the pattern of the reduction for a cast between send types:

$$\mathsf{send}\, v\, (w : !T.S \overset{p}{\Rightarrow} !T'.S') \longrightarrow (\mathsf{send}\, (v : T' \overset{\overline{p}}{\Rightarrow} T)\, w) : S \overset{p}{\Rightarrow} S'$$

The cast on the send decomposes into two casts, one on the value sent and one on the residual session type. The fact that send subtyping (and consistent subtyping) is contravariant on the value sent and covariant on the residual session type is reflected in the fact that the cast on the value

Structural congruence $\boxed{P \equiv Q}$

$$((va)P) \mid Q \equiv (va)(P \mid Q) \quad (va)(vb)P \equiv (vb)(va)P \quad (vc,d)(va)P \equiv (va)(vc,d)P \quad (va)\langle()\rangle \equiv \langle()\rangle$$

Process reduction $\boxed{P \longrightarrow Q}$

$$\langle E[w : \mathbf{T} \overset{p}{\Rightarrow} \star]\rangle \longrightarrow (va)(\langle E[a]\rangle \mid a \mapsto w : \mathbf{T} \overset{p}{\Rightarrow} \star) \qquad \text{if } \mathsf{lin}(\mathbf{T}) \text{ and } E \neq F[[\,] : \star \overset{q}{\Rightarrow} \mathbf{U}]$$

$$\langle E[a : \star \overset{q}{\Rightarrow} \mathbf{U}]\rangle \mid a \mapsto w : \mathbf{T} \overset{p}{\Rightarrow} \star \longrightarrow \langle E[(w : \mathbf{T} \overset{p}{\Rightarrow} \star) : \star \overset{q}{\Rightarrow} \mathbf{U}]\rangle \mid a \mapsto \mathsf{locked}\, p$$

$$\langle E[a : \star \overset{q}{\Rightarrow} \mathbf{U}]\rangle \mid a \mapsto \mathsf{locked}\, p \longrightarrow \langle \mathsf{blame}\, \overline{p}\, q\, (\mathsf{flv}(E))\rangle \mid a \mapsto \mathsf{locked}\, p$$

$$(va)(a \mapsto \mathsf{locked}\, p) \longrightarrow \langle()\rangle$$

$$(va)(a \mapsto w : \mathbf{T} \overset{p}{\Rightarrow} \star) \longrightarrow \langle \mathsf{blame}\, \overline{p}\, (\mathsf{flv}(w))\rangle$$

$$\langle E[(v : \mathbf{T} \overset{p}{\Rightarrow} \star) : \star \overset{q}{\Rightarrow} \mathbf{U}]\rangle \longrightarrow \langle E[v]\rangle \qquad \text{if } \mathbf{T} <: \mathbf{U}$$

$$\langle E[(v : \mathbf{T} \overset{p}{\Rightarrow} \star) : \star \overset{q}{\Rightarrow} \mathbf{U}]\rangle \longrightarrow \langle \mathsf{blame}\, \overline{p}\, q\, (\mathsf{flv}(E) \cup \mathsf{flv}(v))\rangle \qquad \text{if } \mathbf{T} \not<: \mathbf{U}$$

$$\langle E[(v : \mathbf{S} \overset{p}{\Rightarrow} \circledast) : \circledast \overset{q}{\Rightarrow} \mathbf{R}]\rangle \longrightarrow \langle E[v]\rangle \qquad \text{if } \mathbf{S} <: \mathbf{R}$$

$$\langle E[(v : \mathbf{S} \overset{p}{\Rightarrow} \circledast) : \circledast \overset{q}{\Rightarrow} \mathbf{R}]\rangle \longrightarrow \langle \mathsf{blame}\, \overline{p}\, q\, (\mathsf{flv}(E) \cup \mathsf{flv}(v))\rangle \qquad \text{if } \mathbf{S} \not<: \mathbf{R}$$

$$(vc,d)(\langle E[c : \circledast \overset{p}{\Rightarrow} \mathbf{S}]\rangle \mid \langle F[d : \circledast \overset{q}{\Rightarrow} \mathbf{R}]\rangle) \longrightarrow (vc,d)(\langle E[c]\rangle \mid \langle F[d]\rangle) \qquad \text{if } \overline{\mathbf{S}} <: \mathbf{R}$$

$$(vc,d)(\langle E[c : \circledast \overset{p}{\Rightarrow} \mathbf{S}]\rangle \mid \langle F[d : \circledast \overset{q}{\Rightarrow} \mathbf{R}]\rangle) \longrightarrow \langle \mathsf{blame}\, p\, q\, (\mathsf{flv}(E) \cup \mathsf{flv}(F))\rangle \qquad \text{if } \overline{\mathbf{S}} \not<: \mathbf{R}$$

$$\frac{P \longrightarrow Q}{(va)P \longrightarrow (va)Q}$$

Fig. 7. Reduction in Gradual GV, processes

sent is from T' to T and complements the blame label $\overline{p}$, while the cast on the residual session type is from S to S' and leaves the blame label p unchanged. The casts for the remaining session types follow a similar pattern, though covariant on all components.

Also typical of blame calculus, casts to the dynamic type factor through a ground type,

$$v : T \overset{p}{\Rightarrow} \star \longrightarrow (v : T \overset{p}{\Rightarrow} \mathbf{T}) : \mathbf{T} \overset{p}{\Rightarrow} \star$$

when $T \neq \star$, $T \neq \mathbf{T}$, and $T \sim \mathbf{T}$. This factoring is unique because for every type T such that $T \neq \star$ there is a unique ground type $\mathbf{T}$ such that $T \sim \mathbf{T}$. The additional condition $T \neq \mathbf{T}$ ensures that the factoring is non-trivial and that reduction does not enter a loop. Casts from the dynamic type, and casts to and from the dynamic session type are handled analogously.

Additional structural congruences and reductions for expressions appear in Figure 7. Like bindings for channel endpoints, bindings for linear references satisfy scope extrusion and reduction is a congruence with respect to them.

The first five reduction rules for processes deal with linear references, which ensure that a value cast from a linear type to a dynamic type is accessed exactly once. As the only values of dynamic

type are casts from ground type, the expressions of interest take the form

$$w : \mathbf{T} \overset{p}{\Rightarrow} \star$$

where w is a value and $\mathbf{T}$ is a linear ground type. The first rule introduces a linear reference, represented as a separate process of the form $a \mapsto w : \mathbf{T} \overset{p}{\Rightarrow} \star$. The context restriction ensures that a linear reference is only introduced if the value is not immediately accessed. Any attempt to access the linear reference a must take the form

$$E[a : \star \overset{q}{\Rightarrow} \mathbf{U}]$$

where E is an evaluation context and $\mathbf{U}$ is a ground type that may or may not be linear. The second rule implements the first access to a linear value by copying the value v in place of the linear reference a, and updating the reference process to $a \mapsto \mathsf{locked}\ p$, indicating that the linear reference has been accessed once. The third rule implements any subsequent attempt to access a linear value, which allocates blame to both of the casts involved, negative blame $\overline{p}$ for the inner cast and positive blame q for the outer cast, indicating that in both cases blame is allocated to the side of the cast of type $\star$. The blame term also contains $\mathsf{flv}(E)$, the set of free linear variables that appear in the context E, which as mentioned earlier is required to maintain the invariant on linear variables; all occurrences of blame contain corresponding sets of linear variables, which we will not mention further. The final two rules indicate what happens when all processes containing the linear reference finish execution. In practice, these rules would be implemented as part of garbage collection. If the linear reference is locked then it was accessed once, and the reference may be deallocated as usual. If the reference is not locked then it was never accessed, and blame should be allocated to the context of the original cast, which discarded the value rather than using it linearly.

The remaining six rules come in three pairs. Typical of blame calculus is the first pair, often called the *collapse* and *collide* rules:

$$\langle E[(v : \mathbf{T} \overset{p}{\Rightarrow} \star) : \star \overset{q}{\Rightarrow} \mathbf{U}]\rangle \longrightarrow \langle E[v]\rangle \qquad \text{if } \mathbf{T} <: \mathbf{U}$$

$$\langle E[(v : \mathbf{T} \overset{p}{\Rightarrow} \star) : \star \overset{q}{\Rightarrow} \mathbf{U}]\rangle \longrightarrow \langle \mathsf{blame}\ \overline{p}\ q\ (\mathsf{flv}(E) \cup \mathsf{flv}(v))\rangle \qquad \text{if } \mathbf{T} \not<: \mathbf{U}$$

If the source type is a subtype of the target type, the casts collapse to the original value. Types are preserved by subsumption: since v has type $\mathbf{T}$ and $\mathbf{T} <: \mathbf{U}$ then v also has type $\mathbf{U}$. Conversely, if the source type is not a subtype of the target type, then the casts are in collision and reduce to blame. Blame is allocated to both of the casts involved, negative blame $\overline{p}$ for the inner cast and positive blame q for the outer cast, indicating that in both cases blame is allocated to the side of the cast of type $\star$. Our choice to allocate blame to both casts differs from the usual formulation of blame calculus, which only allocates blame to the outer cast. Allocating blame to only the outer cast is convenient if one wishes to implement blame calculus by erasure to a dynamically typed language, where injection of a value to the dynamic type is represented by the value itself, that is, the erasure of $v : \mathbf{T} \overset{p}{\Rightarrow} \star$ is just taken to be the erasure of v itself. However, this asymetric implementation is less appropriate in our situation. For session types, a symmetric formulation is more appropriate, as we will see shortly when we look at the interaction between casts and communication.

The next pair of rules transpose collapse and collide from types to session types. The final pair of rules adapt collapse and collide to the case of communication between two channel endpoints. Here is the adapted collapse rule.

$$(\nu c, d)(\langle E[c : \circledast \overset{p}{\Rightarrow} \mathbf{S}]\rangle \mid \langle F[d : \circledast \overset{q}{\Rightarrow} \mathbf{R}]\rangle) \longrightarrow (\nu c, d)(\langle E[c]\rangle \mid \langle F[d]\rangle) \ \text{ if } \overline{\mathbf{S}} <: \mathbf{R}$$

$$
\begin{aligned}
\lceil x \rceil &= x \\
\lceil () \rceil &= () : \mathsf{unit} \Rightarrow \star \\
\lceil \lambda x.e \rceil &= (\lambda_{\mathsf{un}} x.\lceil e \rceil) : \star \to_{\mathsf{un}} \star \Rightarrow \star \\
\lceil (e, f) \rceil &= (\lceil e \rceil, \lceil f \rceil)_{\mathsf{un}} : \star \times_{\mathsf{un}} \star \Rightarrow \star \\
\lceil e\ f \rceil &= (\lceil e \rceil : \star \Rightarrow \star \to_{\mathsf{lin}} \star)\ \lceil f \rceil \\
\lceil \mathsf{let}\ x, y = e\ \mathsf{in}\ f \rceil &= \mathsf{let}\ x, y = (\lceil e \rceil : \star \Rightarrow \star \times_{\mathsf{lin}} \star)\ \mathsf{in}\ \lceil f \rceil \\
\lceil \mathsf{fork}\ e \rceil &= (\mathsf{fork}\ (\lceil e \rceil : \star \Rightarrow \mathsf{unit})) : \mathsf{unit} \Rightarrow \star \\
\lceil \mathsf{new} \rceil &= \mathsf{new} : \circledast \times_{\mathsf{lin}} \circledast \Rightarrow \star \\
\lceil \mathsf{send}\ e\ f \rceil &= (\mathsf{send}\ \lceil e \rceil\ (\lceil f \rceil : \star \Rightarrow\ !\star.\circledast)) : \circledast \Rightarrow \star \\
\lceil \mathsf{receive}\ e \rceil &= (\mathsf{receive}\ (\lceil e \rceil : \star \Rightarrow\ ?\star.\circledast)) : \star \times_{\mathsf{lin}} \circledast \Rightarrow \star \\
\lceil \mathsf{select}\ l\ e \rceil &= (\mathsf{select}\ l\ (\lceil e \rceil : \star \Rightarrow \oplus\{l : \circledast\})) : \circledast \Rightarrow \star \\
\lceil \mathsf{case}\ e\ \mathsf{of}\ \{l_i : x_i.e_i\}_{i \in I} \rceil &= \mathsf{case}\ (\lceil e \rceil : \star \Rightarrow \&\{l_i : \circledast\}_{i \in I})\ \mathsf{of}\ \{l_i : y_i.\mathsf{let}\ x_i = (y_i : \circledast \Rightarrow \star)\ \mathsf{in}\ \lceil e_i \rceil\}_{i \in I} \\
\lceil \mathsf{close}\ e \rceil &= (\mathsf{close}\ (\lceil e \rceil : \star \Rightarrow \mathsf{end}_!)) : \mathsf{unit} \Rightarrow \star \\
\lceil \mathsf{wait}\ e \rceil &= (\mathsf{wait}\ (\lceil e \rceil : \star \Rightarrow \mathsf{end}_?)) : \mathsf{unit} \Rightarrow \star
\end{aligned}
$$

Fig. 8. Embedding of the unityped calculus

The condition on this rule is symmetric, since $\overline{\mathbf{S}} <: \mathbf{R}$ if and only if $\overline{\mathbf{R}} <: \mathbf{S}$. On the left-hand side of this rule c, d both have session type $\circledast$, while on the right-hand side of the rule c, d have session types $\mathbf{S}, \overline{\mathbf{S}}$ or $\overline{\mathbf{R}}, \mathbf{R}$. Again, types are preserved by subsumption, since if c, d have session types $\mathbf{S}, \overline{\mathbf{S}}$ and $\overline{\mathbf{S}} <: \mathbf{R}$ then c, d also have session types $\mathbf{S}, \mathbf{R}$, and similarly if c, d have session types $\overline{\mathbf{R}}, \mathbf{R}$. Similarly, the last rule adapts collide.

An alternative design might replace the final pair of rules by a structural congruence that slides a cast from one endpoint of a channel to the other:

$$(\nu c, d)(E[c : S \overset{p}{\Rightarrow} R] \mid F[d]) \equiv (\nu c, d)(E[c] \mid F[d : \overline{R} \overset{\overline{p}}{\Rightarrow} \overline{S}]).$$

Setting S to $\circledast$ and R to $\mathbf{S}$, this congruence can reduce the third collapse rule (on channel endpoints) to the second collapse rule (on a nested pair of casts on session types). However, even with this congruence the two collide rules are not quite equivalent. Our chosen formulation, though slightly longer, is more symmetric and more straightforward to implement.

3.2.4 Embedding. Siek et al. [2015] argue that one desideratum for a gradual typing system is that it is possible to embed an untyped (or rather, unityped) language within it. An embedding of an untyped variant of GV into Gradual GV is given in Figure 8. Blame labels are omitted; each cast should receive a unique blame label. The untyped variant has the same syntax as the expressions of GV, but every expression has type $\star$. The embedding extends one from untyped lambda calculus into the blame calculus given by Wadler and Findler [2009].

4 RESULTS

4.1 Preservation and Progress

Proofs in this section are for Gradual GV described in Section 3.2. Proofs for GV of Section 3.1 are similar except that they do not mention casts and blame, which are specific to Gradual GV.

Lemma 4.1. *$<:$ is a pre-order.*

Lemma 4.2 (Weakening). *If $\Gamma \vdash e : T$ and $\mathsf{un}(U)$, then $\Gamma, x : U \vdash e : T$.*

Lemma 4.3 (Preservation for $\equiv$). *If $\Gamma \vdash P$ and $P \equiv Q$ then $\Gamma \vdash Q$.*

Lemma 4.4 (Substitution). *If $\Gamma_1 \vdash v : U$ and $\Gamma_2, x : U \vdash e : T$ and $\Gamma = \Gamma_1 \circ \Gamma_2$ then $\Gamma \vdash e[v/x] : T$.*

The following two lemmas are adapted from Gay and Vasconcelos [2010].

Lemma 4.5 (Sub-derivation introduction). *If $\mathcal{D}$ is a derivation of $\Gamma \vdash E[e] : T$, then there exist Γ_1, Γ_2 and U such that $\Gamma = \Gamma_1 \circ \Gamma_2$ and $\mathcal{D}$ has a sub-derivation $\mathcal{D}'$ concluding $\Gamma_2 \vdash e : U$ and the position of $\mathcal{D}'$ in $\mathcal{D}$ corresponds to the position of the hole in E.*

Lemma 4.6 (Sub-derivation elimination). *$\Gamma \vdash E[f] : T$ holds, if*

- *$\mathcal{D}$ is a derivation of $\Gamma_1 \circ \Gamma_2 \vdash E[e] : T$,*
- *$\mathcal{D}'$ is a sub-derivation of $\mathcal{D}$ concluding $\Gamma_2 \vdash e : U$,*
- *the position of $\mathcal{D}'$ in $\mathcal{D}$ corresponds to the position of the hole in E,*
- *$\Gamma_3 \vdash f : U$,*
- *$\Gamma = \Gamma_1 \circ \Gamma_3$*

Theorem 4.7 (Preservation for reduction).

(1) *If $e \longrightarrow f$ and $\Gamma \vdash e : T$, then $\Gamma \vdash f : T$;*
(2) *If $P \longrightarrow Q$ and $\Gamma \vdash P$, then $\Gamma \vdash Q$.*

Proof. 1. By rule induction on the first hypothesis. For β-reduction and let we use the substitution lemma (Lemma 4.4), using an 'inversion of the typing relation' lemma which we omit.

2. By rule induction on the first hypothesis, using basic properties of context splitting [Vasconcelos 2012; Walker 2005] and weakening (Lemma 4.2). Rules that make use of context use subderivation introduction (Lemma 4.5) to build the derivation for the hypothesis, and subderivation elimination (Lemma 4.6) to build the derivation for the conclusion. Contextual closure uses clause 1 of this theorem. Reduction underneath parallel composition and scope restriction follow by induction. The rule for $\equiv$ uses Lemma 4.3. □

Lemma 4.8 (Ground types, subtyping, and consistent subtyping).

(1) *If $T \neq \star$, there is a unique ground type $\boldsymbol{T}$ such that $\boldsymbol{T} \sim T$.*
(2) *If $S \neq \circledast$, there is a unique ground session type $\boldsymbol{S}$ such that $\boldsymbol{S} \sim S$.*
(3) *$\boldsymbol{T} \lesssim \boldsymbol{U}$ iff $\boldsymbol{T} <: \boldsymbol{U}$.*
(4) *$\boldsymbol{S} \lesssim \boldsymbol{R}$ iff $\boldsymbol{S} <: \boldsymbol{R}$.*

Lemma 4.9 (Canonical forms). *Suppose that $\Gamma \vdash v : T$ where Γ contains session types and $\star$, only.*

(1) *If $T = \star$, then either $v = w : \boldsymbol{T} \overset{p}{\Rightarrow} \star$ with $\mathsf{un}(\boldsymbol{T})$ or $v = a$.*
(2) *If $T = S$, then either $v = c$ or $v = w : \boldsymbol{S} \overset{p}{\Rightarrow} \circledast$ and $S = \circledast$ or $v = w : R_1 \overset{p}{\Rightarrow} R_2$ with $R_2 <: S$.*
(3) *If $T = \mathsf{unit}$, then $v = ()$.*
(4) *If $T = U_1 \to_m U_2$, then either $v = \lambda_n x.e$ with $n <: m$ or $v = w : T_1 \to_{n_1} T_2 \overset{p}{\Rightarrow} U_1' \to_{n_2} U_2'$ with $n_2 <: m$ and $U_1 <: U_1'$ and $U_2' <: U_2$.*
(5) *If $T = T \times_m U$, then $v = (w_1, w_2)_n$ with $n <: m$.*

Theorem 4.10 (Progress for expressions). *Suppose that $\Gamma \vdash e : T$ and that Γ only contains channel endpoints and linear references. Then exactly one of the following cases holds.*

(1) *e is a value,*
(2) *$e \longrightarrow f$,*

(3) $e = E[f]$ *and* f *is a GV operation:* fork f', new, send $v\ c$, receive c, select $l\ c$, case c of $\{l_i : x_i.e_i\}$, close c, *or* wait c,
(4) $e = E[f]$ *and* f *is a Gradual GV operation:* $w : \boldsymbol{T} \overset{p}{\Rightarrow} \star$ *with* $\mathsf{lin}(\boldsymbol{T})$, $a : \star \overset{p}{\Rightarrow} \boldsymbol{U}$, $(v : \boldsymbol{T} \overset{p}{\Rightarrow} \star) : \star \overset{q}{\Rightarrow} \boldsymbol{U}$, $(v : \boldsymbol{S} \overset{p}{\Rightarrow} \circledast) : \circledast \overset{q}{\Rightarrow} \boldsymbol{R}$, *or* $c : \circledast \overset{p}{\Rightarrow} \boldsymbol{S}$ *and* $E[]$ *does not end in a cast.*

Proof. By induction on expressions, using Canonical forms (Lemma 4.9). □

The notion of *run-time errors* helps in stating our type safety result. The *subject* of an expression e, denoted by $\mathsf{subj}(e)$, is c when e falls into one of the following cases and undefined in all other cases.

$$\mathsf{send}\ f\ c \qquad \mathsf{receive}\ c \qquad \mathsf{select}\ l\ c \qquad \mathsf{case}\ c\ \mathsf{of}\ \{l_i : x_i.f_i\}_{i \in I} \qquad \mathsf{close}\ c \qquad \mathsf{wait}\ c$$

Two expressions e and f *agree* on a channel with ends in set $\{c, d\}$ where $c \neq d$, denoted $\mathsf{agree}^{\{c,d\}}\{e, f\}$, a relation on two two-element sets, in the following cases.

(1) $\mathsf{agree}^{\{c,d\}}\{\mathsf{send}\ v\ c, \mathsf{receive}\ d\}$;
(2) $\mathsf{agree}^{\{c,d\}}\{\mathsf{select}\ l_j\ c, \mathsf{case}\ d\ \mathsf{of}\ \{l_i : x_i.f_i\}_{i \in I}\}$ and $j \in I$;
(3) $\mathsf{agree}^{\{c,d\}}\{\mathsf{close}\ c, \mathsf{wait}\ d\}$.

A process is an *error* if it is structurally congruent to some process that contains a subprocess of one of the following forms.

(1) $\langle E[v e]\rangle$ and v is not an abstraction;
(2) $\langle E[\mathsf{let}\ a, b = v\ \mathsf{in}\ e]\rangle$ and v is not a pair;
(3) $\langle E[e]\rangle \mid \langle F[f]\rangle$ and $\mathsf{subj}(e) = \mathsf{subj}(f)$;
(4) $(\nu c, d)(\langle E[e]\rangle \mid \langle F[f]\rangle)$ and $\mathsf{subj}(e) = c$ and $\mathsf{subj}(f) = d$ and not $\mathsf{agree}^{\{c,d\}}\{e, f\}$.

The first two cases are typical of functional languages. The third case ensures no two threads hold references to the same channel endpoint. The fourth case ensures channel endpoints agree at all times: if one process is ready to send then the other is ready to receive, and similarly for select and case, close and wait.

For processes, rather than a progress result, we present a type safety result as our type system does not rule out deadlocks, though we do not open new chances for deadlocks when compared to GV [Gay and Vasconcelos 2010]. Our result holds both for GV and Gradual GV alike.

Theorem 4.11 (Absence of run-time errors). *Let* $\Gamma \vdash P$ *where* Γ *does not contain function or pair types, and let* $P \longrightarrow^* Q$. *Then* Q *is not an error.*

4.2 Blame Safety

Following Wadler and Findler [2009] we introduce three new subtyping relations: $<:^+$, $<:^-$, and $<:_{\mathsf{n}}$, called positive, negative, and naive subtyping respectively, in addition to the ordinary subtyping $<:$ defined in Figure 4.

A cast from T to U with label p may either return a value or may raise blame p (called *positive* blame) or blame $\overline{p}$ (called *negative* blame). Relation $T <: U$ characterizes when a cast from T to U *never* yields blame; relations $T <:^+ U$ and $T <:^- U$ characterize when a cast from T to U cannot yield *positive* or *negative* blame, respectively; and relation $T <:_{\mathsf{n}} U$ characterizes when type T is more *precise* (in the sense of being less dynamic) than type U. All four relations are reflexive and transitive, and subtyping, positive subtyping, and naive subtyping are antisymmetric.

Wadler and Findler [2009] have an additional rule that makes any subtype of a ground type a subtype of $\star$, i.e., $T <: \star$ if $T <: \mathbf{T}$. This rule is not sound in Gradual GV because our collide rule blames both casts:

$$\langle E[(v : \mathbf{T} \overset{p}{\Rightarrow} \star) : \star \overset{q}{\Rightarrow} \mathbf{U}]\rangle \longrightarrow \langle \mathsf{blame}\ \overline{p}\ q\ (\mathsf{flv}(E) \cup \mathsf{flv}(v))\rangle \qquad \text{if } \mathbf{T} \not<: \mathbf{U}$$

Positive and negative subtyping $\boxed{T <:^{+} U \quad T <:^{-} U}$

$$T <:^{+} \star \qquad S <:^{+} \circledast \qquad \star <:^{-} T \qquad \circledast <:^{-} S$$

$$\mathsf{unit} <:^{\pm} \mathsf{unit} \qquad \frac{T' <:^{\mp} T \quad U <:^{\pm} U' \quad m <: n}{T \to_m U <:^{\pm} T' \to_n U'} \qquad \frac{T <:^{\pm} T' \quad U <:^{\pm} U' \quad m <: n}{T \times_m U <:^{\pm} T' \times_n U'}$$

$$\frac{T' <:^{\mp} T \quad S <:^{\pm} S'}{!T.S <:^{\pm} !T'.S'} \qquad \frac{T <:^{\pm} T' \quad S <:^{\pm} S'}{?T.S <:^{\pm} ?T'.S'} \qquad \frac{(S_k <:^{-} R_k)_{k \in I \cap J}}{\oplus\{l_i : S_i\}_{i \in I} <:^{-} \oplus\{l_j : R_j\}_{j \in J}}$$

$$\frac{(S_k <:^{-} R_k)_{k \in I \cap J}}{\&\{l_i : S_i\}_{i \in I} <:^{-} \&\{l_j : R_j\}_{i \in J}} \qquad \frac{J \subseteq I \quad (S_j <:^{+} R_j)_{j \in J}}{\oplus\{l_i : S_i\}_{i \in I} <:^{+} \oplus\{l_j : R_j\}_{j \in J}}$$

$$\frac{I \subseteq J \quad (S_i <:^{+} R_i)_{i \in I}}{\&\{l_i : S_i\}_{i \in I} <:^{+} \&\{l_j : R_j\}_{i \in J}} \qquad \mathsf{end}_! <:^{\pm} \mathsf{end}_! \qquad \mathsf{end}_? <:^{\pm} \mathsf{end}_?$$

Naive subtyping $\boxed{T <:_{\mathsf{n}} U}$

$$T <:_{\mathsf{n}} \star \qquad S <:_{\mathsf{n}} \circledast$$

$$\mathsf{unit} <:_{\mathsf{n}} \mathsf{unit} \qquad \frac{T <:_{\mathsf{n}} T' \quad U <:_{\mathsf{n}} U' \quad m <: n}{T \to_m U <:_{\mathsf{n}} T' \to_n U'} \qquad \frac{T <:_{\mathsf{n}} T' \quad U <:_{\mathsf{n}} U' \quad m <: n}{T \times_m U <: T' \times_n U'}$$

$$\frac{T <:_{\mathsf{n}} T' \quad S <:_{\mathsf{n}} S'}{!T.S <:_{\mathsf{n}} !T'.S'} \qquad \frac{T <:_{\mathsf{n}} T' \quad S <:_{\mathsf{n}} S'}{?T.S <:_{\mathsf{n}} ?T'.S'} \qquad \frac{J \subseteq I \quad (S_j <:_{\mathsf{n}} R_j)_{j \in J}}{\oplus\{l_i : S_i\}_{i \in I} <:_{\mathsf{n}} \oplus\{l_j : R_j\}_{j \in J}}$$

$$\frac{I \subseteq J \quad (S_i <:_{\mathsf{n}} R_i)_{i \in I}}{\&\{l_i : S_i\}_{i \in I} <:_{\mathsf{n}} \&\{l_j : R_j\}_{j \in J}} \qquad \mathsf{end}_! <:_{\mathsf{n}} \mathsf{end}_! \qquad \mathsf{end}_? <:_{\mathsf{n}} \mathsf{end}_?$$

Blame safety $\boxed{e \text{ safe for } p}$

$$\frac{e \text{ safe for } p \quad T <:^{+} U}{e : T \overset{p}{\Rightarrow} U \text{ safe for } p} \qquad \frac{e \text{ safe for } \overline{p} \quad T <:^{-} U}{e : T \overset{p}{\Rightarrow} U \text{ safe for } \overline{p}} \qquad \frac{e \text{ safe for } p \quad q \neq p \quad q \neq \overline{p}}{e : T \overset{q}{\Rightarrow} U \text{ safe for } p}$$

$$\frac{q \neq p \quad q' \neq p}{\mathsf{blame}\ q\ q'\ X \text{ safe for } p} \qquad \frac{q \neq p}{\mathsf{blame}\ q\ X \text{ safe for } p}$$

Fig. 9. Subtyping and blame safety

The four subtyping relations are closely related. Proper subtyping decomposes into positive and negative subtyping, which (after reversing the order on negative subtyping) recompose into naive subtyping, analogous to a tangram.

Theorem 4.12 (Tangram).

(1) *$T <: U$ if and only if $T <:^{+} U$ and $T <:^{-} U$.*
(2) *$T <:_n U$ if and only if $T <:^{+} U$ and $U <:^{-} T$.*

Proof. By induction on types. □

The following technical result is used in the proof of theorem 4.14.

Lemma 4.13.

(1) *If $T \neq \star$ and $T \sim \boldsymbol{T}$, then $T <:^{+} \boldsymbol{T}$.*
(2) *If $S \neq \circledast$ and $S \sim \boldsymbol{S}$, then $S <:^{+} \boldsymbol{S}$.*

Proof. (1) A case analysis on T. Lemma 4.8 tells us that $\mathbf{T}$ is unique. We show the case for functions. Let T be the type $U \to_m V$; we know that $\mathbf{T}$ is $\star \to_m \star$, that $\star <:^- U$, and $V <:^+ \star$. Conclude with the positive subtyping rule rule for functions. (2) Similar. □

We say that a process P is *safe* for blame label p, if all occurrences of casts involving p or $\overline{p}$ correspond to subsumptions in the blame subtyping relation. Figure 9 defines a judgment e safe for p, extended homomorphically to all other forms of expressions and processes. The safe for predicate on well-typed programs is preserved by reduction.

Theorem 4.14 (Preservation of safe terms). *If* $\Gamma \vdash P$ *with* P safe for p *and* $P \longrightarrow Q$, *then* Q safe for p.

Proof. It is sufficient to examine all reductions whose contractum involves coercions. We start with the reductions in Figure 6. The four rules starting from the one with reductum $v : T \overset{p}{\Rightarrow} \star$ follow from Lemma 4.13. Then, the standard function cast is analogous to Wadler and Findler [2009], and the case for pairs is similar. The casts for session types (send, receive, select, case, close, and wait) are new; we concentrate on send.

$$\mathsf{send}\, v\, (w : !T.S \overset{p}{\Rightarrow} !T'.S') \longrightarrow (\mathsf{send}\, (v : T' \overset{\overline{p}}{\Rightarrow} T)\, w) : S \overset{p}{\Rightarrow} S'$$

By assumption $(w : !T.S \overset{p}{\Rightarrow} !T'.S')$ safe for p. Inversion of the safe for relation yields $T' <:^{\mp} T$ and $S <:^{\pm} S'$. Hence $(v : T' \overset{\overline{p}}{\Rightarrow} T)$ safe for p and $(\ldots) : S \overset{p}{\Rightarrow} S'$ safe for p. Finally, all rules in Figure 7 preserve casts. □

A process P blames label p if $P \equiv \Pi(\langle e \rangle \mid Q)$ where e is blame $p\ q\ X$, blame $q\ p\ X$, or blame $p\ X$, for some q and X, and prefix Π of bindings for channel endpoints and linear references.

Theorem 4.15 (Progress of safe terms). *If* $\Gamma \vdash P$ *and* P safe for p, *then* $P \not\longrightarrow Q$ *where* Q *blames* p.

Proof. We analyse all reduction rules whose contractum includes blame. From Figure 6 take the rule with reductum $(v : \mathbf{T} \overset{p}{\Rightarrow} \star) : \star \overset{q}{\Rightarrow} \mathbf{U}$. It may blame $\overline{p}$ and q, if $\mathbf{T} \not<: \mathbf{U}$. However, if it is safe for $\overline{p}$ then $\mathbf{T} <:^- \star$, which cannot hold (because only $\star <:^- \star$ and $\mathbf{T}$ cannot be $\star$), and similar reasoning applies for q and $\mathbf{U}$. The remaining rules are similar. □

We are finally in a position to state the main result of this section.

Corollary 4.16 (Well-typed programs can't be blamed). *Let* P *be a well-typed process with a subterm of the form* $e : T \overset{p}{\Rightarrow} U$ *containing the only occurrence of* p *and* $\overline{p}$ *in* P. *Then:*

- *If* $T <:^+ U$ *then* $P \not\longrightarrow^* Q$ *where* Q *blames* p.
- *If* $T <:^- U$ *then* $P \not\longrightarrow^* Q$ *where* Q *blames* $\overline{p}$.
- *If* $T <: U$ *then* $P \not\longrightarrow^* Q$ *where* Q *blames* p *or* $\overline{p}$.

For example, the redex $(v : \mathbf{T} \overset{p}{\Rightarrow} \star) : \star \overset{q}{\Rightarrow} \mathbf{U}$ may fail and blame $\overline{p}$ and q if $\mathbf{T} \not<: \mathbf{U}$. And indeed we have that $\mathbf{T} \not<:^- \star$ and $\star \not<:^+ \mathbf{U}$, so it is not safe for $\overline{p}$ or q. However, $\mathbf{T} <:^+ \star$ and $\star <:^- \mathbf{U}$, and the redex will not blame p or $\overline{q}$.

5 RELATED WORK

Languages mentioned in the introduction include Agda [Norell 2009], Coq [Chlipala 2013], Idris [Brady 2013], Eff [Bauer and Pretnar 2015], Frank [Lindley et al. 2017], Koka [Leijen 2017], Links [Lindley and Morris 2016b], Scribble [Yoshida et al. 2014], and Singularity OS [Fähndrich et al. 2006].

5.1 Gradual Typing

Findler and Felleisen [2002] introduced two seminal ideas: higher-order *contracts* that dynamically monitor conformance to a type discipline, and *blame* to indicate whether it is the library or the client which is at fault if the contract is violated. Siek and Taha [2006] introduced gradual types to integrate untyped and typed code, while Flanagan [2006] introduced hybrid types to integrate simple types with refinement types. Both used target languages with explicit casts and similar translations from source to target; both exploit contract, but neither allocates blame. Motivated by similarities between gradual and hybrid types, Wadler and Findler [2009] introduced blame calculus, which unifies the two by encompassing untyped, simply-typed, and refinement-typed code. As the name indicates, it also restores blame, which enables a proof of *blame safety*: blame for type errors always lays with less-precisely typed code—"well-typed programs can't be blamed".

Siek et al. [2015] review desirable properties of gradually-typed languages, while Wadler [2015] discusses the history of the blame calculus and why blame is important. These papers provide overviews of the field, each with many further citations.

As noted in the introduction, gradual typing may be important as a bridge to type systems that go beyond what is currently available, including dependent, effect, and session types. Gradual type systems for dependent types include Ou et al. [2004], Flanagan [2006], and Greenberg et al. [2010]. A gradual type systems for effect types was explored by Bañados Schwerter et al. [2014]. Gradual type systems related to session types include the run-time enforcement of affine typing of Tov and Pucella [2010], and the gradual typestate of Wolff et al. [2011]. Thiemann [2014] describes a system with gradual types and session types, but in it only types (and not session types) can be gradual.

Practical languages exploiting gradual types include type Dynamic in C# [Bierman et al. 2010], TypeScript [Bierman et al. 2014], Hack [Verlaguet 2013], and Dart [Team 2014].

TypeScript TPD [Williams et al. 2017] applies contracts to monitor the gradual typing of TypeScript, and evaluates the successes and shortcomings of contracts in this context.

5.2 Session Types

Session types were introduced by Honda [1993]; Honda et al. [1998]. The original system addressed binary sessions, whereby types describe the interaction between two partners. Binary sessions were eventually extended to the more general setting of multiparty session types [Honda et al. 2016]. Recent years have seen the introduction of session types in programming languages, and software development tools. We review the most important works.

Session types inspired the design of several programming languages. Sing# [Fähndrich et al. 2006] constitutes one of the first attempts to introduce session types in programming languages. An extension of C, Sing# was used to implement Singularity, an operating system based on message passing. [Gay et al. 2010] propose attaching session types to class definitions, allowing to treat channels as objects for session-based communication in distributed systems. SePi [Franco and Vasconcelos 2013] is a concurrent, message-passing programming language based on the pi-calculus, featuring a simple form of refinement types. SILL [Pfenning and Griffith 2015; Toninho et al. 2013] is a higher-order session functional programming language, featuring process expressions as first class objects via a linear contextual monad. Concurrent C0 [Willsey et al. 2017] is a type-safe C-like programming language equipped with channel communication governed by session types. Links [Lindley and Morris 2017] is a functional programming language designed for tierless web applications that natively supports binary session types.

Proposals have been made to retroactively introduce session types in mainstream programming languages. Session Java [Hu et al. 2008] introduces API-based session primitives in Java, while [Hu et al. 2010] presents a Java language extension and type discipline for session-based event-driven

programming. Featherweight Erlang [Mostrous and Vasconcelos 2011] imposes a session-based type system to discipline message passing in Erlang. Mungo [Kouzapas et al. 2016] is a tool for checking Java code against session types, presented in the form of typestates. Embedding of session types have been proposed for Haskell [Lindley and Morris 2016a; Orchard and Yoshida 2016; Polakow 2015; Pucella and Tov 2008; Sackman and Eisenbach 2008], OCaml [Padovani 2017], Scala [Scalas and Yoshida 2016], and Rust [Jespersen et al. 2015]. Most of these embeddings delegate linearity checks on the run-time system.

Session types can be used in the software development process under different forms, including languages to describe protocols, specialised libraries to invoke session-based communication primitives, provision for run-time monitoring against session types, and extended type checkers. Scribble [Honda et al. 2011] is a language-agnostic protocol description formalism used in many different tools. Multiparty Session C [Ng et al. 2012] uses Scribble, a compiler plug-in, and a C library to validate against session types. Hu and Yoshida [2016] generate protocol-specific Java APIs from multiparty session types described in Scribble. SPY [Neykova et al. 2013] generates run-time monitors for endpoint communication from Scribble protocols. Neykova and Yoshida [2014] design and implemented a session actor library in Python together with a run-time verification mechanism. Bocchi et al. [2017] present a theory that incorporates both static typing and dynamic monitoring of session types. Fowler [2016] describes a framework for monitoring Erlang applications against multiparty session types. Neykova and Yoshida [2017] investigate failure handling for Erlang processes in a system that dynamically monitors session types. Dingo Hunter [Ng and Yoshida 2016] and Gong [Lange et al. 2017] statically detect (partial) deadlocks in Go programs by extracting behavioural types from programs.

6 CONCLUSIONS

We presented the design of Gradual GV, which combines a session-typed language GV along the lines of Gay and Vasconcelos [2010] with a blame calculus along the lines of Wadler and Findler [2009], and with dynamic enforcement of linearity along the lines of Tov and Pucella [2010]. We established the expected results for such a language, including type safety and blame safety.

Much remains to be done; we consider just one future direction here. The embedding of linear types in the unrestricted dynamic type relies on an indirection through a cell in the store. In our present work, these cells are used once and then discarded. This *one-shot policy* imposes a certain usage pattern on linear values embedded in the unityped language. In particular, the send and receive operations on a channel need to be chained as in $(\mathsf{close}\ (\mathsf{send}\ v_2\ (\mathsf{send}\ v_1\ c)))$. However, one could imagine a unityped language where one may use the channel non-linearly in an imperative style as in $(\mathsf{send}\ v_1\ c; \mathsf{send}\ v_2\ c; \mathsf{close}\ c)$, mimicking the style of network programming in conventional languages. This style can also be supported by a variant of Gradual GV with a *multi-shot policy* that restores an updated channel to the same cell from which it was extracted. We leave the full formalization of this policy to future work.

ACKNOWLEDGMENTS

We would like to thank Alceste Scalas and Nobuko Yoshida for comments and pointing out errors in the definition of subtyping rules, Taro Sekiyama and anonymous reviewers for valuable comments. We are grateful to Hannes Saffrich for implementing a type checker for the calculus in this paper. This work was supported in part by the JSPS KAKENHI Grant Number JP17H01723 (Igarashi) by FCT through the LASIGE Research Unit, ref. UID/CEC/00408/2013 (Vasconcelos), and by EPSRC programme grant EP/K034413/1 (Wadler).

REFERENCES

Felipe Bañados Schwerter, Ronald Garcia, and Éric Tanter. 2014. A theory of gradual effect systems. In *International Conference on Functional Programming (ICFP)*. ACM, 283–295.

H.P. Barendregt. 1984. *The Lambda Calculus: Its Syntax and Semantics*. North-Holland.

Andrej Bauer and Matija Pretnar. 2015. Programming with algebraic effects and handlers. *Journal of Logical and Algebraic Methods in Programming* 84, 1 (2015), 108–123.

Gavin Bierman, Martin Abadi, and Mads Torgersen. 2014. Understanding TypeScript. In *European Conference on Object-Oriented Programming (ECOOP) (LNCS)*, Vol. 8586. Springer, 257–281.

Gavin M. Bierman, Erik Meijer, and Mads Torgersen. 2010. Adding dynamic types to C#. In *European Conference on Object-Oriented Programming (ECOOP) (LNCS)*. Springer, 76–100.

Laura Bocchi, Tzu-Chun Chen, Romain Demangeon, Kohei Honda, and Nobuko Yoshida. 2017. Monitoring networks through multiparty session types. *Theoretical Computer Science* 699 (2017), 33–58.

Edwin Brady. 2013. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming* 23, 05 (2013), 552–593.

Luís Caires and Frank Pfenning. 2010. Session types as intuitionistic linear propositions. In *International Conference on Concurrency Theory (CONCUR) (LNCS)*. Springer, 222–236.

Luis Caires, Frank Pfenning, and Bernardo Toninho. 2014. Linear logic propositions as session types. *Mathematical Structures in Computer Science* 26, 03 (2014), 367–423.

Adam Chlipala. 2013. *Certified Programming with Dependent Types: A Pragmatic Introduction to the Coq Proof Assistant*. MIT Press.

Manuel Fähndrich, Mark Aiken, Chris Hawblitzel, Orion Hodson, Galen C. Hunt, James R. Larus, and Steven Levi. 2006. Language support for fast and reliable message-based communication in Singularity OS. In *European Conference on Computer Systems (EuroSys)*. ACM, 177–190.

Luminous Fennell and Peter Thiemann. 2012. The Blame Theorem for a Linear Lambda Calculus with Type Dynamic. In *Trends in Functional Programming (LNCS)*, Vol. 7829. Springer, 37–52.

Robert Bruce Findler and Matthias Felleisen. 2002. Contracts for Higher-Order Functions. In *International Conference on Functional Programming (ICFP)*. ACM, 48–59.

Cormac Flanagan. 2006. Hybrid Type Checking. In *Principles of Programming Languages (POPL)*. ACM, 245–256.

Simon Fowler. 2016. An Erlang Implementation of Multiparty Session Actors. In *Interaction and Concurrency Experience*. 36–50.

Juliana Franco and Vasco Thudichum Vasconcelos. 2013. A Concurrent Programming Language with Refined Session Types. In *SEFM (LNCS)*, Vol. 8368. Springer, 15–28.

Ronald Garcia, Alison M. Clark, and Éric Tanter. 2016. Abstracting gradual typing. In *Principles of Programming Languages (POPL)*. ACM, 429–442.

Simon Gay and Malcolm Hole. 2005. Subtyping for session types in the pi calculus. *Acta Informatica* 42, 2-3 (2005), 191–225.

Simon Gay and Vasco Vasconcelos. 2010. Linear type theory for asynchronous session types. *Journal of Functional Programming* 20, 01 (2010), 19–50.

Simon J. Gay, Vasco Thudichum Vasconcelos, António Ravara, Nils Gesbert, and Alexandre Z. Caldeira. 2010. Modular session types for distributed object-oriented programming. In *Principles of Programming Languages (POPL)*. ACM, 299–312.

Jean-Yves Girard. 1987. Linear logic. *Theoretical computer science* 50, 1 (1987), 1–101.

Michael Greenberg, Benjamin C. Pierce, and Stephanie Weirich. 2010. Contracts Made Manifest. In *Principles of Programming Languages (POPL)*. ACM, 353–364.

Kohei Honda. 1993. Types for Dyadic Interaction. In *International Conference on Concurrency Theory (CONCUR) (LNCS)*, Vol. 715. Springer, 509–523.

K. Honda, A. Mukhamedov, G. Brown, T. Chen, and N. Yoshida. 2011. Scribbling Interactions with a Formal Foundation. In *ICDCIT (LNCS)*, Vol. 6536. Springer, 55–75.

Kohei Honda, Vasco Vasconcelos, and Makoto Kubo. 1998. Language Primitives and Type Discipline for Structured Communication-Based Programming. In *European Symposium on Programming (ESOP) (LNCS)*. Springer, 122–138.

Kohei Honda, Nobuko Yoshida, and Marco Carbone. 2008. Multiparty asynchronous session types. In *Principles of Programming Languages (POPL)*. ACM, 273–284.

Kohei Honda, Nobuko Yoshida, and Marco Carbone. 2016. Multiparty asynchronous session types. *J. ACM* 63, 1 (2016), 9.

Raymond Hu, Dimitrios Kouzapas, Olivier Pernet, Nobuko Yoshida, and Kohei Honda. 2010. Type-safe eventful sessions in Java. In *European Conference on Object-Oriented Programming (ECOOP) (LNCS)*, Vol. 6183. Springer, 329–353.

Raymond Hu and Nobuko Yoshida. 2016. Hybrid session verification through endpoint API generation. In *Fundamental Approaches to Software Engineering (FASE) (LNCS)*, Vol. 9633. Springer, 401–418.

Raymond Hu, Nobuko Yoshida, and Kohei Honda. 2008. Session-based distributed programming in Java. In *European Conference on Object-Oriented Programming (ECOOP) (LNCS)*, Vol. 5142. Springer, 516–541.

Thomas Bracht Laumann Jespersen, Philip Munksgaard, and Ken Friis Larsen. 2015. Session types for Rust. In *Workshop on Generic Programming (WGP)*. ACM, 13–22.

Dimitrios Kouzapas, Ornela Dardha, Roly Perera, and Simon J. Gay. 2016. Typechecking protocols with Mungo and StMungo. In *Principles and Practice of Declarative Programming (PPDP)*. ACM, 146–159.

Julien Lange, Nicholas Ng, Bernardo Toninho, and Nobuko Yoshida. 2017. Fencing off Go: liveness and safety for channel-based programming. In *Principles of Programming Languages (POPL)*. ACM, 748–761.

Daan Leijen. 2017. Type Directed Compilation of Row-typed Algebraic Effects. *SIGPLAN Not.* 52, 1, 486–499.

Sam Lindley, Conor McBride, and Craig McLaughlin. 2017. Do Be Do Be Do. In *Principles of Programming Languages (POPL)*. ACM, 500–514.

Sam Lindley and J. Garrett Morris. 2016a. Embedding session types in Haskell. In *Symposium on Haskell*. ACM, 133–145.

Sam Lindley and J Garrett Morris. 2016b. Talking bananas: structural recursion for session types. In *International Conference on Functional Programming (ICFP)*. ACM, 434–447.

Sam Lindley and J. Garrett Morris. 2017. *Behavioural Types: from Theory to Tools*. River Publishers, Chapter Lightweight functional session types.

Robin Milner, Joachim Parrow, and David Walker. 1992. A calculus of mobile processes, I. *Information and Computation* 100, 1 (1992), 1–40.

Dimitris Mostrous and Vasco T. Vasconcelos. 2011. Session typing for a featherweight Erlang. In *Coordination Models and Languages (COORDINATION) (LNCS)*, Vol. 6721. Springer, 95–109.

Rumyana Neykova and Nobuko Yoshida. 2014. Multiparty session actors. In *Coordination Models and Languages (COORDINATION) (LNCS)*, Vol. 8459. Springer, 131–146.

Rumyana Neykova and Nobuko Yoshida. 2017. Let it recover: multiparty protocol-induced recovery. In *International Conference on Compiler Construction (CC)*. ACM, 98–108.

Rumyana Neykova, Nobuko Yoshida, and Raymond Hu. 2013. SPY: Local Verification of Global Protocols. In *International Conference on Runtime Verification (RV) (LNCS)*, Vol. 8174. Springer, 358–363.

Nicholas Ng and Nobuko Yoshida. 2016. Static deadlock detection for concurrent Go by global session graph synthesis. In *International Conference on Compiler Construction (CC)*. ACM, 174–184.

Nicholas Ng, Nobuko Yoshida, and Kohei Honda. 2012. Multiparty Session C: Safe parallel programming with message optimisation. In *International Conference on Modelling Techniques and Tools for Computer Performance Evaluation (TOOLS) (LNCS)*, Vol. 7304. Springer, 202–218.

Ulf Norell. 2009. Dependently typed programming in Agda. In *Proceedings of the 4th International Workshop on Types in Language Design and Implementation (TLDI '09)*. ACM, 1–2.

Dominic Orchard and Nobuko Yoshida. 2016. Effects as sessions, sessions as effects. In *Principles of Programming Languages (POPL)*. ACM, 568–581.

Xinming Ou, Gang Tan, Yitzhak Mandelbaum, and David Walker. 2004. Dynamic Typing with Dependent Types. In *IFIP International Conference on Theoretical Computer Science*, Vol. 155. Springer, 437–450.

Luca Padovani. 2017. A simple library implementation of binary sessions. *Journal of Functional Programming* 27 (2017), e4.

Frank Pfenning and Dennis Griffith. 2015. Polarized substructural session types. In *International Conference on Foundations of Software Science and Computation Structures (LNCS)*, Vol. 9034. Springer, 3–22.

Jeff Polakow. 2015. Embedding a full linear Lambda calculus in Haskell. In *Symposium on Haskell*. ACM, 177–188.

Riccardo Pucella and Jesse A Tov. 2008. Haskell session types with (almost) no class. In *Symposium on Haskell*. ACM, 25–36.

Matthew Sackman and Susan Eisenbach. 2008. Session types in Haskell: Updating message passing for the 21st century. (2008).

Alceste Scalas and Nobuko Yoshida. 2016. Lightweight session programming in Scala. In *European Conference on Object-Oriented Programming (ECOOP) (LIPIcs)*. Schloss Dagstuhl, 21:1–21:28.

Jeremy G. Siek and Walid Taha. 2006. Gradual Typing for Functional Languages. In *Scheme and Functional Programming Workshop (Scheme)*. 81–92.

Jeremy G. Siek, Michael M. Vitousek, Matteo Cimini, and John T. Boyland. 2015. Refined criteria for gradual typing. In *Summit on Advances in Programming Languages (SNAPL) (LIPIcs)*, Vol. 32. Schloss Dagstuhl, 274–293.

The Dart Team. 2014. Dart Programming Lanuage Specification. (2014). Google, 1.2 edition.

Peter Thiemann. 2014. Session Types with Gradual Typing. In *TGC (LNCS)*, Vol. 8902. Springer, 144–158.

Bernardo Toninho, Luís Caires, and Frank Pfenning. 2013. Higher-Order Processes, Functions, and Sessions: A Monadic Integration. In *European Symposium on Programming (ESOP) (LNCS)*, Vol. 7792. Springer, 350–369.

Jesse A. Tov and Riccardo Pucella. 2010. Stateful contracts for affine types. In *European Symposium on Programming (ESOP) (LNCS)*, Vol. 6012. Springer, 550–569.

Vasco Thudichum Vasconcelos. 2012. Fundamentals of Session Types. *Information and Computation* 217 (2012), 52–70.

Julien Verlaguet. 2013. Facebook: Analysing PHP Statically. In *Workshop on Commercial Uses of Functional Programming (CUFP)*.

Philip Wadler. 2012. Propositions as sessions. In *International Conference on Functional Programming (ICFP)*. ACM, 273–286.

Philip Wadler. 2014. Propositions as sessions. *Journal of Functional Programming* 24, 2-3 (2014), 384–418.

Philip Wadler. 2015. A Complement to Blame. In *1st Summit on Advances in Programming Languages (SNAPL) (LIPIcs)*, Vol. 32. Schloss Dagstuhl, 309–320.

Philip Wadler and Robert Bruce Findler. 2009. Well-Typed Programs Can't Be Blamed. In *European Symposium on Programming (ESOP) (LNCS)*, Vol. 5502. Springer, 1–16.

David Walker. 2005. *Advanced Topics in Types and Programming Languages*. MIT Press, Chapter Substructural Type Systems, 3–43.

Jack Williams, J. Garrett Morris, Philip Wadler, and Jakub Zalewski. 2017. Mixed Messages: Measuring Conformance and Non-Interference in TypeScript. In *European Conference on Object-Oriented Programming (ECOOP) (LIPIcs)*, Vol. 74. Schloss Dagstuhl, Dagstuhl, Germany, 28:1–28:29.

Max Willsey, Rokhini Prabhu, and Frank Pfenning. 2017. Design and Implementation of Concurrent C0. In *International Workshop on Linearity (EPTCS)*, Vol. 238. 73–82.

Roger Wolff, Ronald Garcia, Éric Tanter, and Jonathan Aldrich. 2011. Gradual typestate. In *European Conference on Object-Oriented Programming (ECOOP) (LNCS)*, Vol. 6813. Springer, 459–483.

Nobuko Yoshida, Raymond Hu, Rumyana Neykova, and Nicholas Ng. 2014. The Scribble protocol language. In *International Symposium on Trustworthy Global Computing (LNCS)*, Vol. 8358. Springer, 22–41.

Nobuko Yoshida and Vasco Vasconcelos. 2007. Language primitives and type discipline for structured communication-based programming revisited: Two systems for higher-order session communication. *ENTCS* 171, 4 (2007), 73–93.