JFP **29**, e17, 56 pages, 2019. © The Author(s) 2019. This is an Open Access article, distributed under the terms of the Creative Commons Attribution licence (http://creativecommons.org/licenses/by/4.0/), which permits unrestricted re-use, distribution, and reproduction in any medium, provided the original work is properly cited.
doi:10.1017/S0956796819000169

Gradual session types

ATSUSHI IGARASHI
Kyoto University, Japan
(e-mail: igarashi@kuis.kyoto-u.ac.jp)

PETER THIEMANN
University of Freiburg, Germany
(e-mail: thiemann@informatik.uni-freiburg.de)

YUYA TSUDA
Kyoto University, Japan
(e-mail: tsuda@fos.kuis.kyoto-u.ac.jp)

VASCO T. VASCONCELOS
LASIGE, Department of Informatics, Faculty of Sciences, University of Lisbon, Portugal
(e-mail: vv@di.fc.ul.pt)

PHILIP WADLER
University of Edinburgh, Scotland
(e-mail: wadler@inf.ed.ac.uk)

Abstract

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 a gradually typed extension of the functional session type system GV. Following a standard framework of gradual typing, Gradual GV consists of an external language, which relaxes the type system of GV using dynamic types; an internal language with casts, for which operational semantics is given; and a cast-insertion translation from the former to the latter. 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.

1 Introduction

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

A survey of the top-20 programming languages to learn for open source projects[1] lists eight dynamically typed languages (JavaScript, Python, Ruby, R, PHP, Perl, Scheme, and

[1] https://www.ubuntupit.com/top-20-most-popular-programming-languages-to-learn-for-your-open-source-project/ accessed in April 2019.

https://doi.org/10.1017/S0956796819000169 Published online by Cambridge University Press

Erlang) and states that developers' salaries for these languages are among the highest in the industry. The survey also suggests to learn languages with elaborate static type systems like Rust, Scala, and Haskell, with developers earning even higher salaries. These languages derive their expressiveness from advanced type system features like linearity, uniqueness, effects, and dependent types as embodied in research languages like Agda (Norell, 2009), Coq (The Coq Development Team, 2019), and Idris (Brady, 2013); and session types as in Links (Lindley & Morris, 2016b). These data indicate two opposing trends in current industrial practice, one asking for dynamically typed programming and another asking for expressive statically typed programming.

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 (Findler & Felleisen, 2002) and their interfacing with TypedRacket (Tobin-Hochstadt & Felleisen, 2008) and then popularized by Siek and others (Siek & Taha, 2006, 2007; Siek *et al.*, 2015b). They are geared toward safely interconnecting dynamically typed parts with statically typed parts of a program by ensuring that type mismatches only occur in the dynamically typed parts (Wadler & Findler, 2009).

Dynamics in C# (Bierman *et al.*, 2010), Microsoft's TypeScript[2] (Bierman *et al.*, 2014), Google's Dart (The Dart Team, 2014; Ernst *et al.*, 2017), and Facebook's Hack (Verlaguet, 2013) and Flow (Chaudhuri *et al.*, 2017) are industrial systems inspired by gradual typing, but focusing on enhancing programmer productivity and bug finding rather than containing type mismatches. Systems such as Racket (Findler & Felleisen, 2002) and Reticulated Python (Vitousek *et al.*, 2017) 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.

At first blush, one might consider gradual types as largely a response to the former trend: they provide a way for developers using dynamically typed languages to evolve their code toward statically typed languages that are deemed easier to maintain. But on second thought, one might consider gradual types as even more helpful in light of the latter trend. Suitably generalized, gradual typing can mediate between simple type systems and type systems that feature dependent types, effect types, or session types, for example. Gradual typing in this sense can help in evolving software development toward languages with more precise type systems.

Hence, an important line of research is to extend gradual typing so that it not only relates dynamically typed and statically typed languages, but also relates less-precisely typed and more-precisely typed languages. There is already some research on doing so for dependent types (Ou *et al.*, 2004; Flanagan, 2006; Greenberg *et al.*, 2010; Lehmann & Tanter, 2017), effect types (Bañados Schwerter *et al.*, 2014), typestate (Wolff *et al.*, 2011), and several others which we review in the section on related work. This paper presents the first system that extends gradual typing to session types.

Session types were introduced by Honda (1993), drawing on Milner's π-calculus (Milner *et al.*, 1992) and Girard's linear logic (1987), and further developed by many others

[2] `https://www.typescriptlang.org/` accessed in April 2019.

https://doi.org/10.1017/S0956796819000169 Published online by Cambridge University Press

(Honda *et al.*, 1998; Yoshida & Vasconcelos, 2007). Gay and Hole (2005) introduced subtyping for session types, 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). Caires, Pfenning, Toninho, and Wadler introduced propositions-as-types interpretations of session types in linear logic (Caires & Pfenning, 2010; Wadler, 2012, 2014; Caires *et al.*, 2014). One important line of research is multiparty session types (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 (Willsey *et al.*, 2017), Links (Cooper *et al.*, 2007), SePi (Franco & Vasconcelos, 2013), SILL (Pfenning & Griffith, 2015), and Singularity (Fähndrich *et al.*, 2006). Industrial uses of session types include Red Hat's support of the Scribble specification language (Yoshida *et al.*, 2014), which has been used as a common interface for several systems based on session types; Estafet's use of session types to manage microservices[3]; and the Ocean Observatories Initiative's use of dynamically checked session types in Python (Demangeon *et al.*, 2015). 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 swapped 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 & Pucella (2010), we require that each dynamically typed reference to a channel endpoint is equipped with a lock. That reference is locked after the channel is used once to ensure it cannot be used again. To ensure that each channel is appropriately terminated, with either a

[3] `http://estafet.com/scribble/` Accessed in April 2019.

https://doi.org/10.1017/S0956796819000169 Published online by Cambridge University Press

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 via gradual typing. It preserves the safety properties of statically typed sessions, namely progress (for expressions), preservation, and absence of runtime 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.

Previous systems that perform dynamic monitoring on session types include the work on Scribble (Yoshida *et al.*, 2014) which applies the ideas developed for distributed monitoring of protocols to multiparty session types (Bocchi *et al.*, 2013, 2017). Gommerstadt and others (2016) consider dynamic monitoring of higher-order session-typed processes in the presence of unreliable communication and malicious communication partners. Their focus is on assigning blame correctly in this setting. The same authors (Gommerstadt *et al.*, 2018) develop a theory of contracts that translate into processes that serve as proxies between the original communication partners. Proxies ensure adherence to the session protocol with dynamic tests. A similar proxy-based monitoring scheme was also proposed by one of the authors (Thiemann, 2014) where gradual typing was restricted to the transmitted values. Melgratti & Padovani (2017) propose a contract system that mediates between (simply typed) sessions and contract-refined sessions. Enforcement is done with an inline monitor.

In contrast to these approaches, our work applies to the mediation between dynamically typed and statically typed code and it relies on gradual principles that enable a pay-as-you-go approach: a protocol is checked statically as much as possible, dynamic checks are only employed if they cannot be avoided. Our work is the first to enable full gradualization that includes the typing of the communication channel rather than just the values transmitted. Moreover, we improve the efficiency over prior approaches by avoiding the introduction of extra proxy processes.

We give our system a compact formulation along the lines of the blame calculus (Wadler & 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 four subtyping relations exhibited by the blame calculus, ordinary, positive, negative, and naive, and show the corresponding 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 of both an external language, in which programs are written and runtime checking is implicit, and an internal language, in which programs are executed after runtime checking in the form of casts is made explicit; typing rules of the two languages; reduction rules for the internal language; cast-insertion translation from the external to the internal

https://doi.org/10.1017/S0956796819000169 Published online by Cambridge University Press

language; and embedding of a dynamically typed language with channel-based communication into our calculus.

- Section 4 presents standard results for our calculus, including progress (for expressions), preservation, session fidelity, the tangram theorem, blame safety, conservativity of the external language typing over fully static typing, and type preservation of the cast insertion translation. We also discuss the gradual guarantee property for the external language. It turns out that it fails to hold—we will analyze counterexamples and discuss why.
- Section 5 describes related work and Section 6 concludes.

Compared to the previous paper (Igarashi *et al.*, 2017a), we extend the development with the external language, the cast-insertion translation, a type-checking algorithm, proofs of their properties, and analysis of the failure of the gradual guarantee, as well as more detailed proofs for the earlier results. These extensions make gradual session types accessible for the programmer, who works in the external language.

2 Motivation

Sy and Rob collaborate on a project whose design is based 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 microservice 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 runtime. Here is the story how they can collaborate safely using Gradual GV,[4] our proposal for a gradually typed functional language with synchronous binary session types.

2.1 A compute service

The compute service is a simplified version of one of the protocols in Sy and Rob's project. The 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 server's view of the compute protocol reads as follows:

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

Sy chooses to implement the server in the language GV that is inspired by previous work (Gay & Vasconcelos, 2010) and that we will describe formally in Section 3.

[4] GV is our name for the functional session type calculus of Gay and Vasconcelos (2010), which is the statically typed baseline for our gradual system.

https://doi.org/10.1017/S0956796819000169 Published online by Cambridge University Press

```
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. The type of c is updated to the 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 with a correspondingly depleted session type. 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 defined in the following. This protocol is *dual* to Compute: sending and receiving operations are swapped.

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

A client of the compute service may always select the same operation and then proceed linearly according the corresponding branch. Such a client can use a simpler supertype of ComputeD with a unary internal choice. For example, a client that only ever asks for negation can implement ComputeDneg.

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

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

```
negationClient : int → ComputeDneg → int
negationClient v c =
  let c = select neg c in
  let c = send v 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

https://doi.org/10.1017/S0956796819000169 Published online by Cambridge University Press

selected branch of the protocol with type $!\mathsf{int}.?\mathsf{int}.\mathsf{end}_?$. 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 safe but does not impose a 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 takes a channel c on which it receives the client's selection. It 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. 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?

One answer would be to use a dependently typed system that can describe the type of the serveOp function adequately. In an extension of a recently proposed system (Toninho & Yoshida, 2018) with iteration on natural numbers and large elimination, we might write that code as follows:

```
Op : nat → Type
Op 0 = int
Op (n+1) = int → Op n

Ch : nat → Session
Ch 0 = !int.end!
Ch (n+1) = ?int.Ch n
```

https://doi.org/10.1017/S0956796819000169 Published online by Cambridge University Press

```
serveOpDep : (n : nat) (op : Op n) (c : Ch n) → unit
serveOpDep 0 op c = close (send op c)
serveOpDep (n+1) op c = let v,c = receive c in
                        serveOpDep n (op v) c
```

However, we are not aware of a fully developed theory of a session-type system that would be able to process this definition.

An alternative that is immediately available is to resort to gradual typing. For this particular program, it will insert casts to make the program type check, but all those casts are semantically guaranteed to succeed because it would have a dependent type. To this end, we rewrite the function dynServer in a gradually typed external language analogous to the gradually typed lambda calculus GTLC (Siek *et al.*, 2015b), but extended with GV's communication operations.

In our example, the rewrite to the external language boils down to providing suitable type signatures for dynServer and serveOp:

```
dynServer : Compute → unit
serveOp : int → ⋆ → ⊛ → unit
```

The first argument n of dynServer 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**: these types are subsumed to the dynamic type ⋆. Similarly to other gradual type systems, an expression of type ⋆ can be used in any context, e.g., addition, function application, or even communication, and any value can be passed where ⋆ is expected. 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. It is important to see that the channel c is handled linearly in functions dynServer and serveOp. 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 shown in earlier work (Fennell & Thiemann, 2012; Thiemann, 2014). Aside from the type annotation, the code remains exactly the same as in the unityped case.

The external language comes with a translation into a blame calculus with explicit casts. This translation inserts just the casts that are necessary to make typing of the code go through. Here is the output of this translation (suffix Cast is appended to the names of the functions to distinguish different versions):

```
dynServerCast : Compute → unit
dynServerCast c =
  case c of {
    neg: c. serveOpCast 1 ((λx.−x) : int → int ⇒^{ℓ1} ⋆)
                          (c : ?int.!int.end! ⇒^{ℓ2} ⊛);
    add: c. serveOpCast 2 ((λx.λy.x+y) : int → int → int ⇒^{ℓ3} ⋆)
                          (c : ?int.?int.!int.end! ⇒^{ℓ4} ⊛)
  }

serveOpCast : int → ⋆ → ⊛ → unit
```

https://doi.org/10.1017/S0956796819000169 Published online by Cambridge University Press

```
serveOpCast n op c =
  if n==0 then
    close ((send op (c : ⊛ ⇒^{ℓ5} !⋆.⊛)) : ⊛ ⇒^{ℓ6} end! )
  else
    let v,c = receive (c : ⊛ ⇒^{ℓ7} ?⋆.⊛) in
    serveOpCast (n−1) ((op : ⋆ ⇒^{ℓ8} ⋆ → ⋆) v) c
```

Casts of the form $e : T_1 \overset{p}{\Rightarrow} T_2$—meaning that e of type T_1 is cast to T_2—are inserted where values are converted from/to $\star$ or $\circledast$, similarly to the translation from GTLC. The *blame labels* $\ell_1, \ell_2, \ldots$ (ranged over by p and q) on the arrow identify casts, when they fail. The resulting casts in dynServerCast and serveOpCast look fairly involved, but we should keep in mind that the programmer does *not* have to write them as they result from the translation. In practice, blame labels may contain information on program locations to help identify how a program fails. For example, if Rob made the following mistake in writing his dynServer

```
neg: c. serveOp 2 (λx.−x) c;
   -- The first argument to serveOp should be 1!
```

then a call to negationClient would fail after the server receives the first integer from the client. More specifically, the failure would identify the cast labeled ℓ_7 failed because a channel endpoint whose session type is !int.end! had been flown from ℓ_2.

2.5 Dynamic linearity

The refined criteria for gradual typing (Siek *et al.*, 2015b) postulate that a gradual type system should come with a full embedding of a unityped calculus. This embedding (which we indicate by ceiling brackets $\lceil \ldots \rceil$) extends the embedding given for the simply typed lambda calculus (Wadler & Findler, 2009) to handle the operations on sessions (see Figure 13 for its definition).

For example, (the unityped version of) the dynServer as written by Rob is compiled and embedded into the gradually typed language as a value dynServer : $\star$. To directly incorporate Rob's code, the gradual type checker enables Sy to write a function callDynServer that accepts a channel of type Compute and returns a value of type **unit**, but internally just calls dynServer.

```
callDynServer : Compute → unit
callDynServer c =
  dynServer c
```

The gradual type checker translates the definition of callDynServer by inserting the appropriate casts: it casts the embedded dynServer (of type $\star$) to the function type $\star \to \star$, it casts the channel argument to this function to $\star$, and it casts the result to **unit**.

```
callDynServer : Compute → unit
callDynServer c =
  ((dynServer : ⋆ ⇒^{ℓ9} ⋆ → ⋆) (c : Compute ⇒^{ℓ10} ⋆)) : ⋆ ⇒^{ℓ11} unit
```

The casts inserted in this code make Sy's expectations completely obvious: dynServer must be a function and it is expected to use c as a channel of type Compute. Any misuse will allocate blame to the respective cast in dynServer.

https://doi.org/10.1017/S0956796819000169 Published online by Cambridge University Press

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. Both actions can lead to protocol violations, which should be detected at runtime. Gradual GV takes care of linearity by factoring the cast $(\mathsf{c} : \mathsf{Compute} \overset{\ell_9}{\Rightarrow} \star)$ through the dynamic session type $\circledast$:

$$((\mathsf{c} : \mathsf{Compute} \overset{\ell_9}{\Rightarrow} \circledast) : \circledast \overset{\ell_9}{\Rightarrow} \star)$$

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 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 $E[v : \circledast \overset{p}{\Rightarrow} \star]$ reduces to $(\nu a)(E[a] \mid a \mapsto v : \circledast \overset{p}{\Rightarrow} \star)$. Here, $(\nu a)P$ represents the scope of a fresh reference to a linear value and the process $a \mapsto v : \circledast \overset{p}{\Rightarrow} \star$ represents the cell storing v at a. Linear use of this cell is controlled at runtime using ideas for runtime monitoring of affine types (Tov & Pucella, 2010; Padovani, 2017).

Any access to a cell comes in the guise of a cast $a : \star \overset{p}{\Rightarrow} T$ 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 type, then blame is allocated to that cast. 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 examples so far tacitly assume that channels are created with a fully specified session type that provides a "ground truth" for the protocol on this channel. 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 (cf. the Compute and ComputeD types in Sections 2.1 and 2.2). 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 $\circledast$ and casting it to $\star$ as in $\mathbf{new} : \circledast \times_{\mathsf{lin}} \circledast \Rightarrow \star$. To make this work, the dynamic session type $\circledast$ is considered *self-dual*, that is $\overline{\circledast} = \circledast$. Gradual GV offers *no static guarantees* for either end of such a channel.

https://doi.org/10.1017/S0956796819000169 Published online by Cambridge University Press

To see what runtime guarantees Gradual GV can offer for a channel of unknown session type, 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.

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

(Here, p and q are metavariables ranging over blame labels.) Now consider running the following unityped program with entry point main.

```
1  client cc =
2    let v,cc = receive cc in wait cc
3  server cs =
4    let cs = send 42 cs in close cs
5  main =
6    let cs,cc = new in
7    let _ = fork (client cc) in
8    server cs
```

After a few computation steps, it reaches a configuration where the client and the server have reduced to $(\nu cc, cs)(\,\mathsf{client} \mid \mathsf{server}\,)$ where

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

for some contexts E and F. The channel ends $cc : \circledast$ and $cs : \circledast$ are the two ends of the channel created in line 6. Fortunately, the two processes use the channel consistently as the cast target $?\star.\circledast$ on one end is dual to the cast target $!\star.\circledast$ at the other end. Hence, Gradual GV has a reduction that drops the casts at both ends in this situation, and retypes the ends to $cc : ?\star.\circledast$ and $cs : !\star.\circledast$, respectively.

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

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 single *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 check locally whether the target types are duals of one another. If they are, then both processes continue; otherwise they allocate blame. As both ends perform the same comparison, the outcome 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. The language is inspired by both the Gay and Vasconcelos' (2010) functional session type calculus and Wadler's (2012, 2014) 'good variant' of the language. A main difference from the former is the introduction of communication primitives and session types to close

https://doi.org/10.1017/S0956796819000169 Published online by Cambridge University Press

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

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

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

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 $\boxed{T <: U}$

$$\mathsf{unit} <: \mathsf{unit} \qquad \frac{T' <: T \quad U <: U' \quad m <: n}{T \to_m U <: T' \to_n U'} \qquad \frac{T <: T' \quad U <: U' \quad m <: n}{T \times_m U <: T' \times_n U'}$$
$$\frac{T' <: T \quad S <: S'}{!T.S <: !T'.S'} \qquad \frac{T <: T' \quad S <: S'}{?T.S <: ?T'.S'} \qquad \frac{J \subseteq I \quad (S_j <: R_j)_{j\in J}}{\oplus\{l_i\colon S_i\}_{i\in I} <: \oplus\{l_j\colon R_j\}_{j\in J}}$$
$$\frac{I \subseteq J \quad (S_i <: R_i)_{i\in I}}{\&\{l_i\colon S_i\}_{i\in I} <: \&\{l_j\colon R_j\}_{i\in J}} \qquad \mathsf{end}_! <: \mathsf{end}_! \qquad \mathsf{end}_? <: \mathsf{end}_?$$

Fig. 1. Types and subtyping in GV.

a session explicitly. Unlike the latter, types are "stratified" into two levels—sessions types and plain types—and deadlock freedom is not guaranteed.

3.1.1 Types and subtyping

Figure 1 summarizes 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 $\oplus\{l_i\colon S_i\}_{i\in I}$, to send one of the labels l_i and then behave as S_i; case $\&\{l_i\colon S_i\}_{i\in I}$ to receive any of the labels l_i and then behave as S_i; close $\mathsf{end}_!$, to close a channel endpoint; and wait $\mathsf{end}_?$, to wait for the other end of the channel to close. In $\oplus\{l_i\colon S_i\}_{i\in I}$ and $\&\{l_i\colon S_i\}_{i\in I}$, the label set must be non-empty. We will call the session type that describes the behavior after send, receive, select, or case the *residual*.

https://doi.org/10.1017/S0956796819000169 Published online by Cambridge University Press

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. To ensure that linear objects are used exactly once our type system imposes the invariant that unrestricted data structures do not contain linear data structures. As an example, type $\mathsf{unit} \times_{\mathsf{un}} \mathsf{end}_!$ cannot be introduced in any derivation. 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 & Vasconcelos, 2010). Function types are contravariant in their domain, covariant in their range, and covariant in their multiplicity, 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 receive any label with an index in J, so long as $I \subseteq J$. (Subtyping on endpoints appears sometimes reversed in process-calculus like systems, such as (Carbone *et al.*, 2007, 2012; Demangeon & Honda, 2011), Wadler's CP (2012, 2014); Gay (2016) discusses the situation.)

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 summarized 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. We sometimes abbreviate expressions of the form $(\lambda_{\mathsf{lin}} x.e)f$ to $\mathsf{let}\, x = e\, \mathsf{in}\, f$, as usual. A GV program is always given as an expression, but as it executes it may fork new processes.

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 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 $(\nu c, d)P$. We assume that c and d in $(\nu c, d)P$ are different. The notions of free and bound names/variables as well that substitution are defined accordingly. The set of the free

https://doi.org/10.1017/S0956796819000169 Published online by Cambridge University Press

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

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

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

$$\frac{\Gamma \vdash e : T \quad \Delta \vdash f : U \quad m^{:>}(T) \quad m^{:>}(U)}{\Gamma \circ \Delta \vdash (e,f)_m : T \times_m U} \qquad \frac{\Gamma \vdash e : T_1 \times_m T_2 \quad \Delta, x : T_1, y : T_2 \vdash f : U}{\Gamma \circ \Delta \vdash \mathsf{let}\,x,y = e\,\mathsf{in}\,f : U}$$

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

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

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

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.

names in P is denoted by $\mathsf{fn}(P)$. 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. Thus, it is never possible to construct a pair of type, e.g., $S \times_{\mathsf{un}} T$, which contains

https://doi.org/10.1017/S0956796819000169 Published online by Cambridge University Press

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

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

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

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

$$
P \mid Q \equiv Q \mid P \qquad P \mid (Q \mid P') \equiv (P \mid Q) \mid P' \qquad P \mid \langle () \rangle \equiv P \qquad (\nu c,d)P \equiv (\nu d,c)P
$$

$$
\begin{array}{rcl}
((\nu c,d)P) \mid Q & \equiv & (\nu c,d)(P \mid Q) \qquad \text{if } \{c,d\} \cap \mathsf{fn}(Q) = \emptyset \\
(\nu c,d)(\nu c',d')P & \equiv & (\nu c',d')(\nu c,d)P \quad \text{if } \{c,d\} \cap \{c',d'\} = \emptyset
\end{array}
$$

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

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

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

Fig. 3. Reduction in GV.

a linear type S under the unrestricted pair type constructor $\times_{\mathsf{un}}$, even though such a type is syntactically allowed for simplicity. The rules for send, receive, select, case, close, and wait match the corresponding session types. For example, the following type judgment

$$
o\colon \mathsf{int}, c\colon !\mathsf{int}.\mathsf{end}_! \vdash \mathsf{close}\,(\mathsf{send}\,o\,c) : \mathsf{unit}
$$

can be derived. 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 Reduction

Values, evaluation contexts, reduction for expressions, structural congruence, and reduction for processes for GV are summarized 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.

https://doi.org/10.1017/S0956796819000169 Published online by Cambridge University Press

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 congruence 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. Similarly for the rule to swap prefixes.

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 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. Following standard frameworks of gradual typing (Siek & Taha, 2006; Siek *et al.*, 2015b), Gradual GV consists of two sublanguages: an external language GGV_e, in which source programs are written, and an internal language GGV_i, to which GGV_e is elaborated by cast-inserting translation to make necessary runtime checks explicit. The operational semantics of a program is given as reduction of processes in GGV_i. We first introduce GGV_i by outlining its differences to GV (Sections 3.2.1–3.2.3). Next, we introduce the syntax of GGV_e, which has only expressions, because it is the language in which source programs are written, its type system, and cast-inserting translation from GGV_e to GGV_i (Sections 3.2.4 and 3.2.5). Finally, we discuss how an untyped variant of GV can be embedded into GGV_i (Section 3.2.6).

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 typeset 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 $\circledast$ to be self-dual: $\overline{\circledast} = \circledast$. 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 in the following) it corresponds to all possible types, both unrestricted and linear, and therefore

https://doi.org/10.1017/S0956796819000169 Published online by Cambridge University Press

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

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

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

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

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

Subtyping $\boxed{T <: U}$

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

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

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

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

$$\frac{I \subseteq J \quad (S_i \lesssim S'_i)_{i \in I}}{\&\{l_i : S_i\}_{i \in I} \lesssim \&\{l_j : S'_j\}_{j \in J}} \qquad \mathsf{end}_! \lesssim \mathsf{end}_! \qquad \mathsf{end}_? \lesssim \mathsf{end}_? \qquad \circledast \lesssim S \qquad S \lesssim \circledast$$

Fig. 4. Types and subtyping in Gradual GV.

we will need to take special care when handling values of type $\star$ that correspond to values of a linear type.

Consistent subtyping is defined over types of Gradual GV also 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 runtime: while a cast using (a) will not fail, a cast using (b) may fail because an expression of type $\&\{l_1 : \circledast\}$ may evaluate to a value of type, say, $\&\{l_1 : \mathsf{end}_!\}$.

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.

https://doi.org/10.1017/S0956796819000169 Published online by Cambridge University Press

Blame labels	p, q	
References	a, b	
Names	z	$::= \cdots \mid a$
Expressions	e, f	$::= \cdots \mid e : T \overset{p}{\Rightarrow} U$
Processes	P, Q	$::= \cdots \mid (\nu a)P \mid a \mapsto w : \mathbf{T} \overset{p}{\Rightarrow} \star \mid a \mapsto \mathsf{locked}\, p \mid \mathsf{blame}\, p\, q\, X \mid \mathsf{blame}\, p\, X$

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

$$\frac{\Gamma \vdash e : T \quad T \lesssim U}{\Gamma \vdash (e : T \overset{p}{\Rightarrow} U) : U}$$

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

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

$$\frac{\mathsf{flv}(\Gamma) = X}{\Gamma \vdash \mathsf{blame}\, p\, q\, X} \qquad \frac{\mathsf{flv}(\Gamma) = X}{\Gamma \vdash \mathsf{blame}\, p\, X}$$

Fig. 5. Expressions, processes, and typing in GGV$_i$.

3.2.2 *Expressions, processes, and typing of GGV$_i$*

Expressions, processes, and type rules of GGV$_i$ are summarized in Figure 5. The expressions of GGV$_i$ are those of GV, plus an additional form for casts. A cast is written

$$e : T \overset{p}{\Rightarrow} U \tag{1}$$

where e is an expression of type T, and p, q range over blame labels such as $\ell_1, \ell_2, \ldots$. For example, the following term

$$SOC = \lambda_{\mathsf{un}} o.\lambda_{\mathsf{un}} c.\mathsf{close}\,((\mathsf{send}\, o\, (c : \circledast \overset{\ell_1}{\Rightarrow} !\star.\circledast)) : \circledast \overset{\ell_2}{\Rightarrow} \mathsf{end}_!)$$

which represents a simplified version of serveOpCast in Section 2, can be given type $\star \rightarrow_{\mathsf{un}} \circledast \rightarrow_{\mathsf{un}} \mathsf{unit}$.

Blame labels carry a polarity, which is either positive or negative. The complement operation, $\overline{p}$, takes a positive label into a negative one and vice versa; complement is an involution, so that $\overline{\overline{p}} = p$. By convention, we assume that all blame labels in a source program are positive, but negative blame labels may arise during evaluation of casts at a function type or a send type. A cast raises *positive blame* if the fault lies with the expression *contained* in the cast (for instance, because it returns an integer where a character is expected), while it raises *negative blame* if the fault lies with the context *containing* the cast (for instance, because it passes an argument or sends a value that is an integer where a character is expected).

In a valid cast $e : T \overset{p}{\Rightarrow} U$, the type T must be a consistent subtype of U ($T \lesssim U$), the type of the entire expression. If a cast in a program fails, it evaluates to $\mathsf{blame}\, p\, q\, X$ or $\mathsf{blame}\, p\, X$ (which, as we see later, are treated as processes) where the blame label p and q indicate the root cause of the failure (we will explain X shortly). If the cast in (1) fails, it means that the value returned by e has type T, but not type U. For example, let $e = 4711 : \mathsf{int} \overset{q}{\Rightarrow} \star$, $T = \star$, and $U = \mathsf{bool}$. As $\star \lesssim \mathsf{bool}$, the resulting expression

https://doi.org/10.1017/S0956796819000169 Published online by Cambridge University Press

$(4711 : \mathsf{int} \overset{q}{\Rightarrow} \star) : \star \overset{p}{\Rightarrow} \mathsf{bool}$ is well typed. However, at runtime it raises blame by reducing to $\mathsf{blame}\, \overline{q}\, p\, \emptyset$, which flags the error that int is not a subtype of bool: that is, $\mathsf{int} \not< : \mathsf{bool}$.

Blame is indicated by processes 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 the 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. In a running program, only free linear variables are channel endpoints, so $\mathsf{flv}(E)$ and $\mathsf{flv}(v)$ can be defined without type information.

The processes of GGV_i are those of GV, plus three additional forms for references to linear values (as well as blame, described above). Recall that a value of type $\star$ may contain a linear value, in which case dynamic checking must ensure that it is used exactly once. The mechanism for doing so is to allocate a reference to a linear value. We let a, b range over references. A reference is of type $\star$, and contains a value w of ground type $\mathbf{T}$, where $\mathbf{T}$ is linear (either $\star \rightarrow_{\mathsf{lin}} \star$ or $\star \times_{\mathsf{lin}} \star$ or the dynamic session type $\circledast$). 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. Bindings for references initially take the first form, but change to the second form after the reference has been accessed once; any subsequent attempt to access the reference a second time will cause an error.

3.2.3 Reduction

Values, evaluation contexts, reductions for expressions, structural congruence, and reductions for processes for GGV_i are summarized in Figures 6 and 7.

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

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

where $S, R \neq \circledast$.

https://doi.org/10.1017/S0956796819000169 Published online by Cambridge University Press

$$
\begin{array}{llll}
\text{Values} & v,w & ::= & \cdots \mid v\colon \mathbf{T} \overset{p}{\Rightarrow} \star \mid v\colon \mathbf{S} \overset{p}{\Rightarrow} \circledast \mid v\colon T \to_m U \overset{p}{\Rightarrow} T' \to_n U' \mid v\colon S \overset{p}{\Rightarrow} R \mid a \\
 & & & \quad \text{where } \mathsf{un}(\mathbf{T}), S \neq \circledast, R \neq \circledast \\
\text{Eval contexts} & E,F & ::= & \cdots \mid E\colon T \overset{p}{\Rightarrow} U
\end{array}
$$

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

$$
\begin{array}{rcll}
v\colon \star \overset{p}{\Rightarrow} \star & \longrightarrow & v & \\
v\colon \circledast \overset{p}{\Rightarrow} \circledast & \longrightarrow & v & \\
v\colon \mathsf{unit} \overset{p}{\Rightarrow} \mathsf{unit} & \longrightarrow & v & \\
(v\colon T \to_m U \overset{p}{\Rightarrow} T' \to_n U')\, w & \longrightarrow & (v\,(w\colon T' \overset{\overline{p}}{\Rightarrow} T))\colon U \overset{p}{\Rightarrow} U' & \\
(v,w)_m\colon T \times_m U \overset{p}{\Rightarrow} T' \times_n U' & \longrightarrow & (v\colon T \overset{p}{\Rightarrow} T', w\colon U \overset{p}{\Rightarrow} U')_n & \\
\mathsf{send}\, v\,(w\colon !T.S \overset{p}{\Rightarrow} !T'.S') & \longrightarrow & (\mathsf{send}\,(v\colon T' \overset{\overline{p}}{\Rightarrow} T)\, w)\colon S \overset{p}{\Rightarrow} S' & \\
\mathsf{receive}\,(w\colon ?T.S \overset{p}{\Rightarrow} ?T'.S') & \longrightarrow & (\mathsf{receive}\, w)\colon T \times_{\mathsf{lin}} S \overset{p}{\Rightarrow} T' \times_{\mathsf{lin}} S' & \\
\mathsf{select}\, l_k\,(w\colon \oplus\{l_i\colon R_i\}_{i\in I} \overset{p}{\Rightarrow} \oplus\{l_j\colon S_j\}_{j\in J}) & & & \\
 & \longrightarrow & (\mathsf{select}\, l_k\, w)\colon R_k \overset{p}{\Rightarrow} S_k & \text{if } k \in J, J \subseteq I \\
\mathsf{case}\,(w\colon \&\{l_i\colon R_i\}_{i\in I} \overset{p}{\Rightarrow} \&\{l_i\colon S_i\}_{i\in J})\,\mathsf{of}\,\{l_j\colon x_j.e_j\}_{j\in J} & & & \\
 & \longrightarrow & \mathsf{case}\, w\, \mathsf{of}\, \{l_i\colon x_i.\mathsf{let}\, x_i = (x_i\colon R_i \overset{p}{\Rightarrow} S_i)\,\mathsf{in}\, e_i\}_{i\in I} & \text{if } I \subseteq J \\
\mathsf{close}\,(v\colon \mathsf{end}_! \overset{p}{\Rightarrow} \mathsf{end}_!) & \longrightarrow & \mathsf{close}\, v & \\
\mathsf{wait}\,(v\colon \mathsf{end}_? \overset{p}{\Rightarrow} \mathsf{end}_?) & \longrightarrow & \mathsf{wait}\, v & \\
v\colon T \overset{p}{\Rightarrow} \star & \longrightarrow & (v\colon T \overset{p}{\Rightarrow} \mathbf{T})\colon \mathbf{T} \overset{p}{\Rightarrow} \star & \text{if } T \neq \star, T \neq \mathbf{T}, T \sim \mathbf{T} \\
v\colon \star \overset{p}{\Rightarrow} T & \longrightarrow & (v\colon \star \overset{p}{\Rightarrow} \mathbf{T})\colon \mathbf{T} \overset{p}{\Rightarrow} T & \text{if } T \neq \star, T \neq \mathbf{T}, T \sim \mathbf{T} \\
v\colon S \overset{p}{\Rightarrow} \circledast & \longrightarrow & (v\colon S \overset{p}{\Rightarrow} \mathbf{S})\colon \mathbf{S} \overset{p}{\Rightarrow} \circledast & \text{if } S \neq \circledast, S \neq \mathbf{S}, S \sim \mathbf{S} \\
v\colon \circledast \overset{p}{\Rightarrow} S & \longrightarrow & (v\colon \circledast \overset{p}{\Rightarrow} \mathbf{S})\colon \mathbf{S} \overset{p}{\Rightarrow} S & \text{if } S \neq \circledast, S \neq \mathbf{S}, S \sim \mathbf{S}
\end{array}
$$

Fig. 6. Reduction in GGV$_i$, expressions.

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

$$
\begin{array}{rl}
((\nu a)P) \mid Q \equiv (\nu a)(P \mid Q) & \text{if } \{c,d\} \cap \mathsf{fn}(Q) = \emptyset \\
(\nu a)(\nu b)P \equiv (\nu b)(\nu a)P & \text{if } a \neq b \\
(\nu c,d)(\nu a)P \equiv (\nu a)(\nu c,d)P & \text{if } a \neq c \text{ and } a \neq d
\end{array}
$$

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

$$
\begin{array}{rcll}
\langle E[v\colon \mathbf{T} \overset{p}{\Rightarrow} \star]\rangle & \longrightarrow & (\nu a)(\langle E[a]\rangle \mid a \mapsto v : \mathbf{T} \overset{p}{\Rightarrow} \star) & \\
 & & & \text{if } \mathsf{lin}(\mathbf{T}) \text{ and } E \neq F[[\,]\colon \star \overset{q}{\Rightarrow} \mathbf{U}] \\
\langle E[a\colon \star \overset{q}{\Rightarrow} \mathbf{U}]\rangle \mid a \mapsto v : \mathbf{T} \overset{p}{\Rightarrow} \star & \longrightarrow & \langle E[(v : \mathbf{T} \overset{p}{\Rightarrow} \star)\colon \star \overset{q}{\Rightarrow} \mathbf{U}]\rangle \mid a \mapsto \mathsf{locked}\, p & \\
\langle E[a\colon \star \overset{q}{\Rightarrow} \mathbf{U}]\rangle \mid a \mapsto \mathsf{locked}\, p & \longrightarrow & \mathsf{blame}\, \overline{p}\, q\,(\mathsf{flv}(E)) \mid a \mapsto \mathsf{locked}\, p & \\
(\nu a)(a \mapsto \mathsf{locked}\, p) & \longrightarrow & \langle ()\rangle & \\
(\nu a)(a \mapsto w : \mathbf{T} \overset{p}{\Rightarrow} \star) & \longrightarrow & \mathsf{blame}\, \overline{p}\,(\mathsf{flv}(w)) & \\
\langle E[(v\colon \mathbf{T} \overset{p}{\Rightarrow} \star)\colon \star \overset{q}{\Rightarrow} \mathbf{U}]\rangle & \longrightarrow & \langle E[v]\rangle & \text{if } \mathbf{T} <: \mathbf{U} \\
\langle E[(v\colon \mathbf{T} \overset{p}{\Rightarrow} \star)\colon \star \overset{q}{\Rightarrow} \mathbf{U}]\rangle & \longrightarrow & \mathsf{blame}\, \overline{p}\, q\,(\mathsf{flv}(E) \cup \mathsf{flv}(v)) & \text{if } \mathbf{T} \not<: \mathbf{U} \\
\langle E[(v\colon \mathbf{S} \overset{p}{\Rightarrow} \circledast)\colon \circledast \overset{q}{\Rightarrow} \mathbf{R}]\rangle & \longrightarrow & \langle E[v]\rangle & \text{if } \mathbf{S} <: \mathbf{R} \\
\langle E[(v\colon \mathbf{S} \overset{p}{\Rightarrow} \circledast)\colon \circledast \overset{q}{\Rightarrow} \mathbf{R}]\rangle & \longrightarrow & \mathsf{blame}\, \overline{p}\, q\,(\mathsf{flv}(E) \cup \mathsf{flv}(v)) & \text{if } \mathbf{S} \not<: \mathbf{R} \\
(\nu c,d)(\langle E[c\colon \circledast \overset{p}{\Rightarrow} \mathbf{S}]\rangle \mid \langle F[d\colon \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} \\
(\nu c,d)(\langle E[c\colon \circledast \overset{p}{\Rightarrow} \mathbf{S}]\rangle \mid \langle F[d\colon \circledast \overset{q}{\Rightarrow} \mathbf{R}]\rangle) & \longrightarrow & \mathsf{blame}\, p\, q\,(\mathsf{flv}(E) \cup \mathsf{flv}(F) \cup \{c,d\}) & \text{if } \overline{\mathbf{S}} \not<: \mathbf{R}
\end{array}
$$

$$
\frac{P \longrightarrow Q}{(\nu a)P \longrightarrow (\nu a)Q}
$$

Fig. 7. Reduction in GGV$_i$, processes.

https://doi.org/10.1017/S0956796819000169 Published online by Cambridge University Press

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:

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

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

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

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

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

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 nontrivial 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 processes appear in Figure 7. Like bindings for channel endpoints, bindings for references to linear values satisfy scope extrusion and reduction is a congruence with respect to them.

The first five reduction rules for processes deal with references to linear values, ensuring that a value cast from a linear type to $\star$ is accessed exactly once. As the only values of the dynamic type are casts from a ground type, expressions of interest take the form

$$v\colon \mathbf{T} \overset{p}{\Rightarrow} \star$$

where v is a value and $\mathbf{T}$ is a linear ground type. The first rule introduces a reference, represented as a separate process of the form $a \mapsto v\colon \mathbf{T} \overset{p}{\Rightarrow} \star$. The context restriction $E \neq F[[\,]\colon \star \overset{q}{\Rightarrow} \mathbf{U}]$ ensures that a reference is only introduced if the value is not immediately accessed; without the restriction this rule would apply to a process of the form $\langle E[(v\colon \mathbf{T} \overset{p}{\Rightarrow} \star)\colon \star \overset{q}{\Rightarrow} \mathbf{U}]\rangle$, to which the sixth or seventh rule should be applied. Any attempt to access the linear reference a must take the form

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

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

https://doi.org/10.1017/S0956796819000169 Published online by Cambridge University Press

of the 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 the two casts involved, negative blame $\overline{p}$ from $\mathsf{locked}\, p$, which was a cast $v\colon \mathbf{T} \overset{p}{\Rightarrow} \star$ before the first access, and positive blame q for the cast to access a, 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 reference finish execution. 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. In practice, these rules would be implemented as part of garbage collection.

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

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

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 $v\colon \mathbf{T} \overset{p}{\Rightarrow} \star$ is just taken to be the erasure of v itself. However, this asymmetric 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.

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

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 $\overline{\mathbf{R}}, \mathbf{R}$. Analogously, 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:

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

https://doi.org/10.1017/S0956796819000169 Published online by Cambridge University Press

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 easier to implement.

Now we show a few examples of reduction, in which we abbreviate a nested cast $(e\colon T_1 \overset{p}{\Rightarrow} T_2)\colon T_2 \overset{q}{\Rightarrow} T_3$ to $e\colon T_1 \overset{p}{\Rightarrow} T_2 \overset{q}{\Rightarrow} T_3$ and use a sequential composition $e_1; e_2$ with obvious typing and reduction rules. First recall the term

$$SOC = \lambda_{\mathsf{un}} o.\lambda_{\mathsf{un}} c.\mathsf{close}\,(\mathsf{send}\, o\,(c\colon \circledast \overset{\ell_1}{\Rightarrow} !\star.\circledast)\colon \circledast \overset{\ell_2}{\Rightarrow} \mathsf{end}_!)$$

introduced above. Given a channel endpoint $d\colon !\mathsf{int}.\mathsf{end}_!$, the term

$$SOC\,(42\colon \mathsf{int} \overset{\ell_3}{\Rightarrow} \star)\,(d\colon !\mathsf{int}.\mathsf{end}_! \overset{\ell_4}{\Rightarrow} \circledast)$$

reduces as follows:

$$\begin{aligned}
&SOC\,(42\colon \mathsf{int} \overset{\ell_3}{\Rightarrow} \star)\,(d\colon !\mathsf{int}.\mathsf{end}_! \overset{\ell_4}{\Rightarrow} \circledast)\\
\longrightarrow\ &(\lambda_{\mathsf{un}} c.\mathsf{close}\,((\mathsf{send}\,(42\colon \mathsf{int} \overset{\ell_3}{\Rightarrow} \star)\,(c\colon \circledast \overset{\ell_1}{\Rightarrow} !\star.\circledast))\colon \circledast \overset{\ell_2}{\Rightarrow} \mathsf{end}_!))\,(d\colon !\mathsf{int}.\mathsf{end}_! \overset{\ell_4}{\Rightarrow} \circledast)\\
\longrightarrow\ &(\lambda_{\mathsf{un}} c.\mathsf{close}\,((\mathsf{send}\,(42\colon \mathsf{int} \overset{\ell_3}{\Rightarrow} \star)\,(c\colon \circledast \overset{\ell_1}{\Rightarrow} !\star.\circledast))\colon \circledast \overset{\ell_2}{\Rightarrow} \mathsf{end}_!))\\
&\qquad\qquad (d\colon !\mathsf{int}.\mathsf{end}_! \overset{\ell_4}{\Rightarrow} !\star.\circledast \overset{\ell_4}{\Rightarrow} \circledast)\\
\longrightarrow\ &\mathsf{close}\,((\mathsf{send}\,(42\colon \mathsf{int} \overset{\ell_3}{\Rightarrow} \star)\,(d\colon !\mathsf{int}.\mathsf{end}_! \overset{\ell_4}{\Rightarrow} !\star.\circledast \overset{\ell_4}{\Rightarrow} \circledast \overset{\ell_1}{\Rightarrow} !\star.\circledast))\colon \circledast \overset{\ell_2}{\Rightarrow} \mathsf{end}_!)\\
\longrightarrow\ &\mathsf{close}\,((\mathsf{send}\,(42\colon \mathsf{int} \overset{\ell_3}{\Rightarrow} \star)\,(d\colon !\mathsf{int}.\mathsf{end}_! \overset{\ell_4}{\Rightarrow} !\star.\circledast))\colon \circledast \overset{\ell_2}{\Rightarrow} \mathsf{end}_!)\\
\longrightarrow\ &\mathsf{close}\,((\mathsf{send}\,(42\colon \mathsf{int} \overset{\ell_3}{\Rightarrow} \star \overset{\overline{\ell}_4}{\Rightarrow} \mathsf{int})\,d)\colon \mathsf{end}_! \overset{\ell_4}{\Rightarrow} \circledast \overset{\ell_2}{\Rightarrow} \mathsf{end}_!)\\
\longrightarrow\ &\mathsf{close}\,((\mathsf{send}\,42\,d)\colon \mathsf{end}_! \overset{\ell_4}{\Rightarrow} \circledast \overset{\ell_2}{\Rightarrow} \mathsf{end}_!).
\end{aligned}$$

Thus, the process

$$(\nu d, e)(\langle SOC\,(42\colon \mathsf{int} \overset{\ell_3}{\Rightarrow} \star)\,(d\colon !\mathsf{int}.\mathsf{end}_! \overset{\ell_4}{\Rightarrow} \circledast)\rangle \mid \langle \mathsf{let}\, x, y = \mathsf{receive}\, e\, \mathsf{in}\, \mathsf{wait}\, y\rangle)$$

reduces as follows:

$$\begin{aligned}
&(\nu d, e)(\langle SOC\,(42\colon \mathsf{int} \overset{\ell_3}{\Rightarrow} \star)\,(d\colon !\mathsf{int}.\mathsf{end}_! \overset{\ell_4}{\Rightarrow} \circledast)\rangle \mid \langle \mathsf{let}\, x, y = \mathsf{receive}\, e\, \mathsf{in}\, \mathsf{wait}\, y\rangle)\\
\longrightarrow^{+}\ &(\nu d, e)(\langle \mathsf{close}\,((\mathsf{send}\,42\,d)\colon \mathsf{end}_! \overset{\ell_4}{\Rightarrow} \circledast \overset{\ell_2}{\Rightarrow} \mathsf{end}_!)\rangle \mid \langle \mathsf{let}\, x, y = \mathsf{receive}\, e\, \mathsf{in}\, \mathsf{wait}\, y\rangle)\\
\longrightarrow\ &(\nu d, e)(\langle \mathsf{close}\,(d\colon \mathsf{end}_! \overset{\ell_4}{\Rightarrow} \circledast \overset{\ell_2}{\Rightarrow} \mathsf{end}_!)\rangle \mid \langle \mathsf{let}\, x, y = (42, e)_{\mathsf{lin}}\, \mathsf{in}\, \mathsf{wait}\, y\rangle)\\
\longrightarrow^{+}\ &(\nu d, e)(\langle \mathsf{close}\, d\rangle \mid \langle \mathsf{wait}\, e\rangle)\\
\longrightarrow\ &(\nu d, e)(\langle ()\rangle \mid \langle ()\rangle)
\end{aligned}$$

However, if d is given type $!\mathsf{int}.!\mathsf{int}.\mathsf{end}_!$, then $SOC\,(42\colon \mathsf{int} \overset{\ell_3}{\Rightarrow} \star)\,(d\colon !\mathsf{int}.!\mathsf{int}.\mathsf{end}_! \overset{\ell_4}{\Rightarrow} \circledast)$ is well typed but reduces to

$$\mathsf{close}\,((\mathsf{send}\,42\,d)\colon !\mathsf{int}.\mathsf{end}_! \overset{\ell_4}{\Rightarrow} \circledast \overset{\ell_2}{\Rightarrow} \mathsf{end}_!)$$

Thus, the process

$$(\nu d, e)(\langle SOC\,(42\colon \mathsf{int} \overset{\ell_3}{\Rightarrow} \star)\,(d\colon !\mathsf{int}.!\mathsf{int}.\mathsf{end}_! \overset{\ell_4}{\Rightarrow} \circledast)\rangle \mid \langle \mathsf{let}\, x, y = \mathsf{receive}\, e\, \mathsf{in} \ldots\rangle)$$

reduces to

$$(\nu d, e)(\langle \mathsf{close}\,(d\colon !\mathsf{int}.\mathsf{end}_! \overset{\ell_4}{\Rightarrow} \circledast \overset{\ell_2}{\Rightarrow} \mathsf{end}_!)\rangle \mid \langle \mathsf{let}\, x, y = (42, e)_{\mathsf{lin}}\, \mathsf{in} \ldots\rangle)$$

https://doi.org/10.1017/S0956796819000169 Published online by Cambridge University Press

and then to

$$(\nu d, e)(\mathsf{blame}\ \overline{\ell}_4\ \ell_2\ \{d\} \mid \langle \mathsf{let}\, x, y = (42, e)_{\mathsf{lin}}\ \mathsf{in} \ldots \rangle)$$

We also show an example of dynamic linearity checking. The following function *foo* takes an argument of type $\star$, cast it to $\mathsf{end}_!$, and closes it:

$$foo = \lambda_{\mathsf{un}} x.\mathsf{close}\,(x\colon \star \overset{\ell}{\Rightarrow} \mathsf{end}_!)$$

Consider an application of *foo* to a channel endpoint c of type $\mathsf{end}_!$. It reduces as follows:

$$\begin{aligned}
&\langle foo\,(c\colon \mathsf{end}_! \overset{\ell'}{\Rightarrow} \star)\rangle \\
\longrightarrow\ &\langle foo\,(c\colon \mathsf{end}_! \overset{\ell'}{\Rightarrow} \circledast \overset{\ell'}{\Rightarrow} \star)\rangle \\
\longrightarrow\ &(\nu a)(\langle foo\, a\rangle \mid a \mapsto c\colon \mathsf{end}_! \overset{\ell'}{\Rightarrow} \circledast \overset{\ell'}{\Rightarrow} \star) \\
\longrightarrow\ &(\nu a)(\langle \mathsf{close}\,(a\colon \star \overset{\ell}{\Rightarrow} \mathsf{end}_!)\rangle \mid a \mapsto c\colon \mathsf{end}_! \overset{\ell'}{\Rightarrow} \circledast \overset{\ell'}{\Rightarrow} \star) \\
\longrightarrow\ &(\nu a)(\langle \mathsf{close}\,(c\colon \mathsf{end}_! \overset{\ell'}{\Rightarrow} \circledast \overset{\ell'}{\Rightarrow} \star \overset{\ell}{\Rightarrow} \mathsf{end}_!)\rangle \mid a \mapsto \mathsf{locked}\ \ell') \\
\longrightarrow^{+}\ &(\nu a)(\langle \mathsf{close}\, c\rangle \mid a \mapsto \mathsf{locked}\ \ell') \\
\longrightarrow\ &\langle \mathsf{close}\, c\rangle
\end{aligned}$$

If the channel endpoint is passed to a function that uses the argument more than once, blame will be raised. Let *bar* be $\lambda_{\mathsf{un}} x.\mathsf{close}\,(x\colon \star \overset{\ell}{\Rightarrow} \mathsf{end}_!); \mathsf{close}\,(x\colon \star \overset{\ell}{\Rightarrow} \mathsf{end}_!)$ and observe that $bar\,(c\colon \mathsf{end}_! \overset{\ell'}{\Rightarrow} \star)$ reduces as follows:

$$\begin{aligned}
&\langle bar\,(c\colon \mathsf{end}_! \overset{\ell'}{\Rightarrow} \star)\rangle \\
\longrightarrow\ &\langle bar\,(c\colon \mathsf{end}_! \overset{\ell'}{\Rightarrow} \circledast \overset{\ell'}{\Rightarrow} \star)\rangle \\
\longrightarrow\ &(\nu a)(\langle bar\, a\rangle \mid a \mapsto c\colon \mathsf{end}_! \overset{\ell'}{\Rightarrow} \circledast \overset{\ell'}{\Rightarrow} \star) \\
\longrightarrow\ &(\nu a)(\langle \mathsf{close}\,(a\colon \star \overset{\ell}{\Rightarrow} \mathsf{end}_!); \mathsf{close}\,(a\colon \star \overset{\ell}{\Rightarrow} \mathsf{end}_!)\rangle \mid a \mapsto c\colon \mathsf{end}_! \overset{\ell'}{\Rightarrow} \circledast \overset{\ell'}{\Rightarrow} \star) \\
\longrightarrow\ &(\nu a)(\langle \mathsf{close}\,(c\colon \mathsf{end}_! \overset{\ell'}{\Rightarrow} \circledast \overset{\ell'}{\Rightarrow} \star \overset{\ell}{\Rightarrow} \mathsf{end}_!); \mathsf{close}\,(a\colon \star \overset{\ell}{\Rightarrow} \mathsf{end}_!)\rangle \mid a \mapsto \mathsf{locked}\ \ell') \\
\longrightarrow^{+}\ &(\nu a)(\langle \mathsf{close}\, c; \mathsf{close}\,(a\colon \star \overset{\ell}{\Rightarrow} \mathsf{end}_!)\rangle \mid a \mapsto \mathsf{locked}\ \ell')
\end{aligned}$$

Then, parallel composition with a process waiting at the other end d of the endpoint c will raise blame as follows:

$$\begin{aligned}
&(\nu c, d)(\langle bar\,(c\colon \mathsf{end}_! \overset{\ell'}{\Rightarrow} \star)\rangle \mid \langle \mathsf{wait}\, d\rangle) \\
\longrightarrow^{+}\ &(\nu c, d)(\nu a)(\langle \mathsf{close}\, c; \mathsf{close}\,(a\colon \star \overset{\ell}{\Rightarrow} \mathsf{end}_!)\rangle \mid a \mapsto \mathsf{locked}\ \ell' \mid \langle \mathsf{wait}\, d\rangle)) \\
\longrightarrow\ &(\nu c, d)(\nu a)(\langle \mathsf{close}\,(a\colon \star \overset{\ell}{\Rightarrow} \mathsf{end}_!)\rangle \mid a \mapsto \mathsf{locked}\ \ell' \mid \langle ()\rangle) \\
\longrightarrow\ &(\nu c, d)(\nu a)(\mathsf{blame}\ \overline{\ell}'\ \ell\ \emptyset \mid a \mapsto \mathsf{locked}\ \ell')
\end{aligned}$$

3.2.4 *External language GGV$_e$*

Having defined the internal language, we introduce the external language GGV$_e$, in which source programs are written. The syntax of expressions of GGV$_e$ is presented in Figure 8. For ease of type checking, variable declarations in functions and channel endpoint

https://doi.org/10.1017/S0956796819000169 Published online by Cambridge University Press

$$\text{Expressions} \quad \mathbb{e}, \mathbb{f} ::= z \mid () \mid \lambda_m x{:}T.\,\mathbb{e} \mid \mathbb{e}\,\mathbb{f} \mid (\mathbb{e}, \mathbb{f})_m \mid \mathsf{let}\, x, y = \mathbb{e}\, \mathsf{in}\, \mathbb{f} \mid \mathsf{fork}\, \mathbb{e}$$
$$\mid \mathsf{new}\, S \mid \mathsf{send}\, \mathbb{e}\, \mathbb{f} \mid \mathsf{receive}\, \mathbb{e} \mid \mathsf{select}\, l\, \mathbb{e} \mid \mathsf{case}\, \mathbb{e}\, \mathsf{of}\, \{l_i : x_i.\, \mathbb{e}_i\}_{i \in I}$$
$$\mid \mathsf{close}\, \mathbb{e} \mid \mathsf{wait}\, \mathbb{e}$$

Fig. 8. Expressions in GGV$_e$.

Matching $\boxed{T \triangleright U}$

$$\begin{array}{lll}
T \to_m U \triangleright T \to_m U & \star \triangleright \star \to_{\mathsf{lin}} \star & \\
T \times_m U \triangleright T \times_m U & \star \triangleright \star \times_{\mathsf{lin}} \star & \\
!T.S \triangleright !T.S & \star \triangleright !\star.\circledast & \circledast \triangleright !\star.\circledast \\
?T.S \triangleright ?T.S & \star \triangleright ?\star.\circledast & \circledast \triangleright ?\star.\circledast \\
 & & \\
\oplus\{l_i : S_i\}_{i \in I} \triangleright \oplus\{l_j : S_j\} \quad (j \in I) & \star \triangleright \oplus\{l_j : \circledast\} & \circledast \triangleright \oplus\{l_j : \circledast\} \\
\&\{l_i : S_i\}_{i \in I} \triangleright \&\{l_j : S_j\}_{j \in J} \quad (I \subseteq J) & \star \triangleright \&\{l_j : \circledast\}_{j \in J} & \circledast \triangleright \&\{l_j : \circledast\}_{j \in J}
\end{array}$$

Fig. 9. Matching.

creations are explicitly typed. There are no processes in GGV$_e$: a program is a well-typed closed expression and it is translated to a GGV$_i$ expression before it runs.

The type system of GGV$_e$ adheres to standard practice for gradually typed languages (Siek *et al.*, 2015b; Cimini & Siek, 2016), but requires a few adaptations to cater for features not covered in previous work. We first introduce a few auxiliary definitions used in typing rules. Figure 9 defines the matching relation $T \triangleright U$ (Cimini & Siek, 2016). Roughly speaking, $T \triangleright U$ means that T can be used, after necessary runtime checking, as U. The second and third columns declare that, if T is $\star$ or $\circledast$, then it can be used as any type or session type, respectively. Otherwise, the matching relation extracts substructure, i.e., the domain type, the codomain type, the first-element type, and so on, from T. So, we have neither $\star \triangleright \mathsf{unit}$ nor $\circledast \triangleright \mathsf{end}_!$ or $\circledast \triangleright \mathsf{end}_?$.

Matching for the internal and external choice types is slightly involved as it has to cater for subtyping. Matching for internal choice is invoked in the type rule for an expression $\mathsf{select}\, l\, \mathbb{e}$. Thanks to subtyping, the type of $\mathbb{e}$ can be any internal choice with a branch for label l. Hence, matching only asks for the presence of this single label and extracts its residual.

Dually, matching for external choice is invoked in the rule for a $\mathsf{case}\, \mathbb{e}\, \mathsf{of} \ldots$ expression. Again due to subtyping, the case expression can check more labels than provided by the type of $\mathbb{e}$. Hence, matching allows extra branches to be checked with arbitrary residual types ($l_j : S_j$ in the definition) while extracting the residual types for all branches provided by $\mathbb{e}$.

Obtaining the result type of a case expression from the types of its branches requires a join operation $T \vee U$ that ensures that its result is (in a certain sense) a supertype of both T and U. Figure 10 contains the definitions of join and its companion meet, which is needed in contravariant positions of the type. Both operations are partial: join or meet is undefined for cases other than those listed in Figure 10.

Join of two $\oplus$-types can be obtained by taking the joins of the types associated with common labels. Note that labels where the joins $S_i \vee R_i$ do not exist will be dropped. On the other hand, the label set of the join of two &-types is the union of the two label sets

https://doi.org/10.1017/S0956796819000169 Published online by Cambridge University Press

Multiplicity join and meet $\boxed{m \vee n \quad m \wedge n}$

$$\begin{array}{llll} \mathsf{un} \vee \mathsf{un} = \mathsf{un} & \mathsf{un} \vee \mathsf{lin} = \mathsf{lin} & \mathsf{lin} \vee \mathsf{un} = \mathsf{lin} & \mathsf{lin} \vee \mathsf{lin} = \mathsf{lin} \\ \mathsf{un} \wedge \mathsf{un} = \mathsf{un} & \mathsf{un} \wedge \mathsf{lin} = \mathsf{un} & \mathsf{lin} \wedge \mathsf{un} = \mathsf{un} & \mathsf{lin} \wedge \mathsf{lin} = \mathsf{lin} \end{array}$$

Type join $\boxed{T \vee U}$

$$\begin{aligned}
\mathsf{unit} \vee \mathsf{unit} &= \mathsf{unit} \\
(T \to_m U) \vee (T' \to_n U') &= (T \wedge T') \to_{m \vee n} (U \vee U') \\
(T \times_m U) \vee (T' \times_n U') &= (T \vee T') \times_{m \vee n} (U \vee U') \\
!T.S \vee !T'.S' &= !(T \wedge T').(S \vee S') \\
?T.S \vee ?T'.S' &= ?(T \vee T').(S \vee S') \\
\oplus\{l_i : S_i\}_{i \in I} \vee \oplus\{l_j : R_j\}_{j \in J} &= \oplus\{l_i : S'_i \mid S'_i = S_i \vee R_i \text{ is defined and } i \in I \cap J\} \\
\&\{l_i : S_i\}_{i \in I} \vee \&\{l_j : R_j\}_{j \in J} &= \&\{l_i : S_i\}_{i \in I \setminus J} \cup \{l_k : S_k \vee R_k\}_{k \in I \cap J} \cup \{l_j : R_j\}_{j \in J \setminus I} \\
\mathsf{end}_! \vee \mathsf{end}_! &= \mathsf{end}_! \\
\mathsf{end}_? \vee \mathsf{end}_? &= \mathsf{end}_? \\
\star \vee T &= T \\
T \vee \star &= T \\
\circledast \vee S &= S \\
S \vee \circledast &= S
\end{aligned}$$

Type meet $\boxed{T \wedge U}$

$$\begin{aligned}
\mathsf{unit} \wedge \mathsf{unit} &= \mathsf{unit} \\
(T \to_m U) \wedge (T' \to_n U') &= (T \vee T') \to_{m \wedge n} (U \wedge U') \\
(T \times_m U) \wedge (T' \times_n U') &= (T \wedge T') \times_{m \wedge n} (U \wedge U') \\
!T.S \wedge !T'.S' &= !(T \vee T').(S \wedge S') \\
?T.S \wedge ?T'.S' &= ?(T \wedge T').(S \wedge S') \\
\oplus\{l_i : S_i\}_{i \in I} \wedge \oplus\{l_j : R_j\}_{j \in J} &= \oplus\{l_i : S_i\}_{i \in I \setminus J} \cup \{l_k : S_k \wedge R_k\}_{k \in I \cap J} \cup \{l_j : R_j\}_{j \in J \setminus I} \\
\&\{l_i : S_i\}_{i \in I} \wedge \&\{l_j : R_j\}_{j \in J} &= \&\{l_i : S'_i \mid S'_i = S_i \wedge R_i \text{ is defined and } i \in I \cap J\} \\
\mathsf{end}_! \wedge \mathsf{end}_! &= \mathsf{end}_! \\
\mathsf{end}_? \wedge \mathsf{end}_? &= \mathsf{end}_? \\
\star \wedge T &= T \\
T \wedge \star &= T \\
\circledast \wedge S &= S \\
S \wedge \circledast &= S
\end{aligned}$$

Fig. 10. Join and meet of types.

from the input. For the common labels in $I \cap J$, the joins $S_k \vee R_k$ must exist. Join or meet is undefined if the resulting type is $\oplus\{\}$ or $\&\{\}$ (with the empty set of labels) as they are ill-formed types.

Without the last four clauses, which deal with $\star$ and $\circledast$, the definitions of the join and meet coincide with those for ordinary subtyping. This is motivated by the static embedding property of the Criteria for Gradual Typing (Siek *et al.*, 2015b), which requires the typability of a GGV$_e$ term without $\star$ (or $\circledast$ in our case) to be the same as the typability under the GV typing rules. There are a few choices for the join (and meet) of $\star$ and other types and

https://doi.org/10.1017/S0956796819000169 Published online by Cambridge University Press

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

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

$$\frac{\Gamma \vdash \mathbb{e}_1 : T_1 \qquad \Delta \vdash \mathbb{e}_2 : T_2 \qquad T_1 \rhd T_{11} \to_m T_{12} \qquad T_2 \lesssim T_{11}}{\Gamma \circ \Delta \vdash \mathbb{e}_1\, \mathbb{e}_2 : T_{12}}$$

$$\frac{\Gamma \vdash \mathbb{e} : T \qquad \Delta \vdash \mathbb{f} : U \qquad m^{>}(T) \qquad m^{>}(U)}{\Gamma \circ \Delta \vdash (\mathbb{e}, \mathbb{f})_m : T \times_m U}$$

$$\frac{\Gamma \vdash \mathbb{e} : T \qquad T \rhd T_1 \times_m T_2 \qquad \Delta, x : T_1, y : T_2 \vdash \mathbb{f} : U}{\Gamma \circ \Delta \vdash \mathsf{let}\, x, y = \mathbb{e}\, \mathsf{in}\, \mathbb{f} : U}$$

$$\frac{\Gamma \vdash \mathbb{e} : T \qquad T \sim \mathsf{unit}}{\Gamma \vdash \mathsf{fork}\, \mathbb{e} : \mathsf{unit}} \qquad \frac{\mathsf{un}(\Gamma)}{\Gamma \vdash \mathsf{new}\, S : S \times_{\mathsf{lin}} \overline{S}}$$

$$\frac{\Gamma \vdash \mathbb{e}_1 : T_1 \qquad \Delta \vdash \mathbb{e}_2 : T_2 \qquad T_2 \rhd !T_3.S \qquad T_1 \lesssim T_3}{\Gamma \circ \Delta \vdash \mathsf{send}\, \mathbb{e}_1\, \mathbb{e}_2 : S}$$

$$\frac{\Gamma \vdash \mathbb{e} : T_1 \qquad T_1 \rhd ?T_2.S}{\Gamma \vdash \mathsf{receive}\, \mathbb{e} : T_2 \times_{\mathsf{lin}} S} \qquad \frac{\Gamma \vdash \mathbb{e} : T \qquad T \rhd \oplus\{l_j : S_j\}}{\Gamma \vdash \mathsf{select}\, l_j\, \mathbb{e} : S_j}$$

$$\frac{\Gamma \vdash \mathbb{e} : T \qquad T \rhd \&\{l_j : R_j\}_{j \in J} \qquad (\Delta, x_j : R_j \vdash \mathbb{e}_j : U_j)_{j \in J} \qquad U = \bigvee\{U_j\}_{j \in J}}{\Gamma \circ \Delta \vdash \mathsf{case}\, \mathbb{e}\, \mathsf{of}\, \{l_j : x_j.\, \mathbb{e}_j\}_{j \in J} : U}$$

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

Typing programs $\boxed{\mathbb{e}\ \mathbf{prog}}$

$$\frac{\vdash \mathbb{e} : T \qquad \mathsf{un}(T)}{\mathbb{e}\ \mathbf{prog}}$$

Fig. 11. Expression typing in GGV_e.

we choose $\star \vee T$ to be T for any T because, as we prove later, our join then corresponds to the least upper bound with respect to negative subtyping (Wadler & Findler, 2009), which is formally defined later, and we can construct a type-checking algorithm that produces a minimal type with respect to the negative subtyping. (The least upper bound with respect to positive subtyping is not a good choice because $\mathsf{int} \vee \mathsf{bool} = \star$ holds, invalidating the static embedding property.)

Typing rules are presented in Figure 11. The matching relation is used in elimination rules. To obtain a syntax-directed inference system, the subsumption is merged into function application, sending, select, and case. Moreover, subtyping is replaced with consistent subtyping. The type of the whole case expression is obtained by joining the types of the branches. Finally, the judgment $\mathbb{e}$ **prog** means that $\mathbb{e}$ is a Gradual GV program, which is a closed, well-typed GGV_e expression of unrestricted type. Cast insertion discussed in the following translates a program to a GGV_i expression e, which runs as a process $\langle e \rangle$. For example, we can derive

$$\vdash \lambda_{\mathsf{un}} o : \star.\lambda_{\mathsf{un}} c : \circledast.\mathsf{close}\,(\mathsf{send}\, o\, c) : \star \to_{\mathsf{un}} \circledast \to_{\mathsf{un}} \mathsf{unit}.$$

We also develop a type-checking algorithm for GGV_e by following the standard approach (Kobayashi *et al.*, 1999; Walker, 2005). We define an algorithm $\textsc{CheckExpr}(\Gamma, \mathbb{e})$, which takes a type environment Γ and an expression $\mathbb{e}$ and returns a type T of $\mathbb{e}$ and the set X of linear variables in $\mathbb{e}$. We avoid nondeterminism involved in environment splitting by introducing X, which is used to check whether subexpressions do not use the same (linear) variable more than once. We present the algorithm in full and

https://doi.org/10.1017/S0956796819000169 Published online by Cambridge University Press

Cast insertion $\boxed{\Gamma \vdash \mathbb{e} \rightsquigarrow f : T}$

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

$$\frac{\Gamma \vdash \mathbb{e} \rightsquigarrow e : T_1 \qquad \Delta \vdash \mathbb{f} \rightsquigarrow f : T_2 \qquad T_1 \triangleright T_{11} \to_m T_{12} \qquad T_2 \lesssim T_{11}}{\Gamma \circ \Delta \vdash \mathbb{e}\mathbb{f} \rightsquigarrow (e : T_1 \overset{p}{\Rightarrow}_? T_{11} \to_m T_{12})\,(f : T_2 \overset{p}{\Rightarrow}_? T_{11}) : T_{12}}$$

$$\frac{\Gamma \vdash \mathbb{e} \rightsquigarrow e : T \qquad \Delta \vdash \mathbb{f} \rightsquigarrow f : U \qquad m^{:>}(T) \qquad m^{:>}(U)}{\Gamma \circ \Delta \vdash (\mathbb{e}, \mathbb{f})_m \rightsquigarrow (e, f)_m : T \times_m U}$$

$$\frac{\Gamma \vdash \mathbb{e} \rightsquigarrow e : T \qquad T \triangleright T_1 \times_m T_2 \qquad \Delta, x : T_1, y : T_2 \vdash \mathbb{f} \rightsquigarrow f : U}{\Gamma \circ \Delta \vdash \mathsf{let}\, x, y = \mathbb{e}\, \mathsf{in}\, \mathbb{f} \rightsquigarrow \mathsf{let}\, x, y = (e : T \overset{p}{\Rightarrow}_? T_1 \times_m T_2)\, \mathsf{in}\, f : U}$$

$$\frac{\Gamma \vdash \mathbb{e} \rightsquigarrow e : T \qquad T \sim \mathsf{unit}}{\Gamma \vdash \mathsf{fork}\, \mathbb{e} \rightsquigarrow \mathsf{fork}\,(e : T \overset{p}{\Rightarrow}_? \mathsf{unit}) : \mathsf{unit}} \qquad \frac{\mathsf{un}(\Gamma)}{\Gamma \vdash \mathsf{new}\, S \rightsquigarrow \mathsf{new} : S \times_{\mathsf{lin}} \overline{S}}$$

$$\frac{\Gamma \vdash \mathbb{e} \rightsquigarrow e : T_1 \qquad \Delta \vdash \mathbb{f} \rightsquigarrow f : T_2 \qquad T_2 \triangleright !T_3.S \qquad T_1 \lesssim T_3}{\Gamma \circ \Delta \vdash \mathsf{send}\, \mathbb{e}\mathbb{f} \rightsquigarrow \mathsf{send}\,(e : T_1 \overset{p}{\Rightarrow}_? T_3)\,(f : T_2 \overset{p}{\Rightarrow}_? !T_3.S) : S}$$

$$\frac{\Gamma \vdash \mathbb{e} \rightsquigarrow e : T_1 \qquad T_1 \triangleright ?T_2.S}{\Gamma \vdash \mathsf{receive}\, \mathbb{e} \rightsquigarrow \mathsf{receive}\,(e : T_1 \overset{p}{\Rightarrow}_? ?T_2.S) : T_2 \times_{\mathsf{lin}} S}$$

$$\frac{\Gamma \vdash \mathbb{e} \rightsquigarrow e : T \qquad T \triangleright \oplus\{l_j : S_j\}}{\Gamma \vdash \mathsf{select}\, l_j\, \mathbb{e} \rightsquigarrow \mathsf{select}\, l_j\,(e : T \overset{p}{\Rightarrow}_? \oplus\{l_j : S_j\}) : S_j}$$

$$\frac{\Gamma \vdash \mathbb{e} \rightsquigarrow e : T \qquad T \triangleright \&\{l_j : R_j\}_{j \in J} \qquad (\Delta, x_j : R_j \vdash \mathbb{e}_j \rightsquigarrow f_j : U_j)_{j \in J} \qquad U = \bigvee\{U_j\}_{j \in J}}{\Gamma \circ \Delta \vdash \mathsf{case}\, \mathbb{e}\, \mathsf{of}\, \{l_j : x_j.\, \mathbb{e}_j\}_{j \in J} \rightsquigarrow \mathsf{case}\,(e : T \overset{p}{\Rightarrow}_? \&\{l_j : R_j\}_{j \in J})\, \mathsf{of}\, \{l_j : x_j.f_j : U_j \overset{p}{\Rightarrow}_? U\}_{j \in J} : U}$$

$$\frac{\Gamma \vdash \mathbb{e} \rightsquigarrow e : T \qquad T \sim \mathsf{end}_!}{\Gamma \vdash \mathsf{close}\, \mathbb{e} \rightsquigarrow \mathsf{close}\,(e : T \overset{p}{\Rightarrow}_? \mathsf{end}_!) : \mathsf{unit}} \qquad \frac{\Gamma \vdash \mathbb{e} \rightsquigarrow e : T \qquad T \sim \mathsf{end}_?}{\Gamma \vdash \mathsf{wait}\, \mathbb{e} \rightsquigarrow \mathsf{wait}\,(e : T \overset{p}{\Rightarrow}_? \mathsf{end}_?) : \mathsf{unit}}$$

Fig. 12. Cast insertion.

prove its correctness in Appendix 6. In particular, the algorithm is shown to compute, for given Γ and $\mathbb{e}$, a minimal type with respect to negative subtyping (if a typing exists).

3.2.5 *Cast-inserting translation*

A well-typed GGV$_e$ expression is translated to a GGV$_i$ expression by dropping type annotations and inserting casts. Figure 12 presents cast insertion. The judgment $\Gamma \vdash \mathbb{e} \rightsquigarrow f : T$ means that "under type environment Γ, a GGV$_e$ expression $\mathbb{e}$ is translated to a GGV$_i$ expression f at type T." Most rules are straightforward: casts are inserted where the matching or consistent subtyping is used. In each rule, blame label p is supposed to be fresh and positive. The notation $f : T \overset{p}{\Rightarrow}_? U$ is used to avoid inserting unnecessary casts.

$$f : T \overset{p}{\Rightarrow}_? U = \begin{cases} f & \text{if } T <: U \\ f : T \overset{p}{\Rightarrow} U & \text{otherwise} \end{cases}$$

Thanks to this optimization, we can show that a program that does not use $\star$ or $\circledast$ is translated to a cast-free GGV$_i$ expression, whose behavior obviously coincides with GV.

For example, we can derive

$$\begin{aligned} &\vdash \lambda_{\mathsf{un}} o : \star.\lambda_{\mathsf{un}} c : \circledast.\mathsf{close}\,(\mathsf{send}\; o\; c) \\ &\qquad \rightsquigarrow \lambda_{\mathsf{un}} o.\lambda_{\mathsf{un}} c.\mathsf{close}\,((\mathsf{send}\; o\; (c : \circledast \overset{p}{\Rightarrow} !\star.\circledast)) : \circledast \overset{q}{\Rightarrow} \mathsf{end}_!) : \star \to_{\mathsf{un}} \circledast \to_{\mathsf{un}} \mathsf{unit} \end{aligned}$$

for some p and q.

https://doi.org/10.1017/S0956796819000169 Published online by Cambridge University Press

$$
\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. 13. Embedding of the unityped calculus.

3.2.6 Embedding

One desideratum for a gradual typing system—if it is equipped with dynamic typing—is that it is possible to embed an untyped (or rather, unityped) language within it (Siek *et al.*, 2015b). An embedding of an untyped variant of GV into GGV_i is given in Figure 13. 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 $\star$ and multiplicities are implicitly assumed to be un. The embedding extends that of (Wadler & Findler, 2009) for the untyped lambda calculus into the blame calculus.

4 Results

We study some of the basic properties (Siek *et al.*, 2015b) of Gradual GV in this section. They include (1) type safety of GGV_i and (2) blame safety of GGV_i, (3) conservative typing of GGV_e over GV, and (4) the gradual guarantee for GGV_e. Since GGV_i do not guarantee deadlock freedom, type safety is stated as the combination of preservation and absence of runtime errors, rather than progress. We show that (1)–(3) hold with their proof sketches. For (4), we show that GGV_e does *not* satisfy the gradual guarantee.

4.1 Preservation and absence of runtime errors for GGV_i

We show preservation and absence of runtime errors for GGV_i. The basic structure of the proof follows Gay and Vasconcelos (2010). In proofs, we often use inversion properties for the typing relation, such as "if $\Gamma \vdash x : T$, then $\Gamma = \Gamma', x : S$ for some S and Γ' such that $S <: T$ and $\mathsf{un}(\Gamma')$," without even stating. They are easy (but tedious) to state and prove because the only rule that makes typing rules not syntax-directed is T-Sub (see, for example, (Pierce, 2002) for details). Similarly, we omit inversion for subtyping, which is syntax-directed.

https://doi.org/10.1017/S0956796819000169 Published online by Cambridge University Press

Lemma 1 (Weakening). *If* $\Gamma \vdash e\colon T$ *and* $\mathsf{un}(U)$*, then* $\Gamma, x\colon U \vdash e\colon T$.

Proof. By induction on $\Gamma \vdash e\colon T$. □

Lemma 2 (Strengthening). *If* $\Gamma, x\colon U \vdash e\colon T$ *and* x *does not occur free in* e*, then* $\Gamma \vdash e\colon T$.

Proof. By induction on $\Gamma, x\colon U \vdash e\colon T$. □

Lemma 3 (Preservation for $\equiv$). *If* $P \equiv Q$*, then* $\Gamma \vdash P$ *if and only if* $\Gamma \vdash Q$.

Proof. By induction on $P \equiv Q$. Use Lemmas 1, 2, and basic properties of context splitting (Vasconcelos, 2012; Walker, 2005) for the scope extrusion rules. □

Lemma 4. *If* $\Gamma = \Gamma_1 \circ \Gamma_2$ *and* $\mathsf{un}(\Gamma_1)$*, then* $\Gamma = \Gamma_2$.

Proof. By induction on $\Gamma = \Gamma_1 \circ \Gamma_2$. □

Lemma 5. *If* $\Gamma \vdash v : T$ *and* $\mathsf{un}(T)$*, then* $\mathsf{un}(\Gamma)$.

Proof. By case analysis on the last rule used to derive $\Gamma \vdash v : T$. □

Lemma 6 (Substitution). *If* $\Gamma_1 \vdash v : U$ *and* $\Gamma_2, x : U \vdash e : T$ *and* $\Gamma = \Gamma_1 \circ \Gamma_2$*, then* $\Gamma \vdash e[v/x] : T$.

Proof. By induction on $\Gamma_2, x : U \vdash e : T$ with case analysis on the last derivation rule used. We show main cases as follows:

Case (variables): If $e = x$ and $T = U$ and $\mathsf{un}(\Gamma_2)$, then we have, by Lemma 4, $\Gamma = \Gamma_1$, finishing the case. If $e = y \neq x$, then Lemma 2 finishes the case.
Case (applications): We have $e = e_1\ e_2$ and $\Gamma_{11} \vdash e_1 : T_2 \to_m T$ and $\Gamma_{12} \vdash e_2 : T_2$ and $\Gamma, x : U = \Gamma_{11} \circ \Gamma_{12}$. We have two subcases depending on whether $\mathsf{un}(U)$ or not.

> **Subcase** $\mathsf{un}(U)$: We have $\Gamma_{11} = \Gamma'_{11}, x : U$ and $\Gamma_{12} = \Gamma'_{12}, x : U$ and $\Gamma = \Gamma'_{11} \circ \Gamma'_{12}$. The induction hypothesis give us $\Gamma'_{11} \circ \Gamma_2 \vdash e_1[v/x] : T_2 \to_m T$ and $\Gamma'_{12} \circ \Gamma_2 \vdash e_2[v/x] : T_2$. By Lemma 5, we have $\mathsf{un}(\Gamma_2)$. The typing rule for applications shows $(\Gamma'_{11} \circ \Gamma_2) \circ (\Gamma'_{12} \circ \Gamma_2) \vdash (e_1\ e_2)[v/x] : T$. Lemma 4 finishes the subcase.
> **Subcase** $\mathsf{lin}(U)$: either (1) $\Gamma_{11} = \Gamma'_{11}$ and $\Gamma_{12} = \Gamma'_{12}, x : U$ and $\Gamma = \Gamma'_{11} \circ \Gamma'_{12}$, in which case we have $\Gamma'_{12} \vdash e_2[v/x] : T_1$ by the induction hypothesis and also $e_1[v/x] = e_1$ and the typing rule for applications finishes; or (2) $\Gamma_{11} = \Gamma'_{11}, x : U$ and $\Gamma_{12} = \Gamma'_{12}$ and $\Gamma = \Gamma'_{11} \circ \Gamma'_{12}$, in which case the conclusion is similarly proved. □

The following two lemmas are adapted from earlier work (Gay & Vasconcelos, 2010).

Lemma 7 (Sub-derivation introduction). *If* $\mathscr{D}$ *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* $\mathscr{D}$ *has a sub-derivation* $\mathscr{D}'$ *concluding* $\Gamma_2 \vdash e : U$ *and the position of* $\mathscr{D}'$ *in* $\mathscr{D}$ *corresponds to the position of the hole in* E.

Proof. By induction on E. □

https://doi.org/10.1017/S0956796819000169 Published online by Cambridge University Press

Lemma 8 (Sub-derivation elimination)**.** $\Gamma \vdash E[f] : T$ *holds, if*

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

Proof. By induction on E. □

Lemma 9. *If* $\Gamma \vdash e : T$, *then* $\mathsf{flv}(\Gamma) = \mathsf{flv}(e)$.

Proof. Easy induction on $\Gamma \vdash e : T$. □

Theorem 1 (Preservation for expressions)**.** *If* $e \longrightarrow f$ *and* $\Gamma \vdash e : T$, *then* $\Gamma \vdash f : T$.

Proof. By rule induction on the first hypothesis. For β-reduction and let we use the substitution lemma (Lemma 6) and inversion of the typing relation. □

Theorem 2 (Preservation for processes)**.** *If* $P \longrightarrow Q$ *and* $\Gamma \vdash P$, *then* $\Gamma \vdash Q$.

Proof. By rule induction on the first hypothesis, using basic properties of context splitting (Walker, 2005; Vasconcelos, 2012) and weakening (Lemma 1). Rules that make use of context use sub-derivation introduction (Lemma 7) to build the derivation for the hypothesis, and sub-derivation elimination (Lemma 8) to build the derivation for the conclusion. Rules for reduction to blame use Lemma 9. Reduction underneath parallel composition and scope restriction follow by induction. The rule for $\equiv$ uses Lemma 3. Closure under evaluation contexts uses Theorem 1. □

Lemma 10 (Ground types, subtyping, and consistent subtyping)**.**

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

Proof.

1. By case analysis on T.
2. By case analysis on S.
3. By case analysis on $\mathbf{T}$ and $\mathbf{U}$.
4. By case analysis on $\mathbf{S}$ and $\mathbf{R}$. □

Lemma 11 (Canonical forms)**.** *Suppose that* $\Gamma \vdash v : T$ *where* Γ *contains session types and* $\star$, *only.*

1. *If* $T = \star$, *then either* $v = w : \mathbf{T} \overset{p}{\Rightarrow} \star$ *with* $\mathsf{un}(\mathbf{T})$ *or* $v = a$.
2. *If* $T = S$, *then either* $v = c$ *or* $v = w : \mathbf{S} \overset{p}{\Rightarrow} \circledast$ *and* $S = \circledast$ *or* $v = w : R_1 \overset{p}{\Rightarrow} R_2$ *with* $R_2 <: S$.

https://doi.org/10.1017/S0956796819000169 Published online by Cambridge University Press

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* $v = w\colon T_1 \to_{n_1} T_2 \overset{p}{\Rightarrow} U_1' \to_{n_2} U_2'$ *with* $n_2 <: m$ *and* $U_1 <: U_1'$ *and* $U_2' <: U_2$.
5. *If* $T = T_1 \times_m T_2$, *then* $v = (w_1, w_2)_n$ *with* $n <: m$.

Proof. By induction on the derivation on $\Gamma \vdash v : T$. □

Theorem 3 (Progress for expressions)**.** *Suppose that* $\Gamma \vdash e : T$ *and that* Γ *only contains channel endpoints and references. Then exactly one of the following cases holds:*

1. *e is a value,*
2. $e \longrightarrow f$ *(as an expression),*
3. $e = E[f]$ *and* f *is one of the GV operations:* $\mathsf{fork}\, f'$, new, $\mathsf{send}\, v\, c$, $\mathsf{receive}\, c$, $\mathsf{select}\, l\, c$, $\mathsf{case}\, c\, \mathsf{of}\{l_i : x_i.e_i\}$, $\mathsf{close}\, c$, *or* $\mathsf{wait}\, c$,
4. $e = E[f]$ *and* f *is a Gradual GV operation:*
   - $w\colon \mathbf{T} \overset{p}{\Rightarrow} \star$, *with* $\mathsf{lin}(\mathbf{T})$,
   - $a\colon \star \overset{p}{\Rightarrow} \mathbf{U}$,
   - $(v\colon \mathbf{T} \overset{p}{\Rightarrow} \star)\colon \star \overset{q}{\Rightarrow} \mathbf{U}$, *with* $\mathsf{un}(\mathbf{T})$,
   - $(v\colon \mathbf{S} \overset{p}{\Rightarrow} \circledast)\colon \circledast \overset{q}{\Rightarrow} \mathbf{R}$, *or*
   - $c\colon \circledast \overset{p}{\Rightarrow} \mathbf{S}$.

Proof. By induction on expressions, using Canonical forms (Lemma 11). □

The notion of *runtime errors* helps us state 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.

https://doi.org/10.1017/S0956796819000169 Published online by Cambridge University Press

For processes, rather than a progress result, we present a type safety result as our type system does not rule out deadlocks, which are formed by a series of processes each waiting for the next in a circular arrangement; these are exactly the deadlocked processes of GV. Our result holds both for GV and Gradual GV alike. The condition on Γ in the statement is to exclude processes getting stuck due to a free variable in an application (xe) or a pair destruction ($\mathsf{let}\, a, b = x\, \mathsf{in}\, e$).

Theorem 4 (Absence of runtime 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.*

Proof. By induction on the length of reduction steps $P \longrightarrow^* Q$. For the base case, where $P = Q$, we show P is not an error by showing all error processes cannot be well typed.

All cases use Lemma 7 and inversion of the typing relation. The cases for application and let follow from the fact that Γ does not contain function or pair types. The third case follows from the fact that c, being the subject of expressions, is of a linear type, hence cannot occur in two distinct processes. The fourth case follows from the fact that typability implies that c and d are of dual types, which in turn implies $\mathsf{agree}^{\{c,d\}}(e, f)$. □

4.2 Blame safety

Following Wadler & Findler (2009), we introduce three new subtyping relations: $<:^+$, $<:^-$, and $\sqsubseteq$, called positive, negative, and naive subtyping (also known as precision), respectively, in Figure 14, 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 labeled p (called *positive* blame) or $\overline{p}$ (called *negative* blame). The original subtyping relation $T <: U$ of GGV_i 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 $T \sqsubseteq U$ 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 & 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:

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

The four subtyping relations are closely related. In previous work (Wadler & Findler, 2009; Siek *et al.*, 2015a) one has that proper subtyping decomposes into positive and negative subtyping, which—after reversing the order on negative subtyping—recompose into naive subtyping. Here we have three-quarters of the previous result.

Theorem 5 (3/4 Tangram)**.**

1. *$T <: U$ implies $T <:^+ U$ and $T <:^- U$.*
2. *$S <: R$ implies $S <:^+ R$ and $S <:^- R$*
3. *$T \sqsubseteq U$ if and only if $T <:^+ U$ and $U <:^- T$.*
4. *$S \sqsubseteq R$ if and only if $S <:^+ R$ and $R <:^- S$*

https://doi.org/10.1017/S0956796819000169 Published online by Cambridge University Press

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

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

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

$$\frac{T' <:^{\mp} T \quad S <:^{\pm} S'}{!T.S <:^{\pm} !T'.S'} \qquad \frac{T <:^{\pm} T' \quad S <:^{\pm} S'}{?T.S <:^{\pm} ?T'.S'}$$

$$\frac{J \subseteq I \quad (S_j <:^{\pm} R_j)_{j \in J}}{\oplus\{l_i : S_i\}_{i \in I} <:^{\pm} \oplus\{l_j : R_j\}_{j \in J}} \qquad \frac{I \subseteq J \quad (S_i <:^{\pm} R_i)_{i \in I}}{\&\{l_i : S_i\}_{i \in I} <:^{\pm} \&\{l_j : R_j\}_{j \in J}}$$

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

Naive subtyping $\boxed{T \sqsubseteq U}$

$$T \sqsubseteq \star \qquad S \sqsubseteq \circledast$$

$$\mathsf{unit} \sqsubseteq \mathsf{unit} \qquad \frac{T \sqsubseteq T' \quad U \sqsubseteq U'}{T \to_m U \sqsubseteq T' \to_m U'} \qquad \frac{T \sqsubseteq T' \quad U \sqsubseteq U'}{T \times_m U <: T' \times_m U'}$$

$$\frac{T \sqsubseteq T' \quad S \sqsubseteq S'}{!T.S \sqsubseteq !T'.S'} \qquad \frac{T \sqsubseteq T' \quad S \sqsubseteq S'}{?T.S \sqsubseteq ?T'.S'}$$

$$\frac{(S_i \sqsubseteq R_i)_{i \in I}}{\oplus\{l_i : S_i\}_{i \in I} \sqsubseteq \oplus\{l_i : R_i\}_{i \in I}} \qquad \frac{(S_i \sqsubseteq R_i)_{i \in I}}{\&\{l_i : S_i\}_{i \in I} \sqsubseteq \&\{l_i : R_i\}_{i \in I}}$$

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

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. 14. Subtyping and blame safety.

Proof. By induction on types. □

Here the first and second items are an implication, rather than an equivalence as in the third and fourth items and previous work. In order to get an equivalence, we would need to alter subtyping such that $T <: \star$ for all T and $S <: \circledast$ for all S, which would interfere with our Canonical Forms lemma (Lemma 11). However, implication in all four items is sufficient to ensure the most important result, Corollary 1.

The definitions of negative subtyping and naive subtyping have been changed since the conference version of the paper. Now, negative subtyping supports width subtyping and naive does not. This change is motivated by the type system for the external language, in particular the join operation. (See the discussion on the join in Section 3.2.4.)

The following technical result is used in the proof of Theorem 6.

Lemma 12.

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

https://doi.org/10.1017/S0956796819000169 Published online by Cambridge University Press

Proof. (1) A case analysis on T. Lemma 10 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 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 (positive or negative) blame subtyping relation. Figure 14 defines judgments e safe for p and P 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 6 (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 12. Then, the standard function cast is analogous to previous work (Wadler & 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 $P \equiv \Pi(Q \mid R)$ where Q is $\mathsf{blame}\, p\, q\, X$, $\mathsf{blame}\, q\, p\, X$, or $\mathsf{blame}\, p\, X$, for some q and X, and prefix Π of bindings for channel endpoints and references.

Theorem 7 (Progress of safe terms)**.** *If $\Gamma \vdash P$ and P* safe for *p, then $P \not\longrightarrow Q$ where Q blames p.*

Proof. We analyze 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 1 (Well-typed programs cannot 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 the following cases holds::*

- *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 $\overline{q}$.

https://doi.org/10.1017/S0956796819000169 Published online by Cambridge University Press

Wadler and Findler (2009) explain how casting between terms related by naive subtyping always places the blame (if any) on the less-precisely typed term or context, as appropriate.

4.3 Properties of GGV$_e$

Now we turn our attention to GGV$_e$ and prove that cast insertion succeeds for well-typed GGV$_e$ expressions and preserves typing and that the GGV$_e$ typing conservatively extends the GV typing. As we need to relate the judgments of different systems, let $\vdash_e$ denote the GGV$_e$ typing, $\vdash_i$ denote the GGV$_i$ typing, and $\vdash_{\mathrm{GV}}$ denote the GV typing.

Proposition 1 goes back to an observation by Siek and Taha (2007).

Proposition 1 (Consistent Subtyping)**.**

1. *$T_1 \lesssim T_2$ if and only if $T_1 \sim T_1'$ and $T_1' <: T_2$ for some T_1'.*
2. *$T_1 \lesssim T_2$ if and only if $T_1 <: T_2'$ and $T_2' \sim T_2$ for some T_2'.*

Proof. The left-to-right direction is proved by induction on $T_1 \lesssim T_2$ and the right-to-left is by induction on subtyping with case analysis on T_1, T_2', and T_2. □

The next lemma clarifies the relation between subtyping, positive and negative subtyping, and consistent subtyping.

Lemma 13 (Subtyping Hierarchy)**.**

1. $<: \subseteq <:^+ \subseteq \lesssim$.
2. $<: \subseteq <:^- \subseteq \lesssim$.

Proof. $<: \subseteq <:^+$ and $<: \subseteq <:^-$ follow from Theorem 5. $<:^+ \subseteq \lesssim$ and $<:^- \subseteq \lesssim$ are by induction on $T_1 <:^+ T_2$ and $T_1 <:^- T_2$, respectively. □

Lemma 14 (Upper bound and lower bound)**.**

1. *If $T_1 \vee T_2 = U$, then $T_1 <:^- U$ and $T_2 <:^- U$.*
2. *If $T_1 \wedge T_2 = U$, then $U <:^+ T_1$ and $U <:^+ T_2$.*

Proof. By simultaneous induction on $T_1 \vee T_2 = U$ (for the first item) and $T_1 \wedge T_2 = U$ (for the second item). □

Lemma 15 (Least upper bound and greatest lower bound)**.**

1. *If $T_1 <:^- U$ and $T_2 <:^- U$, then there exists some U' such that $T_1 \vee T_2 = U'$ and $U' <:^- U$.*
2. *If $U <:^+ T_1$ and $U <:^+ T_2$, then there exists some U' such that $T_1 \wedge T_2 = U'$ and $U <:^+ U'$.*

Proof. The two items are simultaneously proved by induction on $T_1 <:^- U$ and $U <:^+ T_1$. □

Theorem 8 states that cast insertion succeeds for well-typed external language and preserves typing. A few lemmas are required in preparation.

https://doi.org/10.1017/S0956796819000169 Published online by Cambridge University Press

Lemma 16. *If $T_1 \vee T_2 = U$, then $T_1 \lesssim U$ and $T_2 \lesssim U$.*

Proof. Immediate from Lemmas 13 (1) and 14 (1). □

Lemma 17. *If $T \rhd U$, then $T \lesssim U$.*

Proof. By case analysis on $T \rhd U$. □

Theorem 8 (Cast insertion succeeds and preserves typing)**.** *If $\Gamma \vdash_e \mathbb{e} : T$, then there exists some f such that $\Gamma \vdash \mathbb{e} \rightsquigarrow f : T$ and $\Gamma \vdash_i f : T$.*

Proof. By rules induction on the derivation of $\Gamma \vdash_e \mathbb{e} : T$. We show main cases as follows.

Case application rule: We are given

$$\Gamma = \Gamma_1 \circ \Gamma_2 \qquad \mathbb{e} = \mathbb{e}_1\ \mathbb{e}_2 \qquad T = T_{12}$$
$$\Gamma_1 \vdash \mathbb{e}_1 : T_1 \qquad \Gamma_2 \vdash \mathbb{e}_2 : T_2 \qquad T_1 \rhd T_{11} \to_m T_{12} \qquad T_2 \lesssim T_{11}$$

By $\Gamma_1 \vdash \mathbb{e}_1 : T_1$ and the IH, $\Gamma_1 \vdash \mathbb{e}_1 \rightsquigarrow f_1 : T_1$ and $\Gamma_1 \vdash f_1 : T_1$ for some f_1. By $\Gamma_2 \vdash \mathbb{e}_2 : T_2$ and the IH, $\Gamma_2 \vdash \mathbb{e}_2 \rightsquigarrow f_2 : T_2$ and $\Gamma_2 \vdash f_2 : T_2$ for some f_2. Let

$$f = (f_1 : T_1 \overset{p}{\Rightarrow}_? T_{11} \to_m T_{12})\,(f_2 : T_2 \overset{p}{\Rightarrow}_? T_{11})$$

By the application rule, $\Gamma \vdash \mathbb{e} \rightsquigarrow f : T$.
Let us assume $f_1 : T_1 \overset{p}{\Rightarrow}_? T_{11} \to_m T_{12}$ equals $f_1 : T_1 \overset{p}{\Rightarrow} T_{11} \to_m T_{12}$. (If $f_1 : T_1 \overset{p}{\Rightarrow}_? T_{11} \to_m T_{12}$ equals f_1, we could replace the cast rule with the subsumption rule in what follows.) We also take similar assumptions in other cases.
By $T_1 \rhd T_{11} \to_m T_{12}$ and Lemma 17, $T_1 \lesssim T_{11} \to_m T_{12}$. By $\Gamma_1 \vdash f_1 : T_1$ and the cast rule,

$$\Gamma_1 \vdash (f_1 : T_1 \overset{p}{\Rightarrow} T_{11} \to_m T_{12}) : T_{11} \to_m T_{12}$$

By $\Gamma_2 \vdash f_2 : T_2$ and $T_2 \lesssim T_{11}$ and the cast rule,

$$\Gamma_2 \vdash (f_2 : T_2 \overset{p}{\Rightarrow} T_{11}) : T_{11}$$

Thus, by the application rule, $\Gamma \vdash f : T$.

Case case rule: We are given

$$\Gamma = \Gamma' \circ \Delta \qquad \mathbb{e} = \mathsf{case}\ \mathbb{e}'\ \mathsf{of}\ \{l_j : x_j.\ \mathbb{e}_j\}_{j \in J} \qquad T = U$$
$$\Gamma' \vdash \mathbb{e}' : T' \qquad T' \rhd \&\{l_j : R_j\}_{j \in J} (\Delta, x_j : R_j \vdash \mathbb{e}_j : U_j)_{j \in J} \qquad U = \bigvee \{U_j\}_{j \in J}$$

By $\Gamma' \vdash \mathbb{e}' : T'$ and the IH, $\Gamma' \vdash \mathbb{e}' \rightsquigarrow f : T'$ and $\Gamma' \vdash f' : T'$ for some f'. We take some $j \in J$. By $\Delta, x_j : R_j \vdash \mathbb{e}_j : U_j$ and the IH, we have $\Delta, x_j : R_j \vdash \mathbb{e}_j \rightsquigarrow f_j : U_j$ and $\Delta, x_j : R_j \vdash f_j : U_j$ for some f_j. Let

$$f = \mathsf{case}\,(f' : T \overset{p}{\Rightarrow}_? \&\{l_j : R_j\}_{j \in J})\ \mathsf{of}\ \{l_j : x_j.\ f_j : U_j \overset{p}{\Rightarrow}_? U\}_{j \in J}$$

By the case rule, $\Gamma \vdash \mathbb{e} \rightsquigarrow f : T$.
Next, by $T' \rhd \&\{l_j : R_j\}_{j \in J}$ and Lemma 17, $T' \lesssim \&\{l_j : R_j\}_{j \in J}$. By $\Gamma' \vdash f' : T'$ and the cast rule,

https://doi.org/10.1017/S0956796819000169 Published online by Cambridge University Press

$$\Gamma' \vdash (f' : T' \overset{p}{\Rightarrow} \&\{l_j : R_j\}_{j\in J}) : \&\{l_j : R_j\}_{j\in J}$$

We take some $j \in J$. By $U = \bigvee\{U_j\}_{j\in J}$ and Lemma 16, $U_j \lesssim U$. By $\Delta, x_j : R_j \vdash f_j : U_j$ and the cast rule,

$$\Delta, x_j : R_j \vdash (f_j : U_j \overset{p}{\Rightarrow} U) : U$$

Thus, by the case rule, $\Gamma \vdash f : T$. □

We say that a type, a type environment, or an expression is *static* in the following sense:

- A type T is *static* if T does not contain any dynamic types: i.e., $\star$ or $\circledast$.
- A type environment Γ is *static* if Γ contains only static types.
- An expression $\mathbb{e}$ of GGV_e is *static* if all types declared in $\mathbb{e}$ are static.

Lemma 18.

1. *Suppose T, U are static. If $T \lesssim U$, then $T <: U$.*
2. *Suppose $T \triangleright U$ and $T \neq \star, \circledast$.*
   a. *If U is neither &-type nor $\oplus$-type, then $T = U$.*
   b. *If U is either &-type or $\oplus$-type, then $T <: U$.*
   c. *If T is static and U is not &-type, then U is static.*
3. *Suppose T_1, T_2 are static. If $T_1 \vee T_2 = U$, then*
   a. *U is static,*
   b. *$T_1 <: U$ and $T_2 <: U$,*
   c. *$U <: U'$ for any static U' such that $T_1 <: U'$ and $T_2 <: U'$.*
4. *Suppose T_1, T_2, U' are static. If $T_1 <: U'$ and $T_2 <: U'$, then there exists some static U such that $U = T_1 \vee T_2$.*

Proof. The first item is by induction on $T \lesssim U$. The second item is by case analysis on $T \triangleright U$. Here, we can prove

$$\text{if } T, U \text{ are static and } T <:^{-} U, \text{ then } T <: U$$

by induction on $T <:^{-} U$. By Lemma 13, $<: \subseteq <:^{-}$. Thus, we have

$$\text{if } T, U \text{ are static, then } T <: U \text{ if and only if } T <:^{-} U.$$

With this fact, the third and fourth items can be proved by Lemmas 14 and 15, respectively. □

We define the type erasure $|\mathbb{e}|$, which is obtained by removing type annotations from an expression $\mathbb{e}$ of GGV_e. The main cases of its definition are as follows:

$$|\mathsf{new}\ S| = \mathsf{new}$$
$$|\lambda_m x{:}T.\ \mathbb{e}| = \lambda_m x.\ |\mathbb{e}|$$

(It is extended homomorphically for all other forms of expressions.)

https://doi.org/10.1017/S0956796819000169 Published online by Cambridge University Press

Theorem 9 states that the GGV_e typing is a conservative extension of the GV typing. We have to take care of the difference between the declarative type system of GV and the algorithmic type system of GGV_e.

Theorem 9 (Typing Conservation over GV)**.** *Suppose that* Γ *is static.*

1. *If* $\mathbb{e}$ *is static and type environments that appear in the derivation of* $\Gamma \vdash_e \mathbb{e} : T$ *are all static, then* T *is static and* $\Gamma \vdash_{GV} |\mathbb{e}| : T$.
2. *If* f *is an expression of GV and* $\Gamma \vdash_{GV} f : T$*, then* T *is static and there exist static* $\mathbb{e}$ *and static* T' *such that* $|\mathbb{e}| = f$ *and* $\Gamma \vdash_e \mathbb{e} : T'$ *and* $T' <: T$.

Proof. The first item is by induction on $\Gamma \vdash_e \mathbb{e} : T$ with case analysis on the rule applied last. We show the main cases as follows.

Case application rule: We are given

$$\Gamma = \Gamma_1 \circ \Gamma_2 \qquad \mathbb{e} = \mathbb{e}_1\ \mathbb{e}_2 \qquad T = T_{12}$$
$$\Gamma_1 \vdash \mathbb{e}_1 : T_1 \qquad \Gamma_2 \vdash \mathbb{e}_2 : T_2 \qquad T_1 \triangleright T_{11} \to_m T_{12} \qquad T_2 \lesssim T_{11}$$

Since Γ, $\mathbb{e}$ are static, $\Gamma_1, \Gamma_2, \mathbb{e}_1, \mathbb{e}_2$ are also static. By $\Gamma_1 \vdash \mathbb{e}_1 : T_1$ and the IH, T_1 is static and $\Gamma_1 \vdash |\mathbb{e}_1| : T_1$. By $T_1 \triangleright T_{11} \to_m T_{12}$ and Lemma 18 (2), T_{11}, T_{12} are static and $T_1 = T_{11} \to_m T_{12}$. By $\Gamma_2 \vdash \mathbb{e}_2 : T_2$ and the IH, T_2 is static and $\Gamma_2 \vdash |\mathbb{e}_2| : T_2$. By $T_2 \lesssim T_{11}$ and Lemma 18 (1), $T_2 <: T_{11}$. By the subsumption rule, $\Gamma_2 \vdash |\mathbb{e}_2| : T_{11}$. Thus, by

$$\Gamma_1 \vdash |\mathbb{e}_1| : T_{11} \to_m T_{12} \qquad \Gamma_2 \vdash |\mathbb{e}_2| : T_{11} \qquad |\mathbb{e}_1\ \mathbb{e}_2| = |\mathbb{e}_1|\ |\mathbb{e}_2|$$

and the application rule, we have $\Gamma_1 \circ \Gamma_2 \vdash |\mathbb{e}_1\ \mathbb{e}_2| : T_{12}$.

Case case rule: We are given

$$\Gamma = \Gamma' \circ \Delta \qquad \mathbb{e} = \mathsf{case}\ \mathbb{e}'\ \mathsf{of}\ \{l_j : x_j.\ \mathbb{e}_j\}_{j \in J} \qquad T = U$$
$$\Gamma' \vdash \mathbb{e}' : T' \qquad T' \triangleright \&\{l_j : R_j\}_{j \in J} \qquad (\Delta, x_j : R_j \vdash \mathbb{e}_j : U_j)_{j \in J} \qquad U = \bigvee \{U_j\}_{j \in J}$$

Since Γ, $\mathbb{e}$ are static, Γ', Δ, $\mathbb{e}'$, $\mathbb{e}_j$ are also static. Since any type environment $\Delta, x_j : R_j$ is static, any R_j is static. By $\Gamma' \vdash \mathbb{e}' : T'$ and the IH, T' is static and $\Gamma' \vdash |\mathbb{e}'| : T'$. By $T' \triangleright \&\{l_j : R_j\}_{j \in J}$ and Lemma 18 (2), $T' <: \&\{l_j : R_j\}_{j \in J}$. By the subsumption rule,

$$\Gamma' \vdash |\mathbb{e}'| : \&\{l_j : R_j\}_{j \in J}$$

We take some $j \in J$. Since $\Delta, x_j : R_j$ is static, by $\Delta, x_j : R_j \vdash \mathbb{e}_j : U_j$ and the IH, U_j is static and $\Delta, x_j : R_j \vdash |\mathbb{e}_j| : U_j$. By $U = \bigvee \{U_j\}_{j \in J}$ and Lemma 18 (3), $U_j <: U$ and U is static. By the subsumption rule,

$$\Delta, x_j : R_j \vdash |\mathbb{e}_j| : U$$

Thus, by

$$\Gamma' \vdash |\mathbb{e}'| : \&\{l_j : R_j\}_{j \in J} \qquad (\Delta, x_j : R_j \vdash |\mathbb{e}_j| : U)_{j \in J}$$
$$|\mathsf{case}\ \mathbb{e}'\ \mathsf{of}\ \{l_j : x_j.\ \mathbb{e}_j\}_{j \in J}| = \mathsf{case}\ |\mathbb{e}'|\ \mathsf{of}\ \{l_j : x_j.\ |\mathbb{e}_j|\}_{j \in J}$$

the case rule, we have $\Gamma' \circ \Delta \vdash |\mathsf{case}\ \mathbb{e}'\ \mathsf{of}\ \{l_j : x_j.\ \mathbb{e}_j\}_{j \in J}| : U$.

The second item is by induction on $\Gamma \vdash_{\mathrm{GV}} f : T$ with case analysis on the rule applied last. We show the main cases as follows.

https://doi.org/10.1017/S0956796819000169 Published online by Cambridge University Press

Case application rule: We are given

$$\Gamma = \Gamma_1 \circ \Gamma_2 \qquad f = f_1\ f_2 \qquad T = T_{12} \qquad \Gamma_1 \vdash f_1 : T_{11} \to_m T_{12} \qquad \Gamma_2 \vdash f_2 : T_{11}$$

Since Γ is static, Γ_1, Γ_2 are also static. By $\Gamma_1 \vdash f_1 : T_{11} \to_m T_{12}$ and the IH, T_{11}, T_{12} are static and there exist static $\mathbb{e}_1$, U_1 such that

$$\Gamma_1 \vdash \mathbb{e}_1 : U_1 \qquad U_1 <: T_{11} \to_m T_{12} \qquad |\mathbb{e}_1| = f_1$$

By inversion of $<:$, we have $U_1 = U_{11} \to_n U_{12}$ and $n <: m$ and $T_{11} <: U_{11}$ and $U_{12} <: T_{12}$ for some U_{11}, U_{12}, n. Since U_1 is static, U_{11}, U_{12} are also static. By $\Gamma_2 \vdash f_2 : T_{11}$ and the IH, T_{11} is static and there exist static $\mathbb{e}_2$, U_2 such that

$$\Gamma_2 \vdash \mathbb{e}_2 : U_2 \qquad U_2 <: T_{11} \qquad |\mathbb{e}_2| = f_2$$

By $U_2 <: T_{11}$ and $T_{11} <: U_{11}$ and transitivity, $U_2 <: U_{11}$. By Lemma 13, $U_2 \lesssim U_{11}$. From Figure 9, we have $U_{11} \to_n U_{12} \triangleright U_{11} \to_n U_{12}$. Thus, by

$$\Gamma_1 \vdash \mathbb{e}_1 : U_{11} \to_n U_{12} \qquad \Gamma_2 \vdash \mathbb{e}_2 : U_2 \qquad U_{11} \to_n U_{12} \triangleright U_{11} \to_n U_{12} \qquad U_2 \lesssim U_{11}$$

and the application rule, $\Gamma_1 \circ \Gamma_2 \vdash \mathbb{e}_1\ \mathbb{e}_2 : U_{12}$. Additionally,

$$|\mathbb{e}_1\ \mathbb{e}_2| = |\mathbb{e}_1|\ |\mathbb{e}_2| = f_1\ f_2 = f \qquad U_{12} <: T_{12} = T$$

Case case rule: We are given

$$\Gamma = \Gamma' \circ \Delta \qquad f = \mathsf{case}\ f'\ \mathsf{of}\ \{l_j : x_j.\ f_j\}_{j \in J}$$
$$\Gamma' \vdash f' : \&\{l_j : R_j\}_{j \in J} \qquad (\Delta, x_j : R_j \vdash f_j : T)_{j \in J}$$

Since Γ is static, Γ', Δ are also static. By $\Gamma' \vdash f' : \&\{l_j : R_j\}_{j \in J}$ and the IH, all R_j are static and there exist static $\mathbb{e}'$, T' such that

$$|\mathbb{e}'| = f' \qquad \Gamma' \vdash \mathbb{e}' : T' \qquad T' <: \&\{l_j : R_j\}_{j \in J}$$

By $T' <: \&\{l_j : R_j\}_{j \in J}$ and Lemma 13, $T' \lesssim \&\{l_j : R_j\}_{j \in J}$. We take some $j \in J$. By $\Delta, x_j : R_j \vdash f_j : T$ and the IH, T is static and there exist static $\mathbb{e}_j$, T_j such that

$$|\mathbb{e}_j| = f_j \qquad \Delta, x_j : R_j \vdash \mathbb{e}_j : T_j \qquad T_j <: T$$

So, $(T_j <: T)_{j \in J}$. By Lemma 18 (4), there exist some static U such that $U = \bigvee\{T_j\}_{j \in J}$. By Lemma 18 (3), $U <: T$. Thus, by

$$\Gamma' \vdash \mathbb{e}' : T' \qquad (\Delta, x_j : R_j \vdash \mathbb{e}_j : T_j)_{j \in J} \qquad T' \lesssim \&\{l_j : R_j\}_{j \in J} \qquad U = \bigvee\{T_j\}_{j \in J}$$

and the case rule, $\Gamma' \circ \Delta \vdash \mathsf{case}\ \mathbb{e}'\ \mathsf{of}\ \{l_j : x_j.\ \mathbb{e}_j\}_{j \in J} : U$. Additionally,

$$|\mathsf{case}\ \mathbb{e}'\ \mathsf{of}\ \{l_j : x_j.\ \mathbb{e}_j\}_{j \in J}| = \mathsf{case}\ |\mathbb{e}'|\ \mathsf{of}\ \{l_j : x_j.\ |\mathbb{e}_j|\}_{j \in J} = \mathsf{case}\ f'\ \mathsf{of}\ \{l_j : x_j.\ f_j\}_{j \in J} = f$$

We already have $U <: T$.

Case subsumption rule: We are given $\Gamma \vdash f : U$ and $U <: T$. By the IH, U is static and there exist static $\mathbb{e}$ and U' such that

$$\Gamma \vdash \mathbb{e} : U' \qquad U' <: U \qquad |\mathbb{e}| = f$$

By $U' <: U$ and $U <: T$ and transitivity, $U' <: T$. □

https://doi.org/10.1017/S0956796819000169 Published online by Cambridge University Press

Proposition 2 states that the cast-insertion translation does not insert casts for static expressions, which can be seen as expressions of GV if type annotations are removed. The proof is similar to that of Theorem 9 (1).

Proposition 2. *Suppose that* Γ *and* $\mathbb{e}$ *are both static. If* $\Gamma \vdash \mathbb{e} \rightsquigarrow f : T$*, then* T *is static and* $|\mathbb{e}| = f$.

Proof. By induction on $\Gamma \vdash \mathbb{e} \rightsquigarrow f : T$. with case analysis on the rule applied last. We show one of the main cases as follows.

Case application rule: We are given

$$\Gamma = \Gamma_1 \circ \Gamma_2 \qquad \mathbb{e} = \mathbb{e}_1\ \mathbb{e}_2 \qquad f = (f_1 : T_1 \overset{p}{\Rightarrow}_? T_{11} \to_m T_{12})\,(f_2 : T_2 \overset{p}{\Rightarrow}_? T_{11}) \qquad T = T_{12}$$
$$\Gamma_1 \vdash \mathbb{e}_1 \rightsquigarrow f_1 : T_1 \qquad \Gamma_2 \vdash \mathbb{e}_2 \rightsquigarrow f_2 : T_2 \qquad T_1 \rhd T_{11} \to_m T_{12} \qquad T_2 \lesssim T_{11}$$

Since Γ, $\mathbb{e}$ are static, Γ_1, Γ_2, $\mathbb{e}_1$, $\mathbb{e}_2$ are also static. By $\Gamma_1 \vdash \mathbb{e}_1 \rightsquigarrow f_1 : T_1$ and the IH, T_1 is static and $|\mathbb{e}_1| = f_1$. By $T_1 \rhd T_{11} \to_m T_{12}$ and Lemma 18 (2), T_{11}, T_{12} are static and $T_1 = T_{11} \to_m T_{12}$. So,

$$f_1 : T_1 \overset{p}{\Rightarrow}_? T_{11} \to_m T_{12} = f_1 = |\mathbb{e}_1|$$

By $\Gamma_2 \vdash \mathbb{e}_2 \rightsquigarrow f_2 : T_2$ and the IH, T_2 is static and $|\mathbb{e}_2| = f_2$. By $T_2 \lesssim T_{11}$ and Lemma 18 (1), $T_2 <: T_{11}$. So,

$$f_2 : T_2 \overset{p}{\Rightarrow}_? T_{11} = f_2 = |\mathbb{e}_2|$$

Thus, $f = |\mathbb{e}_1|\ |\mathbb{e}_2| = |\mathbb{e}_1\ \mathbb{e}_2|$. □

4.4 (Failure of) The gradual guarantee

In a gradually typed language, changing type annotations in a program should not change the static or dynamic behavior—except for runtime errors caused by casts. Such an expectation is formalized by Siek *et al.* (2015b) as the gradual guarantee property. It usually consists of two statements concerning the static and dynamic aspects of programs. The static counterpart of the gradual guarantee (simply called the static gradual guarantee) states that less precise type annotations make the type of an expression less precise, whereas the dynamic gradual guarantee states that making type annotations less precise does not change the final outcome of a program.

We will show that, unfortunately, GGV_e satisfies neither the static nor dynamic gradual guarantee by constructing counterexamples. We analyze the problem and argue that it is not easy to recover without losing other good properties.

First, to capture the notion of programs with more precise type annotations formally, the precision over types is extended to type environments and expressions. The relation $\Gamma_1 \sqsubseteq \Gamma_2$ is the least relation that satisfies $\cdot \sqsubseteq \cdot$ and $\Gamma_1, x : T_1 \sqsubseteq \Gamma_2, x : T_2$ if $\Gamma_1 \sqsubseteq \Gamma_2$ and $T_1 \sqsubseteq T_2$ and the relation $\mathbb{e}_1 \sqsubseteq \mathbb{e}_2$ is the least precongruence that is closed under the following rules:

$$\frac{T_1 \sqsubseteq T_2 \qquad \mathbb{e}_1 \sqsubseteq \mathbb{e}_2}{\lambda_m x{:}T_1.\,\mathbb{e}_1 \sqsubseteq \lambda_m x{:}T_2.\,\mathbb{e}_2} \qquad \frac{S_1 \sqsubseteq S_2}{\mathsf{new}\ S_1 \sqsubseteq \mathsf{new}\ S_2} \tag{1}$$

https://doi.org/10.1017/S0956796819000169 Published online by Cambridge University Press

Using the precision, the static gradual guarantee can be stated as follows.

If $\Gamma_1 \sqsubseteq \Gamma_2$, $\mathbb{e}_1 \sqsubseteq \mathbb{e}_2$, and $\Gamma_1 \vdash \mathbb{e}_1 : T_1$, then $\Gamma_2 \vdash \mathbb{e}_2 : T_2$ and $T_1 \sqsubseteq T_2$ for some T_2.

However, it does not hold.

Theorem 10 (Failure of the Static Gradual Guarantee)**.** *There exist Γ_1, Γ_2, $\mathbb{e}_1$, $\mathbb{e}_2$, and T_1 such that $\Gamma_1 \sqsubseteq \Gamma_2$, $\mathbb{e}_1 \sqsubseteq \mathbb{e}_2$, $\Gamma_1 \vdash_e \mathbb{e}_1 : T_1$ and, for any T_2 such that $\Gamma_2 \vdash_e \mathbb{e}_2 : T_2$, $T_1 \not\sqsubseteq T_2$.*

Proof. Let

$$\begin{aligned}
\Gamma_1 &= x : T_1, y : T_2, z : \&\{l_1 : \mathsf{end}_!, l_2 : \mathsf{end}_!\} \\
\Gamma_2 &= x : \star, y : T_2, z : \&\{l_1 : \mathsf{end}_!, l_2 : \mathsf{end}_!\} \\
\mathbb{e}_1 &= \mathbb{e}_2 = \mathsf{case}\, z\, \mathsf{of}\, \{l_1 : x_1.\, \mathsf{close}\, x_1; x,\ l_2 : x_2.\, \mathsf{close}\, x_2; y\} \\
T_1 &= \mathsf{unit} \to_{\mathsf{lin}} \mathsf{unit} \\
T_2 &= \mathsf{unit} \to_{\mathsf{un}} \mathsf{unit}
\end{aligned}$$

(where $\mathbb{e}_1; \mathbb{e}_2$ stands for usual sequential composition). Then, $\Gamma_1 \sqsubseteq \Gamma_2$, $\mathbb{e}_1 \sqsubseteq \mathbb{e}_2$, $\Gamma_1 \vdash_e \mathbb{e}_1 : T_1$, and $\Gamma_2 \vdash_e \mathbb{e}_2 : T_2$; but $T_1 \not\sqsubseteq T_2$. □

Although we do not state the dynamic gradual guarantee formally, we expect at least that, if two programs $\mathbb{e}_1$ and $\mathbb{e}_2$ satisfy $\mathbb{e}_1 \sqsubseteq \mathbb{e}_2$ and the execution of $\mathbb{e}_1$ (after cast insertion) terminates normally (at $\langle () \rangle$), then $\mathbb{e}_2$ also terminates normally. Unfortunately, it would not be very difficult to see such an expectation fail. Let's consider

$$\mathbb{e}_1' = (\lambda_{\mathsf{lin}} x{:}T_1.\, \mathbb{e}_1)\, (\lambda_{\mathsf{lin}} x_1{:}\mathsf{unit}.\, x_1)$$

and a more imprecise expression

$$\mathbb{e}_2' = (\lambda_{\mathsf{lin}} x{:}\star.\, \mathbb{e}_2)\, (\lambda_{\mathsf{lin}} x_1{:}\mathsf{unit}.\, x_1)$$

The former will return $\lambda_{\mathsf{lin}} x_1{:}\mathsf{unit}.\, x_1$ if l_1 is selected by another process. However, $\mathbb{e}_2'$ shows different behavior: the cast-inserting translation puts a cast from $\star$ to $T_2 = \mathsf{unit} \to_{\mathsf{un}} \mathsf{unit}$ on x in the first branch of case in $\mathbb{e}_2$ but x will be bound to (a reference to) a linear function and, if l_1 is selected, the cast will fail and raise blame.

The problem seems to stem from the fact that $\vee$ has subtle interaction with $\sqsubseteq$. For typing case-expressions, we would naturally require precision to be preserved by the join operation, i.e., if $T_1 \sqsubseteq T_1'$, then $T_1 \vee T_2 \sqsubseteq T_1' \vee T_2$. However, the current definition of $\vee$ breaks this property as the counterexample to the static gradual guarantee above shows. Also, as we can see from the counterexample to the dynamic gradual guarantee, join with a more precise type can yield a supertype—that is, $T_1 \sqsubseteq \star$ and $\star \vee T_2 <: T_1 \vee T_2$ hold.

One possible workaround is to adapt the "lifted join" operation $\tilde{\ddot{\vee}}$ of the GTFL$_{\lesssim}$ language (Garcia *et al.*, 2016) to Gradual GV. Like $\vee$, $\mathsf{unit} \to_{\mathsf{lin}} \mathsf{unit} \tilde{\ddot{\vee}} \mathsf{unit} \to_{\mathsf{un}} \mathsf{unit} = \mathsf{unit} \to_{\mathsf{lin}} \mathsf{unit}$ but, unlike $\vee$, $\star \tilde{\ddot{\vee}} \mathsf{unit} \to_{\mathsf{un}} \mathsf{unit} = \star$. Thus, the lifted join would perhaps recover the gradual guarantee. However, it seems that the lifted join is the least upper bound operation for no known ordering between types and we would lose the minimal type property of GGV$_e$ if we used $\tilde{\ddot{\vee}}$. Also, the lifted join has the following property: $T \tilde{\ddot{\vee}} \star$ is T only if T does not have nontrivial supertypes; otherwise $T \tilde{\ddot{\vee}} \star$ is $\star$. For example, $\mathsf{unit} \tilde{\ddot{\vee}} \star = \mathsf{unit}$ and $\mathsf{unit} \to_{\mathsf{lin}} \mathsf{unit} \tilde{\ddot{\vee}} \star = \mathsf{unit} \to_{\mathsf{lin}} \mathsf{unit}$ but $\mathsf{unit} \to_{\mathsf{un}} \mathsf{unit} \tilde{\ddot{\vee}} \star = \star$

https://doi.org/10.1017/S0956796819000169 Published online by Cambridge University Press

(because $\mathsf{unit} \to_{\mathsf{un}} \mathsf{unit} <: \mathsf{unit} \to_{\mathsf{lin}} \mathsf{unit}$). It means that the standard narrowing property—if $\Gamma, x : T_1 \vdash_e \mathbb{e} : T$ and $T_2 <: T_1$, then $\Gamma_1, x : T_2 \vdash_e \mathbb{e} : T$—does not hold. We leave more detailed analysis of the problem and possible remedy for future work.

5 Related work

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, 2007) 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 contracts, 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."

While the first investigations of gradual typing were based on simply typed calculi, subsequent work has explored gradual typing for a range of typing features. Polymorphism (Ahmed *et al.*, 2011; Igarashi *et al.*, 2017b; Toro *et al.*, 2019) has proved to be quite tricky, with one important question about the Jack-of-All-Trades Principle (Ahmed *et al.*, 2011) still open. A gradual treatment of record types may be found in the paper on Abstract Gradual Typing (AGT) (Garcia *et al.*, 2016). Variant types have proved elusive, but union types have been considered (Siek & Tobin-Hochstadt, 2016) along with intersection types and polymorphism as part of a set-theoretical reevaluation of gradual principles (Castagna *et al.*, 2019).

Moving toward session types, systems with gradual typestate have been considered (Wolff *et al.*, 2011; Garcia *et al.*, 2014); they extend an object-oriented language with typestate by a dynamic type and define a suitable translation to an internal language with casts. The additional complication is to track the current typestate at runtime. Thiemann (2014) describes a system with gradual types and session types, but in it only types (and not session types) can be gradual.

Effect systems have been gradualized based on ideas from abstract interpretation by Banados Schwerter and others (2014). While the former work only presented a gradualization of effects themselves a subsequent extension adds a full treatment of types (Schwerter *et al.*, 2016). Related ideas are explored by Thiemann and Fennell (2014), who present an approach to gradualize annotated type systems, like units and security labels. Following an earlier approach for gradual security typing for simply typed lambda calculus (Disney & Flanagan, 2011), Fennell and Thiemann (2013) developed gradual security for an ML core language with references and subsequently for a Java core language LJGS with polymorphic security labels (Fennell & Thiemann, 2016). Toro and others (2018) developed a gradual calculus with slightly different features from first principles using the AGT (Garcia *et al.*, 2016) approach. In each of these approaches, special measures have to be taken to

https://doi.org/10.1017/S0956796819000169 Published online by Cambridge University Press

ensure the key property of noninterference. Gradual type systems related to session types also include the runtime enforcement of affine typing of Tov and Pucella Tov and Pucella (2010).

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. There is a range of gradual type systems for dependent types. Ou and others (2004) bridge the gap between simply typed lambda calculus and a calculus with indexed types. In Flanagan's hybrid typing (2006), subtyping judgments are either proved or disproved statically by SMT theorem proving or residualized as runtime checks. Greenberg and others (2010) consider different styles of contracts in simply typed and dependently typed settings. Lehmann and Tanter (2017) present an approach that uses the AGT methodology to obtain a gradual system that mediates between simple types and dependent refinement types. This work has been augmented with type inference by Vazou and others (2018) and it has been extended to verification (Bader *et al.*, 2018) where specifications may contains unknown subformulas. Jafery & Dunfield (2017) consider gradualized refinements for sum types with the goal to control errors in pattern matching.

Gradual ownership types (Sergey & Clarke, 2012) is a gradualization of the Owners as Dominators principle of ownership. Its theory is built with similar principles as other gradual languages, but its flavor is different as ownership is not a semantic property, but a structure imposed by the programmer.

Siek and others (2015b) review desirable properties of gradually typed languages, while Wadler Wadler (2015) discusses history of the blame calculus and why blame is important. These papers provide overviews of the field, each with many further citations. Many of the above-cited works strive to fulfill the properties of Siek and others, not all of them are successful, but further discussion of the properties exceeds the scope of this survey of related work.

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) and 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 and others (2010) propose attaching session types to class definitions, allowing to treat channels as objects for session-based communication in distributed systems. SePi (Franco & Vasconcelos, 2013) is a concurrent, message-passing programming language based on the pi-calculus, featuring a simple form of refinement types. SILL (Toninho *et al.*, 2013; Pfenning & Griffith, 2015) is a higher-order session

https://doi.org/10.1017/S0956796819000169 Published online by Cambridge University Press

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 & 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 & 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 (Pucella & Tov, 2008; Sackman & Eisenbach, 2008; Polakow, 2015; Orchard & Yoshida, 2016; Lindley & Morris, 2016a), OCaml (Padovani, 2017), Scala (Scalas & Yoshida, 2016), and Rust (Jespersen *et al.*, 2015). Most of these embeddings delegate linearity checks to the runtime system.

Session types can be used in the software development process under different forms, including languages to describe protocols, specialized libraries to invoke session-based communication primitives, provision for runtime 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 runtime monitors for endpoint communication from Scribble protocols. Neykova and Yoshida (2014) designed and implemented a session actor library in Python together with a runtime verification mechanism. Bocchi and others (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.

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 expected results for such a language, including type safety and blame safety. The gradual guarantee however does not hold, and it is not clear how it can be recovered without losing other good properties.

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

https://doi.org/10.1017/S0956796819000169 Published online by Cambridge University Press

to be chained as in (close (send v_2 (send v_1 c))). However, one could imagine a uni-typed language where one may use the channel nonlinearly in an imperative style as in (send v_1 c; send v_2 c; 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, Kaede Kobayashi for pointing out subtle errors in the operational semantics, and anonymous reviewers for constructive comments. We are also 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/00408/2019 and project Confident ref. PTDC/EEI-CTP/4503/2014 (Vasconcelos), and by EPSRC program grant EP/K034413/1 (Wadler).

References

Ahmed, A., Findler, R. B., Siek, J. G. & Wadler, P. (2011) Blame for all. In Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26–28, 2011, Ball, T. & Sagiv, M. (eds), ACM, pp. 201–214.

Bader, J., Aldrich, J. & Tanter, É. (2018) Gradual program verification. In *Verification, Model Checking, and Abstract Interpretation* – 19th International Conference, VMCAI 2018, Los Angeles, CA, USA, January 7–9, 2018, Proceedings, Dillig, I. & Palsberg, J. (eds). Lecture Notes in Computer Science, vol. 10747. Springer, pp. 25–46.

Bañados Schwerter, F., Garcia, R. & Tanter, É. (2014) A theory of gradual effect systems. In International Conference on Functional Programming (ICFP). ACM, pp. 283–295.

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

Bierman, G., Abadi, M. & Torgersen, M. (2014) Understanding TypeScript. In European Conference on Object-Oriented Programming (ECOOP). LNCS, vol. 8586. Springer, pp. 257–281.

Bierman, G. M., Meijer, E. & Torgersen, M. (2010) Adding dynamic types to C#. In European Conference on Object-Oriented Programming (ECOOP). LNCS. Springer, pp. 76–100.

Bocchi, L., Chen, T.-C., Demangeon, R., Honda, K. & Yoshida, N. (2013) Monitoring networks through multiparty session types. In *Formal Techniques for Distributed Systems*, Beyer, D. & Boreale, M. (eds). Springer, pp. 50–65.

Bocchi, L., Chen, T.-C., Demangeon, R., Honda, K. & Yoshida, N. (2017) Monitoring networks through multiparty session types. *Theor. Comput. Sci.* **699**, 33–58.

Brady, E. (2013) Idris, a general-purpose dependently typed programming language: Design and implementation. *J. Funct. Program.* **23**(05), 552–593.

Caires, L. & Pfenning, F. (2010) Session types as intuitionistic linear propositions. In International Conference on Concurrency Theory (CONCUR). LNCS. Springer, pp. 222–236.

Caires, L., Pfenning, F. & Toninho, B. (2014) Linear logic propositions as session types. *Math. Struct. Comput. Sci.* **26**(03), 367–423.

Carbone, M., Honda, K. & Yoshida, N. (2007) Structured communication-centred programming for web services. In De Nicola, R. (ed), Programming Languages and Systems, 16th European Symposium on Programming, ESOP 2007, Held as Part of the Joint European Conferences

https://doi.org/10.1017/S0956796819000169 Published online by Cambridge University Press

on Theory and Practics of Software, ETAPS 2007, Braga, Portugal, March 24–April 1, 2007, Proceedings. Lecture Notes in Computer Science, vol. 4421. Springer, pp. 2–17.

Carbone, M., Honda, K. & Yoshida, N. (2012) Structured communication-centered programming for web services. *ACM Trans. Program. Lang. Syst.* **34**(2), 8.

Castagna, G., Lanvin, V., Petrucciani, T. & Siek, J. G. (2019) Gradual typing: A new perspective. *PACMPL* **3**(POPL), 16:1–16:32.

Chaudhuri, A., Vekris, P., Goldman, S., Roch, M. & Levi, G. (2017) Fast and precise type checking for JavaScript. *PACMPL* **1**(OOPSLA), 48:1–48:30.

Cimini, M. & Siek, J. G. (2016) The Gradualizer: A methodology and algorithm for generating gradual type systems. In Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20–22, 2016, pp. 443–455.

Cooper, E., Lindley, S., Wadler, P. & Yallop, J. (2007) Links: Web programming without tiers. In *Formal Methods for Components and Objects*, de Boer, F. S., Bonsangue, M. M., Graf, S. & de Roever, W. P. (eds). Springer, pp. 266–296.

Demangeon, R. & Honda, K. (2011) Full abstraction in a subtyped pi-calculus with linear types. In Katoen, J.-P. & König, B. (eds), CONCUR 2011 - Concurrency Theory - 22nd International Conference, CONCUR 2011, Aachen, Germany, September 6–9, 2011. Proceedings. Lecture Notes in Computer Science, vol. 6901. Springer, pp. 280–296.

Demangeon, R., Honda, K., Hu, R., Neykova, R. & Yoshida, N. (2015) Practical interruptible conversations: Distributed dynamic verification with multiparty session types and python. *Formal Methods Syst. Des.* **46**(3), 197–225.

Disney, T. & Flanagan, C. (2011) Gradual information flow typing. In *STOP*. Available at: https://users.soe.ucsc.edu/~cormac/papers/stop11.pdf

Ernst, E., Møller, A., Schwarz, M. & Strocco, F. (2017) Message safety in Dart. *Sci. Comput. Program.* **133**, 51–73.

Fähndrich, M., Aiken, M., Hawblitzel, C., Hodson, O., Hunt, G. C., Larus, J. R. & Levi, S. (2006) Language support for fast and reliable message-based communication in Singularity OS. In European Conference on Computer Systems (EuroSys). ACM, pp. 177–190.

Fennell, L. & Thiemann, P. (2012). The blame theorem for a linear lambda calculus with type dynamic. In *Trends in Functional Programming*, Loidl, H-W. & Peña, R. (eds). LNCS, vol. 7829. Springer, pp. 37–52.

Fennell, L. & Thiemann, P. (2013) Gradual security typing with references. In 2013 IEEE 26th Computer Security Foundations Symposium, New Orleans, LA, USA, June 26–28, 2013. IEEE Computer Society, pp. 224–239.

Fennell, L. & Thiemann, P. (2016) LJGS: Gradual security types for object-oriented languages. In Krishnamurthi, S. & Lerner, B. S. (eds), 30th European Conference on Object-Oriented Programming, ECOOP 2016, July 18–22, 2016, Rome, Italy. LIPIcs, vol. 56. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, pp. 9:1–9:26.

Findler, R. B. & Felleisen, M. (2002) Contracts for higher-order functions. In International Conference on Functional Programming (ICFP). ACM, pp. 48–59.

Flanagan, C. (2006) Hybrid type checking. In *Principles of Programming Languages (POPL)*, Morrisett, J. G. & Peyton Jones, S. L. (eds). ACM, pp. 245–256.

Fowler, S. (2016) An Erlang implementation of multiparty session actors. In Interaction and Concurrency Experience, pp. 36–50.

Franco, J. & Vasconcelos, V. T. (2013) A concurrent programming language with refined session types. In *SEFM*. LNCS, vol. 8368. Springer, pp. 15–28.

Garcia, R., Clark, A. M. & Tanter, É. (2016) Abstracting gradual typing. In Principles of Programming Languages (POPL). ACM, pp. 429–442.

Garcia, R., Tanter, É., Wolff, R. & Aldrich, J. (2014) Foundations of typestate-oriented programming. *ACM Trans. Program. Lang. Syst.* **36**(4), 12:1–12:44.

Gay, S. & Hole, M. (2005) Subtyping for session types in the pi calculus. *Acta Informatica* **42**(2–3), 191–225.

https://doi.org/10.1017/S0956796819000169 Published online by Cambridge University Press

Gay, S. & Vasconcelos, V. (2010) Linear type theory for asynchronous session types. *J. Funct. Program.* **20**(01), 19–50.

Gay, S. J. (2016) Subtyping supports safe session substitution. In *A List of Successes That Can Change the World - Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*, Lindley, S., McBride, C., Trinder, P. W. & Sannella, D. (eds). Lecture Notes in Computer Science, vol. 9600. Springer, pp. 95–108.

Gay, S. J., Vasconcelos, V. T., Ravara, A., Gesbert, N. & Caldeira, A. Z. (2010) Modular session types for distributed object-oriented programming. In Principles of Programming Languages (POPL). ACM, pp. 299–312.

Girard, J.-Y. (1987) Linear logic. *Theoret. Comput. Sci.* **50**(1), 1–101.

Gommerstadt, H., Jia, L. & Pfenning, F. (2018) Session-typed concurrent contracts. In Programming Languages and Systems - 27th European Symposium on Programming, ESOP 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14–20, 2018, Proceedings, Ahmed, A. (ed). Lecture Notes in Computer Science, vol. 10801. Springer, pp. 771–798.

Greenberg, M., Pierce, B. C. & Weirich, S. (2010) Contracts made manifest. In Principles of Programming Languages (POPL). ACM, pp. 353–364.

Honda, K. (1993) Types for dyadic interaction. In International Conference on Concurrency Theory (CONCUR). LNCS, vol. 715. Springer, pp. 509–523.

Honda, K., Mukhamedov, A., Brown, G., Chen, T. & Yoshida, N. (2011) Scribbling interactions with a formal foundation. In *ICDCIT*. LNCS, vol. 6536. Springer, pp. 55–75.

Honda, K., Vasconcelos, V. & Kubo, M. (1998) Language primitives and type discipline for structured communication-based programming. In European Symposium on Programming (ESOP). LNCS. Springer, pp. 122–138.

Honda, K., Yoshida, N. & Carbone, M. (2008) Multiparty asynchronous session types. In Principles of Programming Languages (POPL). ACM, pp. 273–284.

Honda, K., Yoshida, N. & Carbone, M. (2016) Multiparty asynchronous session types. *J. ACM* **63**(1), 9.

Hu, R., Kouzapas, D., Pernet, O., Yoshida, N. & Honda, K. (2010) Type-safe eventful sessions in Java. In European Conference on Object-Oriented Programming (ECOOP). LNCS, vol. 6183. Springer, pp. 329–353.

Hu, R. & Yoshida, N. (2016) Hybrid session verification through endpoint API generation. In *Fundamental Approaches to Software Engineering (FASE)*. LNCS, vol. 9633. Springer, pp. 401–418.

Hu, R., Yoshida, N. & Honda, K. (2008) Session-based distributed programming in Java. In European Conference on Object-Oriented Programming (ECOOP). LNCS, vol. 5142. Springer, pp. 516–541.

Igarashi, A., Thiemann, P., Vasconcelos, V. T. & Wadler, P. (2017a) Gradual session types. *PACMPL* **1**(ICFP), 38:1–38:28.

Igarashi, Y., Sekiyama, T. & Igarashi, A. (2017b) On polymorphic gradual typing. *PACMPL* **1**(ICFP), 40:1–40:29.

Jafery, K. A. & Dunfield, J. (2017) Sums of uncertainty: Refinements go gradual. In Castagna, G. & Gordon, A. D. (eds), Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18–20, 2017. ACM, pp. 804–817.

Jespersen, T. B. L., Munksgaard, P. & Larsen, K. F. (2015) Session types for Rust. In Workshop on Generic Programming (WGP). ACM, pp. 13–22.

Jia, L., Gommerstadt, H. & Pfenning, F. (2016) Monitors and blame assignment for higher-order session types. In Bodík, R. & Majumdar, R. (eds), Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20–22, 2016. ACM, pp. 582–594.

Kobayashi, N., Pierce, B. C. & Turner, D. N. (1999) Linearity and the pi-calculus. *ACM Trans. Program. Lang. Syst.* **21**(5), 914–947.

https://doi.org/10.1017/S0956796819000169 Published online by Cambridge University Press

Kouzapas, D., Dardha, O., Perera, R. & Gay, S. J. (2016) Typechecking protocols with Mungo and StMungo. In Principles and Practice of Declarative Programming (PPDP). ACM, pp. 146–159.

Lehmann, N. & Tanter, É. (2017) Gradual refinement types. In Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18–20, 2017, Castagna, G. & Gordon, A. D. (eds). ACM, pp. 775–788.

Lindley, S. & Morris, J. G. (2016a) Embedding session types in Haskell. In Symposium on Haskell. ACM, pp. 133–145.

Lindley, S. & Morris, J. G. (2016b) Talking bananas: Structural recursion for session types. In International Conference on Functional Programming (ICFP). ACM, pp. 434–447.

Lindley, S. & Morris, J. G. (2017) Lightweight functional session types. In *Behavioural Types: From Theory to Tools*. River Publishers.

Melgratti, H. C. & Padovani, L. (2017) Chaperone contracts for higher-order sessions. *PACMPL* **1**(ICFP), 35:1–35:29.

Milner, R., Parrow, J. & Walker, D. (1992) A calculus of mobile processes, I. *Inform. Comput.* **100**(1), 1–40.

Mostrous, D. & Vasconcelos, V. T. (2011) Session typing for a featherweight Erlang. In Coordination Models and Languages (COORDINATION). LNCS, vol. 6721. Springer, pp. 95–109.

Neykova, R. & Yoshida, N. (2014) Multiparty session actors. In Coordination Models and Languages (COORDINATION). LNCS, vol. 8459. Springer, pp. 131–146.

Neykova, R. & Yoshida, N. (2017) Let it recover: Multiparty protocol-induced recovery. In International Conference on Compiler Construction (CC). ACM, pp. 98–108.

Neykova, R., Yoshida, N. & Hu, R. (2013) SPY: Local verification of global protocols. In International Conference on Runtime Verification (RV). LNCS, vol. 8174. Springer, pp. 358–363.

Ng, N., Yoshida, N. & Honda, K. (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, pp. 202–218.

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

Orchard, D. & Yoshida, N. (2016) Effects as sessions, sessions as effects. In Principles of Programming Languages (POPL). ACM, pp. 568–581.

Ou, X., Tan, G., Mandelbaum, Y. & Walker, D. (2004) Dynamic typing with dependent types. In IFIP International Conference on Theoretical Computer Science, vol. 155. Springer, pp. 437–450.

Padovani, L. (2017) A simple library implementation of binary sessions. *J. Funct. Program* **27**, e4.

Pfenning, F. & Griffith, D. (2015) Polarized substructural session types. In International Conference on Foundations of Software Science and Computation Structures. LNCS, vol. 9034. Springer, pp. 3–22.

Pierce, B. C. (2002) *Types and Programming Languages*. MIT Press.

Polakow, J. (2015) Embedding a full linear lambda calculus in Haskell. In Symposium on Haskell. ACM, pp. 177–188.

Pucella, R. & Tov, J. A. (2008) Haskell session types with (almost) no class. In Symposium on Haskell. ACM, pp. 25–36.

Sackman, M. & Eisenbach, S. (2008) Session Types in Haskell: Updating Message Passing for the 21st Century. Available at: https://spiral.imperial.ac.uk:8443/handle/10044/1/5918.

Scalas, A. & Yoshida, N. (2016) Lightweight session programming in Scala. In European Conference on Object-Oriented Programming (ECOOP). LIPIcs. Schloss Dagstuhl, pp. 21:1–21:28.

Schwerter, F. B., Garcia, R. & Tanter, É. (2016) Gradual type-and-effect systems. *J. Funct. Program.* **26**, e19.

Sergey, I. & Clarke, D. (2012) Gradual ownership types. In Programming Languages and Systems - 21st European Symposium on Programming, ESOP 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24–April 1, 2012. Proceedings, Seidl, H. (ed). Lecture Notes in Computer Science, vol. 7211. Springer, pp. 579–599.

https://doi.org/10.1017/S0956796819000169 Published online by Cambridge University Press

Siek, J., Thiemann, P. & Wadler, P. (2015a) Blame and coercion: Together again for the first time. In Programming Language Design and Implementation (PLDI), ACM, pp. 425–435.

Siek, J. G., & Taha, W. (2006) Gradual typing for functional languages. In Scheme and Functional Programming Workshop (Scheme), pp. 81–92. Available at: http://scheme2006.cs.uchicago.edu and the paper URL is http://scheme2006.cs.uchicago.edu/13-siek.pdf

Siek, J. G. & Taha, W. (2007) Gradual typing for objects. In *ECOOP*. Lecture Notes in Computer Science, vol. 4609. Springer, pp. 2–27.

Siek, J. G. & Tobin-Hochstadt, S. (2016) The recursive union of some gradual types. In Lindley, S., McBride, C., Trinder, P. W. & Sannella, D. (eds), *A List of Successes That Can Change the World - Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*. Lecture Notes in Computer Science, vol. 9600. Springer, pp. 388–410.

Siek, J. G., Vitousek, M. M., Cimini, M. & Boyland, J. T. (2015b) Refined criteria for gradual typing. In Summit on Advances in Programming Languages (SNAPL). LIPIcs, vol. 32. Schloss Dagstuhl, pp. 274–293.

The Coq Development Team (2019) *The coq proof assistant, version 8.9.0*.

The Dart Team (2014) *Dart programming language specification*. Google, 1.2 edition.

Thiemann, P. (2014) Session types with gradual typing. In *TGC*. LNCS, vol. 8902. Springer, pp. 144–158.

Thiemann, P. & Fennell, L. (2014) Gradual typing for annotated type systems. In Programming Languages and Systems - 23rd European Symposium on Programming, ESOP 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5–13, 2014, Proceedings, Shao, Z. (ed). Lecture Notes in Computer Science, vol. 8410. Springer, pp. 47–66.

Tobin-Hochstadt, S. & Felleisen, M. (2008) The design and implementation of typed Scheme. In Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7–12, 2008, Necula, G. C. & Wadler, P. (eds). ACM, pp. 395–406.

Toninho, B., Caires, L. & Pfenning, F. (2013) Higher-order processes, functions, and sessions: A monadic integration. In European Symposium on Programming (ESOP). LNCS, vol. 7792. Springer, pp. 350–369.

Toninho, B. & Yoshida, N. (2018) Depending on session-typed processes. In *FoSSaCS*. Lecture Notes in Computer Science, vol. 10803. Springer, pp. 128–145.

Toro, M., Garcia, R. & Tanter, É. (2018) Type-driven gradual security with references. *ACM Trans. Program. Lang. Syst.* **40**(4), 16:1–16:55.

Toro, M., Labrada, E. & Tanter, É. (2019) Gradual parametricity, revisited. *PACMPL* **3**(POPL), 17:1–17:30.

Tov, J. A. & Pucella, R. (2010) Stateful contracts for affine types. In European Symposium on Programming (ESOP). LNCS, vol. 6012. Springer, pp. 550–569.

Vasconcelos, V. T. (2012) Fundamentals of session types. *Inform. Comput.* **217**, 52–70.

Vazou, N., Tanter, É. & Horn, D. V. (2018) Gradual liquid type inference. *PACMPL* **2**(OOPSLA), 132:1–132:25.

Verlaguet, J. (2013) Facebook: Analysing PHP statically. In Workshop on Commercial Uses of Functional Programming (CUFP).

Vitousek, M. M., Swords, C. & Siek, J. G. (2017) Big types in little runtime: Open-world soundness and collaborative blame for gradual type systems. In Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18–20, 2017, Castagna, G. & Gordon, A. D. (eds). ACM, pp. 762–774.

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

Wadler, P. (2014) Propositions as sessions. *J. Funct. Program.* **24**(2–3), 384–418.

Wadler, P. (2015) A complement to blame. In 1st Summit on Advances in Programming Languages (SNAPL). LIPIcs, vol. 32. Schloss Dagstuhl, pp. 309–320.

Wadler, P. & Findler, R. B. (2009) Well-typed programs can't be blamed. In European Symposium on Programming (ESOP). LNCS, vol. 5502. Springer, pp. 1–16.

https://doi.org/10.1017/S0956796819000169 Published online by Cambridge University Press

Walker, D. (2005) Substructural type systems. In *Advanced Topics in Types and Programming Languages*. MIT Press, pp. 3–43.

Williams, J., Morris, J. G., Wadler, P. & Zalewski, J. (2017) Mixed messages: Measuring conformance and non-interference in TypeScript. In European Conference on Object-Oriented Programming (ECOOP). LIPIcs, vol. 74. Dagstuhl, Germany: Schloss Dagstuhl, pp. 28:1–28:29.

Willsey, M., Prabhu, R. & Pfenning, F. (2017) Design and implementation of concurrent C0. In International Workshop on Linearity. EPTCS, vol. 238, pp. 73–82.

Wolff, R., Garcia, R., Tanter, É. & Aldrich, J. (2011) Gradual typestate. In European Conference on Object-Oriented Programming (ECOOP). LNCS, vol. 6813. Springer, pp. 459–483.

Yoshida, N., Hu, R., Neykova, R. & Ng, N. (2014) The Scribble protocol language. In *International Symposium on Trustworthy Global Computing*. LNCS, vol. 8358. Springer, pp. 22–41.

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

Appendix: Type-checking algorithm for the external language

We give a type-checking algorithm for GGV_e and show that it is correct. The type-checking algorithm is slightly involved due to linearity: $\textsc{CheckExpr}(\Gamma, \mathbb{e})$ outputs a pair of type T and a set X of variables, containing the linear variables occurring free in $\mathbb{e}$.

function $\textsc{CheckExpr}(\Gamma, \mathbb{e})$
  case $\mathbb{e}$ **of**
    $\mid z \Rightarrow$
      assert $z \in \mathsf{dom}(\Gamma)$
      $T := \Gamma(z)$
      if $\mathsf{lin}(T)$ **then return** $T, \{z\}$
      else return $T, \emptyset$
    $\mid () \Rightarrow$ **return** $\mathsf{unit}, \emptyset$
    $\mid \lambda_m x{:}T_1.\, \mathbb{e}_1 \Rightarrow$
      $T_2, Y := \textsc{CheckExpr}((\Gamma, x : T_1), \mathbb{e}_1)$
      if $\mathsf{lin}(T_1)$ and $m = \mathsf{un}$ **then**
        assert $Y = \{x\}$
        return $T_1 \to_{\mathsf{un}} T_2, \emptyset$
      else if $\mathsf{lin}(T_1)$ and $m = \mathsf{lin}$ **then**
        assert $x \in Y$
        return $T_1 \to_{\mathsf{lin}} T_2, Y \setminus \{x\}$
      else if $\mathsf{un}(T_1)$ and $m = \mathsf{un}$ **then**
        assert $Y = \emptyset$
        return $T_1 \to_{\mathsf{un}} T_2, \emptyset$
      else return $T_1 \to_{\mathsf{lin}} T_2, Y$
    $\mid \mathbb{e}_1\, \mathbb{e}_2 \Rightarrow$
      $T_1, X := \textsc{CheckExpr}(\Gamma, \mathbb{e}_1);\ T_2, Y := \textsc{CheckExpr}(\Gamma, \mathbb{e}_2)$
      assert $X \cap Y = \emptyset$
      $T_{11} \to_m T_{12} := \textsc{MatchingFun}(T_1)$
      assert $T_2 \lesssim T_{11}$
      return $T_{12}, X \cup Y$

https://doi.org/10.1017/S0956796819000169 Published online by Cambridge University Press

$|\ (\mathbb{e}_1,\ \mathbb{e}_2)_m \Rightarrow$
$\quad T_1, X := \textsc{CheckExpr}(\Gamma, \mathbb{e}_1);\ T_2, Y := \textsc{CheckExpr}(\Gamma, \mathbb{e}_2)$
$\quad$ **assert** $X \cap Y = \emptyset$
$\quad$ **if** $m = \mathsf{un}$ **then assert** $\mathsf{un}(T_1)$ and $\mathsf{un}(T_2)$
$\quad$ **return** $T_1 \times_m T_2, X \cup Y$

$|\ \mathsf{let}\, x_1, x_2 = \mathbb{e}_1\ \mathsf{in}\ \mathbb{e}_2 \Rightarrow$
$\quad T, Y := \textsc{CheckExpr}(\Gamma, \mathbb{e}_1)$
$\quad T_1 \times_m T_2 := \textsc{MatchingProd}(T)$
$\quad U, Z := \textsc{CheckExpr}((\Gamma, x : T_1, y : T_2), \mathbb{e}_2)$
$\quad$ **if** $\mathsf{lin}(T_1)$ **then**
$\qquad$ **assert** $x_1 \in Z$
$\qquad Z := Z \setminus \{x_1\}$
$\quad$ **if** $\mathsf{lin}(T_2)$ **then**
$\qquad$ **assert** $x_2 \in Z$
$\qquad Z := Z \setminus \{x_2\}$
$\quad$ **assert** $Y \cap Z = \emptyset$
$\quad$ **return** $U, Y \cup Z$

$|\ \mathsf{fork}\ \mathbb{e}_1 \Rightarrow$
$\quad T, X := \textsc{CheckExpr}(\Gamma, \mathbb{e}_1)$
$\quad$ **assert** $T \sim \mathsf{unit}$
$\quad$ **return** unit, X

$|\ \mathsf{new}\, S \Rightarrow$ **return** $S \times_{\mathsf{lin}} \overline{S}, \emptyset$

$|\ \mathsf{send}\ \mathbb{e}_1\ \mathbb{e}_2 \Rightarrow$
$\quad T_1, X := \textsc{CheckExpr}(\Gamma, \mathbb{e}_1);\ T_2, Y := \textsc{CheckExpr}(\Gamma, \mathbb{e}_2)$
$\quad$ **assert** $X \cap Y = \emptyset$
$\quad !T_3.\, S := \textsc{MatchingSend}(T_2)$
$\quad$ **assert** $T_1 \lesssim T_3$
$\quad$ **return** $S, X \cup Y$

$|\ \mathsf{receive}\ \mathbb{e}_1 \Rightarrow$
$\quad T_1, X := \textsc{CheckExpr}(\Gamma, \mathbb{e}_1)$
$\quad ?T_2.\, S := \textsc{MatchingReceive}(T_1)$
$\quad$ **return** $T_2 \times_{\mathsf{lin}} S, X$

$|\ \mathsf{select}\, l_j\ \mathbb{e}_1 \Rightarrow$
$\quad T, X := \textsc{CheckExpr}(\Gamma, \mathbb{e}_1)$
$\quad \oplus\{l_j : S_j\} := \textsc{MatchingSelect}(T, l_j)$
$\quad$ **return** S_j, X

$|\ \mathsf{case}\ \mathbb{e}_0\ \mathsf{of}\ \{l_1 : x_1.\, \mathbb{e}_1, \ldots, l_k : x_k.\, \mathbb{e}_k\} \Rightarrow$
$\quad T, X := \textsc{CheckExpr}(\Gamma, \mathbb{e}_0)$
$\quad \&\{l_1 : R_1, \ldots, l_k : R_k\} := \textsc{MatchingCase}(T, \{l_1, \ldots, l_k\})$
$\quad$ **for** $j = 1$ to k **do**
$\qquad U_j, Y_j := \textsc{CheckExpr}((\Gamma, x_j : R_j), \mathbb{e}_j)$
$\qquad$ **assert** $x_j \in Y_j$
$\qquad Y_j := Y_j \setminus \{x_j\}$

https://doi.org/10.1017/S0956796819000169 Published online by Cambridge University Press

```
            assert Y_1 = ··· = Y_k (=: Y)
            U := U_1 ∨ ... ∨ U_k
            assert X ∩ Y = ∅
            return U, X ∪ Y
        | close e_1 ⇒
            T, X := CheckExpr(Γ, e_1)
            assert T ∼ end_!
            return unit, X
        | wait e_1 ⇒
            T, X := CheckExpr(Γ, e_1)
            assert T ∼ end_?
            return unit, X

function MatchingFun(T)
    case T of
        | T_1 →_m T_2 ⇒ return T_1 →_m T_2
        | ⋆ ⇒ return ⋆ →_lin ⋆
        | _ ⇒ error
function MatchingProd(T)
    case T of
        | T_1 ×_m T_2 ⇒ return T_1 ×_m T_2
        | ⋆ ⇒ return ⋆ ×_lin ⋆
        | _ ⇒ error
function MatchingSend(T)
    case T of
        | !T'.S ⇒ return !T'.S
        | ⊛ | ⋆ ⇒ return !⋆.⊛
        | _ ⇒ error
function MatchingReceive(T)
    case T of
        | !T'.S ⇒ return !T'.S
        | ⊛ | ⋆ ⇒ return !⋆.⊛
        | _ ⇒ error
function MatchingSelect(T, l_j)
    case T of
        | ⊕{l_i : S_i}_{i∈I} ⇒
            assert j ∈ I
            return ⊕{l_j : S_j}
        | ⊛ | ⋆ ⇒ return ⊕{l_j : ⊛}
        | _ ⇒ error
function MatchingCase(T, {l_j}_{j∈J})
    case T of
```

https://doi.org/10.1017/S0956796819000169 Published online by Cambridge University Press

$\mid \&\{l_i : S_i\}_{i\in I} \Rightarrow$
if $I \subseteq J$ **then return** $\&\{l_i : S_i\}_{i\in I} \cup \{l_j : \circledast\}_{j\in J\setminus I}$
else error
$\mid \circledast \mid \star \Rightarrow$ **return** $\&\{l_j : \circledast\}_{j\in J}$
$\mid _ \Rightarrow$ **error**

Theorem 11 states soundness of the type-checking algorithm. A few lemmas are required in preparation. Let $\mathsf{rm}(\Gamma, X)$ denote the operation that removes variables X from a type environment Γ.

Lemma 19. *Suppose* $y\colon U \in \Gamma$ *with* $\mathsf{lin}(U)$.

$$\textit{If } \textsc{CheckExpr}(\Gamma, \mathbb{e}) = T, X \textit{ and } y \notin X, \textit{ then } \textsc{CheckExpr}(\mathsf{rm}(\Gamma, \{y\}), \mathbb{e}) = T, X.$$

Proof. By induction on $\mathbb{e}$, we show one important case as follows.

Case $\mathbb{e} = \mathbb{e}_1\ \mathbb{e}_2$: We are given

$$\Gamma(y) = U \qquad \mathsf{lin}(U) \qquad \textsc{CheckExpr}(\Gamma, \mathbb{e}_1\ \mathbb{e}_2) = T, X \qquad y \notin X$$

By the definition of the algorithm, $\textsc{CheckExpr}(\Gamma, \mathbb{e}_i) = T_i, X_i$ for $i = 1, 2$ and

$$T_1 \triangleright T_{11} \rightarrow_m T_{12} \qquad T_2 \lesssim T_{11} \qquad T = T_{12} \qquad X = X_1 \uplus X_2$$

Since $y \notin X$, we have $y \notin X_1$ and $y \notin X_2$. By $\textsc{CheckExpr}(\Gamma, \mathbb{e}_i) = T_i, X_i$ and $y \notin X_i$ and the IH for $i = 1, 2$, we have

$$\textsc{CheckExpr}(\mathsf{rm}(\Gamma, \{y\}), \mathbb{e}_1) = T_1, X_1 \qquad \textsc{CheckExpr}(\mathsf{rm}(\Gamma, \{y\}), \mathbb{e}_2) = T_2, X_2$$

Thus, by the definition of the algorithm, $\textsc{CheckExpr}(\mathsf{rm}(\Gamma, \{y\}), \mathbb{e}_1\ \mathbb{e}_2) = T, X$. □

Lemma 20. *Suppose* $\mathsf{flv}(\Gamma) = X_1 \uplus X_2$. *If* $\Gamma_1 = \mathsf{rm}(\Gamma, X_2)$ *and* $\Gamma_2 = \mathsf{rm}(\Gamma, X_1)$, *then* $\Gamma = \Gamma_1 \circ \Gamma_2$ *and* $\mathsf{flv}(\Gamma_1) = X_1$ *and* $\mathsf{flv}(\Gamma_2) = X_2$.

Proof. By induction on Γ. □

Theorem 11 (Soundness of the type-checking algorithm)**.** *If* $\textsc{CheckExpr}(\Gamma, \mathbb{e}) = T, X$ *and* $\mathsf{flv}(\Gamma) = X$, *then* $\Gamma \vdash_e \mathbb{e} : T$.

Proof. By induction on $\mathbb{e}$, we show main cases as follows.

Case $\mathbb{e} = \lambda_m x{:}T_1.\ \mathbb{e}_1$: We are given

$$\textsc{CheckExpr}(\Gamma, \lambda_m x{:}T_1.\ \mathbb{e}_1) = T, X \qquad \mathsf{flv}(\Gamma) = X$$

We consider only when $m = \mathsf{un}$ and $\mathsf{lin}(T_1)$. By the definition of the algorithm,

$$\textsc{CheckExpr}((\Gamma, x : T_1), \mathbb{e}_1) = T_2, \{x\} \qquad X = \emptyset$$

So, $\mathsf{flv}(\Gamma) = X = \emptyset$. By $\mathsf{lin}(T_1)$, we have $\mathsf{flv}(\Gamma, x : T_1) = \{x\}$. Thus, by the IH, we have $\Gamma, x : T_1 \vdash \mathbb{e}_1 : T_2$. Here, by $\mathsf{flv}(\Gamma) = \emptyset$, we have $\mathsf{un}(\Gamma)$. So, $\mathsf{un}^{:>}(\Gamma)$. We finish by

$$\Gamma, x : T_1 \vdash \mathbb{e}_1 : T_2 \qquad \mathsf{un}^{:>}(\Gamma)$$

and the abstraction rule.

https://doi.org/10.1017/S0956796819000169 Published online by Cambridge University Press

Case $\mathbb{e} = \mathbb{e}_1\ \mathbb{e}_2$: We are given

$$\textsc{CheckExpr}(\Gamma, \mathbb{e}_1\ \mathbb{e}_2) = T, X \qquad \mathsf{flv}(\Gamma) = X$$

By the definition of the algorithm, $\textsc{CheckExpr}(\Gamma, \mathbb{e}_i) = T_i, X_i$ for $i = 1, 2$ and

$$T_1 \triangleright T_{11} \to_m T_{12} \qquad T_2 \lesssim T_{11} \qquad T = T_{12} \qquad X = X_1 \uplus X_2$$

Let $\Gamma_1 = \mathsf{rm}(\Gamma, X_2)$ and $\Gamma_2 = \mathsf{rm}(\Gamma, X_1)$. By Lemma 20, $\Gamma = \Gamma_1 \circ \Gamma_2$ and $\mathsf{flv}(\Gamma_1) = X_1$ and $\mathsf{flv}(\Gamma_2) = X_2$. By $\textsc{CheckExpr}(\Gamma, \mathbb{e}_i) = T_i, X_i$ and Lemma 19, we have $\textsc{CheckExpr}(\Gamma_i, \mathbb{e}_i) = T_i, X_i$ for $i = 1, 2$. By $\mathsf{flv}(\Gamma_i) = X_i$ and $\textsc{CheckExpr}(\Gamma_i, \mathbb{e}_i) = T_i, X_i$ and the IH, we have $\Gamma_i \vdash \mathbb{e}_i : T_i$ for $i = 1, 2$. We finish by

$$\Gamma_1 \vdash \mathbb{e}_1 : T_1 \qquad \Gamma_2 \vdash \mathbb{e}_2 : T_2 \qquad T_1 \triangleright T_{11} \to_m T_{12} \qquad T_2 \lesssim T_{11}$$

and the application rule. □

We will also show the converse of the theorem above. Completeness states that $\textsc{CheckExpr}(\Gamma, \mathbb{e})$ computes a minimal type with respect to negative subtyping.

Theorem 12 (Completeness of the type-checking algorithm)**.** *If* $\Gamma \vdash_e \mathbb{e} : T$, *then* $\textsc{CheckExpr}(\Gamma, \mathbb{e}) = T', X$ *and* $T' <:^- T$ *and* $\mathsf{flv}(\Gamma) = X$ *for some* T'.

To prove this theorem, we need a stronger statement, namely Lemma 24. We define environment positive consistent subtyping, written $\Gamma' <:^- \Gamma$, as $\mathsf{dom}(\Gamma) \subseteq \mathsf{dom}(\Gamma')$ and $\Gamma(x) <:^- \Gamma'(x)$, for any $x \in \mathsf{dom}(\Gamma)$. Then, the theorem follows from the fact that $<:^-$ on type environments is reflexive. We start with a few lemmas about $<:^-$.

Lemma 21.

1. *If* $T'_1 <:^- T_1$ *and* $T_1 \triangleright T_{11} \to_m T_{12}$, *then there exist some* T'_{11}, T'_{12}, *and* n *such that* $\textsc{MatchingFun}(T'_1) = T'_{11} \to_n T'_{12}$ *and* $T'_{11} \to_n T'_{12} <:^- T_{11} \to_m T_{12}$.
2. *If* $T'_1 <:^- T_1$ *and* $T_1 \triangleright T_{11} \times_m T_{12}$, *then there exist some* T'_{11}, T'_{12}, *and* n *such that* $\textsc{MatchingProd}(T'_1) = T'_{11} \times_n T'_{12}$ *and* $T'_{11} \times_n T'_{12} <:^- T_{11} \times_m T_{12}$.
3. *If* $T'_1 <:^- T_1$ *and* $T_1 \triangleright\ !T_{11}.S_{12}$, *then there exist some* T'_{11} *and* S'_{12} *such that* $\textsc{MatchingSend}(T'_1) =\ !T'_{11}.S'_{12}$ *and* $!T'_{11}.S'_{12} <:^-\ !T_{11}.S_{12}$.
4. *If* $T'_1 <:^- T_1$ *and* $T_1 \triangleright\ ?T_{11}.S_{12}$, *then there exist some* T'_{11} *and* S'_{12} *such that* $\textsc{MatchingReceive}(T'_1) =\ ?T'_{11}.S'_{12}$ *and* $?T'_{11}.S'_{12} <:^-\ ?T_{11}.S_{12}$.
5. *If* $T'_1 <:^- T_1$ *and* $T_1 \triangleright \oplus\{l_j : S_j\}$, *then there exist some* S'_j *such that* $\textsc{MatchingSelect}(T'_1, l_j) = \oplus\{l_j : S'_j\}$ *and* $\oplus\{l_j : S'_j\} <:^- \oplus\{l_j : S_j\}$.
6. *If* $T'_1 <:^- T_1$ *and* $T_1 \triangleright \&\{l_1 : S_1, \ldots, l_k : S_k\}$, *then there exist some* $S'_1, \ldots, S'_k$ *such that* $\textsc{MatchingCase}(T'_1, \{l_1, \ldots, l_k\}) = \&\{l_1 : S'_1, \ldots, l_k : S'_k\}$ *and* $\&\{l_1 : S'_1, \ldots, l_k : S'_k\} <:^- \&\{l_1 : S_1, \ldots, l_k : S_k\}$.

Proof. By case analysis on $T \triangleright U$. □

Lemma 22. *If* $T <:^- U$ *and* $\mathsf{un}(U)$, *then* $\mathsf{un}(T)$.

Proof. By case analysis on $T <:^- U$. □

https://doi.org/10.1017/S0956796819000169 Published online by Cambridge University Press

Lemma 23.

1. *If* $T_1 <:^- T_2$ *and* $T_2 \lesssim T_3$*, then* $T_1 \lesssim T_3$*.*
2. *If* $T_1 \lesssim T_2$ *and* $T_2 <:^+ T_3$*, then* $T_1 \lesssim T_3$*.*

Proof. Both items are proved by simultaneous induction on $\lesssim$. □

Lemma 24. *If* $\Gamma \vdash_e \mathbb{e} : T$ *and* $\Gamma' <:^- \Gamma$*, then there exists* T' *such that* $\textsc{CheckExpr}(\Gamma', \mathbb{e}) = T', X$ *and* $T' <:^- T$ *and* $\mathsf{flv}(\Gamma) = X$*.*

Proof. By induction on $\mathbb{e}$, we show main cases as follows.

Case $\mathbb{e} = \mathbb{e}_1\ \mathbb{e}_2$: By inversion of the typing relation, $\Gamma_1 \vdash_e \mathbb{e}_1 : T_1$ and $\Gamma_2 \vdash_e \mathbb{e}_2 : T_2$ and $\Gamma = \Gamma_1 \circ \Gamma_2$ and $T_1 \rhd T_{11} \rightarrow_m T_{12}$ and $T_2 \lesssim T_{11}$ for some Γ_1, Γ_2, T_1, T_2, T_{11}, T_{12}, and m. It is easy to show $\Gamma' <:^- \Gamma_1$ and $\Gamma' <:^- \Gamma_2$. By the induction hypothesis, for some T'_1, T'_2, X, and Y, $T'_1, X = \textsc{CheckExpr}(\Gamma', \mathbb{e}_1)$ and $T'_1 <:^- T_1$ and $\mathsf{flv}(\Gamma_1) = X$ and $T'_2, Y = \textsc{CheckExpr}(\Gamma', \mathbb{e}_2)$ and $T'_2 <:^- T_2$ and $\mathsf{flv}(\Gamma_2) = Y$. Since $\Gamma_1 \circ \Gamma_2$ is well defined, $X \cap Y$ must be $\emptyset$. By Lemma 21, $T'_{11} \rightarrow_n T'_{12} = \textsc{MatchingFun}(T'_1)$ and $T'_{11} \rightarrow_n T'_{12} <:^- T_{11} \rightarrow_m T_{12}$ for some T'_{11} and T'_{12}. By inversion of $<:^-$, we have $T_{11} <:^+ T'_{11}$ and $T'_{12} <:^- T_{12}$. Then, $T'_2 \lesssim T'_{11}$ is shown by Lemma 23. It is easy to show $X \cup Y = \mathsf{flv}(\Gamma)$ because $\Gamma = \Gamma_1 \circ \Gamma_2$. Finally, $T'_{12} <:^- T_{12}$ finishes the case.

Case $\mathbb{e} = \mathsf{case}\ \mathbb{e}_0\ \mathsf{of}\ \{l_j : x_j.\ \mathbb{e}_j\}_{j \in J}$: By inversion of the typing relation, we have $\Gamma_1 \vdash_e \mathbb{e}_0 : T_0$ and $T_0 \rhd \&\{l_j : R_j\}_{j \in J}$ and $(\Gamma_2, x_j : R_j \vdash_e \mathbb{e}_j : U_j)_{j \in J}$ and $T = \vee\{U_j\}_{j \in J}$ and $\Gamma = \Gamma_1 \circ \Gamma_2$ for some Γ_1, Γ_2, R_j, and U_j (for $j \in J$). It is easy to show $\Gamma' <:^- \Gamma_1$ and $\Gamma' <:^- \Gamma_2$. By the induction hypothesis, $T'_0, X = \textsc{CheckExpr}(\Gamma', \mathbb{e}_0)$ and $T'_0 <:^- T_0$ and $\mathsf{flv}(\Gamma_1) = X$ for some T'_0 and X. By Lemma 21, $\textsc{MatchingCase}(T'_0, \{l_1, \ldots, l_k\}) = \&\{l_1 : R'_1, \ldots, l_k : R'_k\}$ and $\&\{l_j : R'_j\}_{j \in J} <:^- \&\{l_j : R_j\}_{j \in J}$ for some $(R'_j)_{j \in J}$. By inversion of $<:^-$, we have $(R'_j <:^- R_j)_{j \in J}$. By the induction hypothesis, for any $j \in J$, there exist U'_j and Y_j such that $U'_j, Y_j = \textsc{CheckExpr}((\Gamma', x_j : R'_j), \mathbb{e}_j)$ and $U'_j <:^- U_j$ and $\mathsf{flv}(\Gamma_2, x_j : R_j) = Y_j$. It is easy to show that $x_j \in Y_j$ for any $j \in J$ and $Y_1 = \cdots = Y_k$ and $X \cap Y_1 = \emptyset$ because R_j is linear and $\Gamma_1 \circ \Gamma_2$ is well defined. It is also easy to show $X \cup (Y_1 \setminus \{x_1\}) = \mathsf{flv}(\Gamma)$ because $\Gamma = \Gamma_1 \circ \Gamma_2$. Finally, $\vee\{U'_j\}_{j \in J} <:^- \vee\{U_j\}_{j \in J}$ is shown by Lemma 15. □

https://doi.org/10.1017/S0956796819000169 Published online by Cambridge University Press