Splitting Is Stressful but Merging Is Manageable

Co-contextual Typing for Substructural and Session Types

Olivia Weston[a] and Simon Fowler[a]

a University of Glasgow, Scotland, UK

Abstract **Context** Behavioural type systems such as session types, typestate, and mailbox types allow a developer to specify and check behavioural properties within their application code, giving programs that are correct-by-construction. Session types in particular allow a developer that communication in their code follows pre-defined protocols.

Inquiry Implementing session types in functional languages requires some form of resource control, most commonly achieved through the use of linear typing. However, linear type systems as specified declaratively *cannot be implemented algorithmically*. Even when using techniques such as leftover typing, implementing other substructural modes (e.g., affine and relevant) requires ad-hoc changes to typing rules.

Approach We investigate *co-contextual type systems* as a general framework for implementing substructural type systems. Whereas standard type systems require a type environment as an *input* to a typechecking algorithm, *co-contextual* type systems—a class of type system originally introduced for efficient incremental typechecking—produce a type environment as an *output* of the typechecking judgement.

Knowledge We show how to generalise co-contextual type systems to support algorithmic implementations of substructural typing: simple changes to typechecker parameters allow us to instantiate our typechecker to check unrestricted, linear, affine, and relevant λ-calculi. Building on this approach, we present a second language with an extended constraint system that can support mixed-linearity typing, and we use this to implement an algorithmic type system for the GV session-typed functional language.

Grounding We prove four instantiations of our algorithm sound and complete with respect to declarative unrestricted, linear, affine, and relevant λ-calculi. We prove our our co-contextual session typing algorithm sound and complete with respect to a declarative presentation of GV. Finally, we implement our typechecking algorithm in OCaml and demonstrate it on a series of case study applications.

Importance This work shows the versatility of co-contextual typing as a framework for supporting algorithmic implementations of substructural type systems, in addition to its originally-presented benefits (e.g., support for incremental and parallel typechecking). Our work further lays the groundwork for implementing programming languages with more sophisticated type combination operations that are out-of-reach of current approaches.

ACM CCS 2012

- **Computing methodologies** → Distributed programming languages; Concurrent programming languages;
- **Software and its engineering** → *Syntax*; **Functional languages**; **Concurrent programming structures**; *Data types and structures*; Constraints;

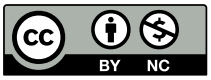
Keywords algorithmic type systems, substructural types, session types

The Art, Science, and Engineering of Programming

Submitted June 1, 2026

Published October 15, 2026

DOI 10.22152/programming-journal.org/2026/11/7



© Olivia Weston and Simon Fowler
This work is licensed under a "CC BY-NC 4.0" license
In *The Art, Science, and Engineering of Programming*, vol. 11, no. 3, 2026, article 7; 29 pages.

1 Introduction

Typed functional programming languages statically rule out many classes of erroneous programs. For example, a record data type such as $\mathsf{AuthData} = (\mathsf{username} : \mathsf{String}, \mathsf{password} : \mathsf{SecureString})$ statically ensures that an AuthData type consists of a username and password, and rules out errors like multiplying a username by 2.

Concurrent and distributed programming is notoriously difficult to get right, yet errors can lead to significant problems if undetected. As the need for robust distributed software grows, *behavioural types* have emerged as a lightweight way of ensuring correctness of the *behaviour* of a program: for example allowing the programmer to rule out code that would cause communication errors, like protocol violations and deadlocks at compile time. An example behavioural type is the *session type* $\mathsf{AuthProto} = ?\mathsf{AuthData}.!\mathsf{Bool}.\mathsf{end}_!$. The AuthProto type states that a server should receive an AuthData record and respond with a Boolean value denoting whether the credentials are correct.

In spite of their promise for ensuring correct code, implementing typecheckers for behavioural type systems is generally more complex, since behavioural type systems must track the usage of variables in a program at any given point. Many behavioural type systems are not specified algorithmically and therefore do not immediately lend themselves to an implementation, hindering their adoption.

This paper shows how we can generalise *co-contextual typing*, a methodology originally designed for incremental typechecking, to support the resource control needed for implementing substructural functional programming languages. Additionally, we build upon an extension of this language to implement the GV session-typed language as a case study.

1.1 Session Types

Session types are a behavioural type system for communication protocols, first introduced by Honda [12] and which have spawned an influential line of research since. A session type describes a structured sequence of interactions between participants. We concentrate on *binary* (2-party) session types in this paper, but since our main contribution is on the algorithmic typechecking of linear types, we are confident that our ideas would also extend to languages based on multiparty session types [14].

For example, consider the following session type that describes interaction with a cash machine (ATM):

$$\mathsf{ATMServer} \triangleq ?(\mathsf{CardNumber} \times \mathsf{PIN}) . \oplus \left\{ \begin{array}{l} \mathsf{Authenticated}: \& \left\{ \begin{array}{l} \mathsf{Withdraw}: ?\mathsf{Amount} . !\mathsf{Money} . \mathsf{end}_!, \\ \mathsf{Balance}: !\mathsf{Balance} . \mathsf{end}_!, \\ \mathsf{Quit}: \mathsf{end}_! \end{array} \right\}, \\ \mathsf{WrongPIN}: \mathsf{end}_! \end{array} \right\}$$

The ATM first receives a card number and PIN code. If it is incorrect, then the ATM will select the WrongPIN branch and terminate the session. If it is correct, then the ATM will select the Authenticated branch and then offer three options: Withdraw, at

```
atmServer : ATMServer → end!
atmServer ≜ λs.
  let ((cardNum, pin), s) = receive s in
  if pinCorrect (cardNum, pin) then
    let s = select Authenticated s in
    offer s {
      Withdraw(s) ↦
        let (amount, s) = receive s in
        decreaseBalance(cardNum, amount);
        send dispense(amount) s
      Balance(s) ↦ send balance(cardNum) s,
      Quit(s) ↦ s
    }
  else select WrongPIN s
```

(a) ATM server process

```
atmClient : ATMClient → Unit
atmClient ≜ λc.
  let c = send ("4659 1234 1234 1234",
      "5710") c in
  offer c {
    Authenticated(c) ↦
      let c = select Withdraw c in
      let c = send 1000 c in
      let (cash, c) = receive c in
      spend(cash);
      wait c,
    WrongPIN(c) ↦ wait c
  }
```

(b) ATM client process

Figure 1 Implementations of the ATM and client

which point the user will send the amount of money to withdraw and the ATM will send the money; Balance, at which point the ATM will send the current balance; or Quit, at which point the ATM will close the session.

We can view the protocol from the side of the client as follows:

$$\mathsf{ATMClient} \triangleq !(\mathsf{CardNumber} \times \mathsf{PIN}) \,.\, \& \left\{ \begin{array}{l} \mathsf{Authenticated}\colon \oplus \left\{ \begin{array}{l} \mathsf{Withdraw}\colon !\mathsf{Amount} \,.\, ?\mathsf{Money} \,.\, \mathsf{end}_?, \\ \mathsf{Balance}\colon ?\mathsf{Balance} \,.\, \mathsf{end}_?, \\ \mathsf{Quit}\colon \mathsf{end}_? \end{array} \right\}, \\ \mathsf{WrongPIN}\colon \mathsf{end}_? \end{array} \right\}$$

Here the client of the ATM has the *dual* session type. Where the ATM sends, the client receives, and vice-versa. Inspired by logical accounts of session types [4, 20, 30] we have two dual types $\mathsf{end}_!$ and $\mathsf{end}_?$ to indicate the end of a session.

Figure 1 shows expressions implementing the cash machine and its client. The atmServer definition (Figure 1a) is a function that takes a *channel endpoint s* with session type ATMServer. First, the server uses the **receive** construct to receive the account number and PIN code from the client. The **receive** construct returns a pair of the received value as well as a copy of the channel endpoint with an updated type to be used in the remainder of the protocol.

Figure 1b shows an implementation of the client of the ATM. The client begins by sending the card number and credentials to the ATM and proceeds to offer branches based on the authentication result. In the Authenticated branch, the client selects the Withdraw option, requests a withdrawal of 1000 units of currency, and receives the cash before using **wait** *s* to dispose of the finished session. In the case of the WrongPIN branch the client closes the connection immediately.

1.2 The Need for Linearity

The type of a session endpoint (e.g., s and c) is updated by *re-binding* an endpoint as the result of a communication operation (e.g., **send** or **receive**). However, this alone is *not enough* to ensure safety. For example, by ignoring the endpoint returned from the receive expression after a withdrawal, we could write a program that receives the money multiple times:

$$\textbf{let}\,(cash1, _) = \textbf{receive}\,s\,\textbf{in}\;\textbf{let}\,(cash2, _) = \textbf{receive}\,s\,\textbf{in}\;\textbf{let}\,(cash3, s) = \textbf{receive}\,s\,\textbf{in}\;\ldots$$

For this reason, systems making use of session types commonly require *linear types*, meaning that each session channel endpoint must be used *precisely once*. Whereas implementing a typechecker for the simply-typed λ-calculus (STLC) is straightforward to implement algorithmically (assuming type annotations on function parameters), typechecking a linear λ-calculus is more challenging. To illustrate, consider writing a typechecker (with OCaml type signature `typecheck : (variable * ty) list -> expr -> ty`) for variables and let-expressions in the STLC:

$$\frac{x : A \in \Gamma}{\Gamma \vdash x : A} \qquad \frac{\Gamma \vdash M : A \qquad \Gamma, x : A \vdash M : B}{\Gamma \vdash \textbf{let}\; x = M\; \textbf{in}\; N : B}$$

```
1  typecheck env (EVar var) =
2    List.assoc var env
```

```
1  typecheck env (ELet (var, e1, e2)) =
2    let ty1 = typecheck env e1 in
3    let ext_env = (var, ty1) :: env in
4    typecheck ext_env e2
```

The variable rule (left) states that a variable x can be given type A if the type environment Γ contains an entry mapping x to type A, and implementing the variable rule involves looking up the variable's type in the environment. The rule for checking a **let**-binder (right) involves typechecking the first subexpression to produce type A, and then typechecking the second subexpression under an environment extended with a mapping from x to A.

The linear λ-calculus enforces linearity by making two changes to the usual λ-calculus typing rules: first, variables must be typed in a singleton environment (meaning that every variable must be used at least once); second, the environment must be *split* when typing subexpressions (meaning that every variable must only be used at most once). We can see the adapted rules and attempts at implementations as follows:

$$\frac{}{x : A \vdash x : A} \qquad \frac{\Gamma_1 \vdash M : A \qquad \Gamma_2, x : A \vdash M : B}{\Gamma_1, \Gamma_2 \vdash \textbf{let}\; x = M\; \textbf{in}\; N : B}$$

```
1  typecheck env (EVar var) =
2    if List.length env == 1 then
3      List.assoc var env
4    else
5      raise "Linearity error"
```

```
1  typecheck env (ELet (var, e1, e2)) =
2    let (env1, env2) = split env in
3    let ty1 = typecheck env1 e1 in
4    let ext_env = (var, ty1) :: env2 in
5    typecheck ext_env e2
```

At first glance these definitions seem to make sense, but the difficulty comes with the `split env` expression on line 2 of the let-rule. When implementing the rule we need to

read it bottom-up, and it is not immediately clear *how* to split the environment: which mappings go into env1, and which go into env2? Indeed, the rule is not *algorithmic*.

There are several implementation strategies that allow us to implement linearity. For example, we could annotate each subterm with its free variables (similar in spirit to the co-de-Bruijn representation [22]), and use this to guide the split (as done in e.g., [27]), but this would require an additional tree traversal and additional memory.

In practice, there are two main approaches to implementing linearity. *Subtractive* approaches [31] modify the typing judgement to have an input and output environment ($\Gamma_1 \vdash M : A \dashv \Gamma_2$), where a variable is "removed" from an environment by the variable rule. *Additive* approaches, as used in quantitative type theory [2] and language implementations such as Granule [25] and Links [5], annotate each variable binding in a type environment with a usage annotation.

In this paper we propose a different approach based on *co-contextual typing* [8]. Originally defined for the design of efficient incremental typecheckers, co-contextual type systems work by producing a type environment and a set of constraints as an *output*, as opposed to requiring a type environment as an *input*.

Our key observation is that it is far easier to *merge* two environments that are produced by typing two subterms than it is to *split* an environment. We show how a generalisation of the original system presented by Erdweg, Bračevac, Kuci, Krebs, and Mezini [8] allows us to describe unrestricted, linear, affine, and relevant type systems *without* changing the typing rules. This gives us typechecking algorithms for many substructural bases by simply changing three parameters, though the metatheory must be proved separately. Building on this foundation we introduce an extended language that shows how co-contextual typing can be used to implement a typechecker for the (finite and monomorphic) GV session-typed functional language [20, 30], and prove our implementation sound and complete with respect to a declarative presentation of the rules.

Contributions This work provides the following original contributions:

- A generalisation of Erdweg, Bračevac, Kuci, Krebs, and Mezini [8]'s co-contextual typing system to support substructural typing, along with instantiations for unrestricted, linear, affine, and relevant modes (Section 2). We show that the co-contextual systems are sound and complete with respect to their declarative counterparts (Section 2.5.2). All proofs can be found in the extended version [36].
- An extension of the base co-contextual typing calculus to support simultaneous use of resourceful and restricted types, also known as a 'mixed linearity' system; we use this as a basis for implementing an algorithmic version of the GV session-typed λ-calculus (Section 3). Again we prove this formulation sound and complete with respect to its declarative counterpart (Section 3.3). All proofs can be found in the extended version [36].
- An OCaml implementation of the co-contextual formulation of GV, along with several examples (Section 4) [35].

2 Co-contextual Typing for Substructural Types

In this section we firstly introduce co-contextual typing as presented by Erdweg, Bračevac, Kuci, Krebs, and Mezini [8], before showing our generalisation of the system to support substructural typing.

Syntax

$$\begin{array}{llll} \text{Terms} & L,M,N & ::= & x \mid c \mid \lambda x.M \\ & & & \mid M\ N \mid M \odot N \end{array}$$

$$\begin{array}{llll} \text{Types} & A,B & ::= & C \mid A \to B \\ \text{Base types} & C & ::= & \mathsf{Int} \mid \mathsf{Bool} \mid \cdots \\ \text{Type environments} & \Gamma & ::= & \cdot \mid \Gamma, x:A \end{array}$$

Declarative Typing Rules $\boxed{\Gamma \vdash M : A}$

$$\frac{x:A \in \Gamma}{\Gamma \vdash x : A}\ \text{T-Var} \qquad \frac{c \text{ has base type } C}{\Gamma \vdash c : C}\ \text{T-Const} \qquad \frac{\Gamma, x:A \vdash M : B}{\Gamma \vdash \lambda x.M : A \to B}\ \text{T-Abs}$$

$$\frac{\Gamma \vdash M : A \to B \qquad \Gamma \vdash N : A}{\Gamma \vdash M\ N : B}\ \text{T-App} \qquad \frac{\mathsf{ty}(\odot) = A_1 \to A_2 \to B \qquad \Gamma \vdash M : A_1 \qquad \Gamma \vdash N : A_2}{\Gamma \vdash M \odot N : B}\ \text{T-Op}$$

Figure 2 Declarative typing rules for the simply-typed λ-calculus

2.1 Declarative Core Language

We begin by introducing a declarative core language based on the STLC. Figure 2 shows the syntax and declarative typing rules of the simply-typed λ-calculus with binary operators; we will later extend this language with sums and products. Types A, B include base types C, (we assume at least Boolean types Bool and integer types Int) and function types $A \to B$. The typing judgement has the form $\Gamma \vdash M : A$, read "under environment Γ, term M has type A" and the rules are standard (where T-Op types a binary operator $\odot$ such as $+$ or $-$). Note that, as written, the typing rules are not algorithmic due to the lack of an annotation on λ-abstractions.

2.2 Co-contextual Typing

Figure 3 shows the *co-contextual* formulation of the typing rules from Figure 2, as originally described by Erdweg, Bračevac, Kuci, Krebs, and Mezini [8]. We have modified the syntax of their typing judgement and modified the TE-Abs rule to not require a type annotation. We let X, Y range over *type variables* that stand for an as-yet unknown part of a type; we let τ, σ range over types that may contain type variables; and let Δ range over environments whose types may contain type variables.

The typing judgement has the form $M : \tau \triangleright \Delta \,;\, \mathscr{C}$, which can be read "Term M has type τ, producing environment Δ and constraint set $\mathscr{C}$". Unlike the declarative typing rules, it can be implemented algorithmically: term M is an input to the typechecking algorithm, while τ, Δ, and $\mathscr{C}$ are *outputs* of the algorithm. The fact that Δ is an *output* of the judgement is key to allowing us to support multiple modes of substructural typing.

As with typical type inference algorithms, $\mathscr{C}$ is a set of equality constraints of the form $\tau \doteq \sigma$ constraining τ to be equal to σ. Standard techniques such as first-order unification [28] can be used to produce a solution to the constraint set. This constraint set will help us solve all the open types and type variables into closed types later on.

The original work does not aim to relate a declarative presentation and an algorithmic presentation, rather, it aims to relate a standard type inference system with a co-contextual type inference system. In contrast, the aim of our work is to show how (non-algorithmic) declarative presentations of substructural typing can be implemented algorithmically using co-contextual typing.

Additional Syntax

$$\begin{array}{llll} \text{Type Variables} & X, Y & & \\ \text{Base types} & C & ::= & \mathsf{Int} \mid \mathsf{Bool} \mid \cdots \end{array} \qquad \begin{array}{llll} \text{Open Types} & \tau, \sigma, \rho & ::= & \underbrace{C \mid X} \mid \tau \to \sigma \\ \text{Constraint Sets} & \mathscr{C} & ::= & \tau \doteq \sigma \end{array}$$

Co-contextual Typing Rules $\boxed{M : \tau \triangleright \Delta \,;\, \mathscr{C}}$

$$\text{TE-Var}\ \frac{X \text{ is fresh}}{x : X \triangleright x : X \,;\, \emptyset} \qquad \text{TE-Abs}\ \frac{X \text{ is fresh} \quad M : \tau \triangleright \Delta \,;\, \mathscr{C}_1 \quad \mathscr{C}_2 = \{X \doteq \Delta(x) \mid \text{if } x \in \mathsf{dom}(\Delta)\}}{\lambda x.M : X \to \tau \triangleright \Delta - x \,;\, \mathscr{C}_1 \cup \mathscr{C}_2}$$

$$\text{TE-App}\ \frac{M : \tau \triangleright \Delta_1 \,;\, \mathscr{C}_1 \quad N : \sigma \triangleright \Delta_2 \,;\, \mathscr{C}_2 \quad \mathsf{merge}(\Delta_1, \Delta_2) = (\Delta, \mathscr{C}_3) \quad X \text{ is fresh}}{M\,N : X \triangleright \Delta \,;\, \mathscr{C}_1 \cup \mathscr{C}_2 \cup \mathscr{C}_3 \cup \{\tau \doteq \sigma \to X\}}$$

$$\text{TE-Const}\ \frac{c \text{ has base type } C}{c : C \triangleright \cdot \,;\, \emptyset} \qquad \text{TE-Op}\ \frac{\mathsf{ty}(\odot) = A_1 \to A_2 \to B \quad M : \tau \triangleright \Delta_1 \,;\, \mathscr{C}_1 \quad N : \sigma \triangleright \Delta_2 \,;\, \mathscr{C}_2 \quad \mathsf{merge}(\Delta_1, \Delta_2) = (\Delta, \mathscr{C}_3)}{M \odot N : B \triangleright \Delta \,;\, \mathscr{C}_1 \cup \mathscr{C}_2 \cup \mathscr{C}_3 \cup \{\tau \doteq A_1, \sigma \doteq A_2\}}$$

Meta-level Definition

$$\begin{array}{l} \mathsf{merge}(\Delta_1, \Delta_2) = (\Delta, \mathscr{C}) \\ \quad \text{where} \quad \Delta = \Delta_1 \cup \{x : \Delta_2(x) \mid x \in \mathsf{dom}(\Delta_2) \setminus \mathsf{dom}(\Delta_1)\} \\ \qquad\quad\ \ \mathscr{C} = \{\Delta_1(x) \doteq \Delta_2(x) \mid x \in \mathsf{dom}(\Delta_1) \cap \mathsf{dom}(\Delta_2)\} \end{array}$$

Figure 3 Co-contextual typing rules for the simply-typed λ-calculus [8]

Typing rules Rule TE-Var types a variable. Since co-contextual type systems do not use a type environment as an input, we do not lookup types. Instead, the rule generates a fresh, unconstrained type variable X and produces a singleton environment $x : X$.

Rule TE-Const types a constant, producing an empty environment and constraint set. Rule TE-Abs shows how to type a function abstraction $\lambda x.M$: similarly to typical type inference systems the rule generates a fresh type variable as a placeholder for the argument type. Next, the rule checks the body M, producing type τ, environment Δ, and constraint set $\mathscr{C}_1$. Finally, if x occurs in Δ, then the rule constrains X to the inferred type of x in Δ. The produced type is function type $X \to \tau$; the produced environment is Δ with x removed (since x is bound by the abstraction); and the produced constraints are the union of the constraints produced by checking the body and checking the parameter type.

The TE-App rule types function application. We create a fresh type variable X for the result of the function application, then typecheck the function M, and argument N. Next, we *combine* the two produced environments. We do this using the merge meta-function: the resulting environment is the union of Δ_1 with the non-overlapping mappings in Δ_2. We generate equality constraints for each overlapping mapping. Given that $\mathsf{merge}(\Delta_1, \Delta_2) = (\Delta, \mathscr{C}_3)$, the overall return type is X; the produced type environment is Δ; the produced constraint set is the union of all constraint sets along with an additional constraint specifying that τ must have the shape $\sigma \to X$.

Finally, rule TE-Op types a binary operator, following the same pattern as before: we typecheck both subterms, each producing environments and constraint sets. We merge the produced environments and constrain the inferred types to those specified by the type of the operator, returning the result type.

Example As an example, let us contrast the contextual and co-contextual typing derivations for the expression $(\lambda x.1 * x)\,7$. The contextual derivation is as follows:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{x : \mathsf{Int} \vdash 1 : \mathsf{Int}} \quad \dfrac{}{x : \mathsf{Int} \vdash x : \mathsf{Int}}}{x : \mathsf{Int} \vdash 1 * x : \mathsf{Int}}}{\cdot \vdash \lambda x.1 * x : \mathsf{Int} \to \mathsf{Int}} \quad \dfrac{}{\cdot \vdash 7 : \mathsf{Int}}}{\cdot \vdash (\lambda x.1 * x)\,7 : \mathsf{Int}}$$

Let us now look at the equivalent co-contextual derivation. First, let **D** be the following subderivation:

$$\dfrac{\dfrac{1 \text{ has base type } \mathsf{Int}}{1 : \mathsf{Int} \triangleright \cdot ; \emptyset} \quad \dfrac{X \text{ is fresh}}{x : X \triangleright (x : X) ; \emptyset} \quad \begin{array}{l}\mathsf{ty}(*) = \mathsf{Int} \to \mathsf{Int} \to \mathsf{Int} \\ \mathsf{merge}(\cdot, x : X) = (x : X, \emptyset)\end{array}}{1 * x : \mathsf{Int} \triangleright (x : X) ; \{X \doteq \mathsf{Int}, \mathsf{Int} \doteq \mathsf{Int}\}}$$

Then, we can construct the following co-contextual derivation:

$$\dfrac{\begin{array}{l}\mathbf{D} \\ \mathscr{C} = \{Y \doteq X, X \doteq \mathsf{Int}, \mathsf{Int} \doteq \mathsf{Int}\} \\ Y \text{ is fresh}\end{array}}{\lambda x.1 * x : Y \to \mathsf{Int} \triangleright \cdot ; \mathscr{C}} \quad \dfrac{7 \text{ has base type } \mathsf{Int}}{7 : \mathsf{Int} \triangleright \cdot ; \emptyset} \quad \begin{array}{l}Z \text{ is fresh} \\ \mathsf{merge}(\cdot, \cdot) = (\cdot, \emptyset)\end{array}$$
$$(\lambda x.1 * x)\,7 : Z \triangleright \cdot ; \mathscr{C} \cup \{Y \to \mathsf{Int} \doteq \mathsf{Int} \to Z\}$$

In this case the final constraint set is $\{Y \doteq X, X \doteq \mathsf{Int}, \mathsf{Int} \doteq \mathsf{Int}, Y \to \mathsf{Int} \doteq \mathsf{Int} \to Z\}$. Solving this constraint set via first-order unification gives us a solution $Y \mapsto X, Z \mapsto \mathsf{Int}, X \mapsto \mathsf{Int}$ which resolves the expression's type Z to Int.

Sums, products, and units Figure 4 shows the usual typing rules for sum, product, and unit types. We also provide a co-contextual version of the case elimination rule (an adaptation of the rule for records described by [8]); the remaining corresponding co-contextual rules are in the extended version [36]. We make use of common syntactic sugar shorthands for let constructs and if-then-else constructs. We define $\mathbf{let}\, x = M\, \mathbf{in} N$ as $(\lambda x.N)\,M$. We define $\mathbf{if}\, L\, \mathbf{then}\, M\, \mathbf{else} N$ by first defining $\mathsf{true} \triangleq \mathbf{inl}()$ and $\mathsf{false} \triangleq \mathbf{inr}()$, then by defining the if construction as $\mathbf{case}\ L\ \{\mathbf{inl}\,() \mapsto M ; \mathbf{inr}\,() \mapsto N\}$.

Declarative typing rules for sums, products, and units $\boxed{\Gamma \vdash M : A}$

$$\textsc{T-Unit}\ \frac{}{\Gamma \vdash () : \mathsf{Unit}} \qquad \textsc{T-Seq}\ \frac{\Gamma \vdash M : \mathsf{Unit} \qquad \Gamma \vdash N : A}{\Gamma \vdash M ; N : A} \qquad \textsc{T-Inl}\ \frac{\Gamma \vdash M : A}{\Gamma \vdash \mathbf{inl}\, M : (A + B)} \qquad \textsc{T-Inr}\ \frac{\Gamma \vdash M : B}{\Gamma \vdash \mathbf{inr}\, M : (A + B)}$$

$$\textsc{T-Case}\ \frac{\Gamma \vdash L : (A_1 + A_2) \qquad \Gamma, x : A_1 \vdash M : B \qquad \Gamma, y : A_2 \vdash N : B}{\Gamma \vdash \mathbf{case}\, L\, \{\mathbf{inl}\, x \mapsto M, \mathbf{inr}\, y \mapsto N\} : B} \qquad \textsc{T-Pair}\ \frac{\Gamma \vdash M : A \qquad \Gamma \vdash N : B}{\Gamma \vdash (M, N) : (A \times B)}$$

$$\textsc{T-LetPair}\ \frac{\Gamma \vdash M : (A_1 \times A_2) \qquad \Gamma, x : A_1, y : A_2 \vdash N : B}{\Gamma \vdash \mathbf{let}\, (x, y) = M\, \mathbf{in}\, N : B}$$

Co-contextual typing rule for case-elimination $\boxed{M : \tau \triangleright \Delta \,;\, \mathscr{C}}$

$$\textsc{TE-Case}\ \frac{\begin{array}{c} L : \tau \triangleright \Delta_1 \,;\, \mathscr{C}_1 \qquad M : \sigma_1 \triangleright \Delta_2 \,;\, \mathscr{C}_2 \qquad N : \sigma_2 \triangleright \Delta_3 \,;\, \mathscr{C}_3 \qquad \mathsf{merge}((\Delta_2 - x), (\Delta_3 - y)) = (\Delta_4, \mathscr{C}_4) \\ \mathsf{merge}(\Delta_1, \Delta_4) = (\Delta, \mathscr{C}_5) \qquad \rho_1 = \Delta_2(x) \text{ if } x \in \mathsf{dom}(\Delta_2) \text{ otherwise } X \text{ for fresh } X \\ \rho_2 = \Delta_3(x) \text{ if } y \in \mathsf{dom}(\Delta_3) \text{ otherwise } Y \text{ for fresh } Y \end{array}}{\mathbf{case}\, L\, \{\mathbf{inl}\, x \mapsto M, \mathbf{inr}\, y \mapsto N\} : \sigma_1 \triangleright \Delta \,;\, \mathscr{C}_1 \cup \ldots \cup \mathscr{C}_5 \cup \{\tau \doteq (\rho_1 + \rho_2), \sigma_1 \doteq \sigma_2\}}$$

Figure 4 Sum, Product, and Unit Types

2.3 Generalising Co-contextual Type Systems

Figure 5 shows our generalisation of the co-contextual system presented by Erdweg, Bračevac, Kuci, Krebs, and Mezini [8]. The type system is now parameterised by *three* operations:

- $\Delta_1 \bowtie \Delta_2 = \Delta ; \mathscr{C}$ describes the *sequential merge* of environments Δ_1 and Δ_2, resulting in environment Δ and constraints $\mathscr{C}$. This is used when merging two environments obtained from sequentially-evaluated subterms (e.g., $M\, N$).
- $\Delta_1 \asymp \Delta_2 = \Delta ; \mathscr{C}$ describes the *branching merge* of environments Δ_1 and Δ_2, resulting in environment Δ and constraints $\mathscr{C}$. This is used when merging two environments obtained from typing subterms that use branching control flow (e.g., in constructs such as $\mathbf{case}\, L\, \{\mathbf{inl}\, x \mapsto M ; \mathbf{inr}\, y \mapsto N\}$)
- $\mathsf{check}(x, \Delta) = \tau ; \mathscr{C}$ describes the *check function*, which is used when ensuring that the inferred type of a variable is consistent with its type at a binder. The check function looks up the type of variable x in environment Δ, producing type τ and constraints $\mathscr{C}$. The constraint set $\mathscr{C}$ will be empty for the four main substructural systems, but is useful when discussing *mixed linearity* (Section 3).

The main differences between the system in Figure 5 and the system presented by Erdweg, Bračevac, Kuci, Krebs, and Mezini [8] are that the *sequential merge* is used whenever we want to combine environments derived from typing subterms that will be evaluated in sequential evaluation frames (e.g., TC-App and TC-Pair), and the *branching merge* is used for combining environments derived from subterms used in branching control flow (e.g., TC-Case). Furthermore, the check function is used whenever a typing rule needs to check the type of a variable at its binding site, and the approach avoids the more ad-hoc checks done in TE-Abs and TE-Case.

Generalised Co-contextual Typing Rules $\boxed{M : \tau \triangleright \Delta \,;\, \mathscr{C}}$

$$\textsc{TC-Const}\ \frac{c \text{ has base type } C}{c : C \triangleright \cdot \,;\, \emptyset} \qquad \textsc{TC-Abs}\ \frac{M : \tau \triangleright \Delta \,;\, \mathscr{C}_1 \qquad \mathsf{check}(x, \Delta) = \sigma;\ \mathscr{C}_2}{\lambda x.M : \sigma \to \tau \triangleright (\Delta - x) \,;\, \mathscr{C}_1 \cup \mathscr{C}_2} \qquad \textsc{TC-Inl}\ \frac{X \text{ is fresh} \qquad M : \tau \triangleright \Delta \,;\, \mathscr{C}}{\mathbf{inl}\, M : (\tau + X) \triangleright \Delta \,;\, \mathscr{C}}$$

$$\textsc{TC-App}\ \frac{M : \tau \triangleright \Delta_1 \,;\, \mathscr{C}_1 \qquad N : \sigma \triangleright \Delta_2 \,;\, \mathscr{C}_2 \qquad \Delta_1 \bowtie \Delta_2 = \Delta;\ \mathscr{C}_3 \qquad X \text{ is fresh}}{M\,N : X \triangleright \Delta \,;\, \mathscr{C}_1 \cup \mathscr{C}_2 \cup \mathscr{C}_3 \cup \{\tau \doteq \sigma \to X\}} \qquad \textsc{TC-Pair}\ \frac{M : \tau \triangleright \Delta_1 \,;\, \mathscr{C}_1 \qquad N : \sigma \triangleright \Delta_2 \,;\, \mathscr{C}_2 \qquad \Delta_1 \bowtie \Delta_2 = \Delta;\ \mathscr{C}_3}{(M, N) : (\tau \times \sigma) \triangleright \Delta \,;\, \mathscr{C}_1 \cup \mathscr{C}_2 \cup \mathscr{C}_3} \qquad \textsc{TC-Var}\ \frac{X \text{ is fresh}}{x : X \triangleright x : X \,;\, \emptyset}$$

$$\textsc{TC-Op}\ \frac{\mathsf{ty}(\odot) = A_1 \to A_2 \to B \qquad M : \tau \triangleright \Delta_1 \,;\, \mathscr{C}_1 \qquad N : \sigma \triangleright \Delta_2 \,;\, \mathscr{C}_2 \qquad \Delta_1 \bowtie \Delta_2 = \Delta;\ \mathscr{C}_3}{M \odot N : B \triangleright \Delta \,;\, \mathscr{C}_1 \cup \mathscr{C}_2 \cup \mathscr{C}_3 \cup \{\tau \doteq A_1, \sigma \doteq A_2\}} \qquad \textsc{TC-Seq}\ \frac{M : \tau \triangleright \Delta_1 \,;\, \mathscr{C}_1 \qquad N : \sigma \triangleright \Delta_2 \,;\, \mathscr{C}_2 \qquad \Delta_1 \bowtie \Delta_2 = \Delta;\ \mathscr{C}_3}{M; N : \sigma \triangleright \Delta \,;\, \mathscr{C}_1 \cup \mathscr{C}_2 \cup \mathscr{C}_3 \cup \{\tau \doteq \mathsf{Unit}\}}$$

$$\textsc{TC-LetPair}\ \frac{M : \tau \triangleright \Delta_1 \,;\, \mathscr{C}_1 \qquad N : \sigma \triangleright \Delta_2 \,;\, \mathscr{C}_2 \qquad \Delta_1 \bowtie (\Delta_2 - x - y) = \Delta;\ \mathscr{C}_3 \qquad \mathsf{check}(x, \Delta_2) = \rho_1;\ \mathscr{C}_4 \qquad \mathsf{check}(y, \Delta_2) = \rho_2;\ \mathscr{C}_5}{\mathbf{let}\,(x, y) = M\ \mathbf{in}\, N : \sigma \triangleright \Delta \,;\, \mathscr{C}_1 \cup \ldots \cup \mathscr{C}_5 \cup \{\tau \doteq (\rho_1 \times \rho_2)\}} \qquad \textsc{TC-Inr}\ \frac{X \text{ is fresh} \qquad M : \tau \triangleright \Delta \,;\, \mathscr{C}}{\mathbf{inr}\, M : (X + \tau) \triangleright \Delta \,;\, \mathscr{C}}$$

$$\textsc{TC-Case}\ \frac{L : \tau \triangleright \Delta_1 \,;\, \mathscr{C}_1 \qquad M : \sigma_1 \triangleright \Delta_2 \,;\, \mathscr{C}_2 \qquad N : \sigma_2 \triangleright \Delta_3 \,;\, \mathscr{C}_3 \qquad (\Delta_2 - x) \asymp (\Delta_3 - y) = \Delta_4;\ \mathscr{C}_4 \qquad \Delta_1 \bowtie \Delta_4 = \Delta;\ \mathscr{C}_5 \qquad \mathsf{check}(\Delta_2, x) = \rho_1;\ \mathscr{C}_6 \qquad \mathsf{check}(\Delta_3, y) = \rho_2;\ \mathscr{C}_7}{\mathbf{case}\ L\ \{\mathbf{inl}\, x \mapsto M, \mathbf{inr}\, y \mapsto N\} : \sigma_1 \triangleright \Delta \,;\, \mathscr{C}_1 \cup \ldots \cup \mathscr{C}_7 \cup \{\tau \doteq (\rho_1 + \rho_2), \sigma_1 \doteq \sigma_2\}}$$

$$\textsc{TC-Unit}\ \frac{}{() : \mathsf{Unit} \triangleright \cdot \,;\, \emptyset}$$

Figure 5 Generalised co-contextual typing rules (parametric in $\bowtie$, $\asymp$, and $\mathsf{check}(-,-)$)

2.4 Instantiating the Parameters

Now that we have introduced the generalised co-contextual system, we can instantiate the parameters to allow us to typecheck using different structural modes. In each instantiation of the parameters we give two representative declarative rules for the given structural mode: the variable rule (which demonstrates constraints on variable use), and the the case elimination rule (which demonstrates sequential control flow, branching control flow, and variable binding). A co-contextual typing judgement without a structural annotation is considered to hold for all the structural modes.

2.4.1 Unrestricted

Figure 6 shows the declarative rules and co-contextual parameters used to implement an unrestricted co-contextual type system, which allows each variable to be used zero or many times. Concretely, the declarative variable rule allows a variable to be typed even if other variables are present in the type environment, and (read bottom-up), the **case**-elimination rule does not require any variables to be split when typing subterms.

The sequential merge operator (similar to the original merge function) allows two environments to be merged if overlapping variables have the same types; this

Unrestricted

Representative Declarative Rules $\boxed{\Gamma \vdash_{\mathsf{unr}} M : A}$

$$\frac{x : A \in \Gamma}{\Gamma \vdash_{\mathsf{unr}} x : A} \qquad \frac{\Gamma \vdash_{\mathsf{unr}} L : (A_1 + A_2) \qquad \Gamma, x : A_1 \vdash_{\mathsf{unr}} M : B \qquad \Gamma, y : A_2 \vdash_{\mathsf{unr}} N : B}{\Gamma \vdash_{\mathsf{unr}} \mathbf{case}\ L\ \{\mathbf{inl}\ x \mapsto M, \mathbf{inr}\ y \mapsto N\} : B}$$

Sequential Merge $\boxed{\Delta_1 \bowtie_{\mathsf{unr}} \Delta_2 = \Delta;\ \mathscr{C}}$

$$\frac{}{\cdot \bowtie_{\mathsf{unr}} \cdot = \cdot;\ \emptyset} \qquad \frac{\Delta_1 \bowtie_{\mathsf{unr}} \Delta_2 = \Delta;\ \mathscr{C} \qquad x \notin \mathsf{dom}(\Delta_2)}{(\Delta_1, x : \tau) \bowtie_{\mathsf{unr}} \Delta_2 = (\Delta, x : \tau);\ \mathscr{C}}$$

$$\frac{\Delta_1 \bowtie_{\mathsf{unr}} \Delta_2 = \Delta;\ \mathscr{C} \qquad x \notin \mathsf{dom}(\Delta_1)}{\Delta_1 \bowtie_{\mathsf{unr}} (\Delta_2, x : \tau) = (\Delta, x : \tau);\ \mathscr{C}} \qquad \frac{\Delta_1 \bowtie_{\mathsf{unr}} \Delta_2 = \Delta;\ \mathscr{C}}{(\Delta_1, x : \tau) \bowtie_{\mathsf{unr}} (\Delta_2, x : \sigma) = (\Delta, x : \tau);\ \mathscr{C} \cup \{\tau \doteq \sigma\}}$$

Branching Merge $\boxed{\Delta_1 \asymp_{\mathsf{unr}} \Delta_2 = \Delta;\ \mathscr{C}}$

$$\Delta_1 \asymp_{\mathsf{unr}} \Delta_2 \triangleq \Delta_1 \bowtie_{\mathsf{unr}} \Delta_2$$

Check $\boxed{\mathsf{check}_{\mathsf{unr}}(x, \Delta) = \tau,\ \mathscr{C}}$

$$\mathsf{check}_{\mathsf{unr}}(x, \Delta) = \begin{cases} (\Delta(x), \emptyset) & \text{if } x \in \mathsf{dom}(\Delta) \\ (X, \emptyset) & \text{otherwise, with } X \text{ fresh} \end{cases}$$

Figure 6 Parameters for Unrestricted Co-contextual Type Systems

is ensured by generating equality constraints. Since there are no constraints on the number of variable usages, sequential and branching merges coincide. Finally, $\mathsf{check}(x, \Delta)$ returns $\Delta(x)$ if x is contained in Δ, and a fresh type variable otherwise.

2.4.2 Linear

Figure 7 shows the declarative rules and co-contextual parameters for implementing a linear co-contextual type system, where each variable must be used precisely once.

The *linear sequential merge* operator, unlike unrestricted, *does not* allow the merging of two environments with any overlapping variables. This ensures that a variable cannot be used twice. The *linear branching merge* operator has two rules: one that allows the combination of two empty environments, and one that allows the combination of two environments containing variable x. In the latter case, the rule generates a constraint ensuring that the variable has an equal type in each environment. Finally, the *linear* `check` *function* becomes partial: the function is undefined whenever looking up a variable that is not present in the environment. The `check` function is used at every binding site, and thus ensures that every variable is used at least once.

2.4.3 Affine

Affine type systems restrict contraction (disallowing multiple variable uses) but not weakening (allowing a variable to remain unused). Figure 8 (top) shows the declarative rules and co-contextual parameters. The declarative rules allow an arbitrary type environment in the variable rule, but split contexts when typing subterms.

We instantiate the co-contextual system using previous definitions: we use the linear sequential merge operator to rule out multiple variable uses, and the unrestricted

Linear

Representative Declarative Rules $\boxed{\Gamma \vdash_{\mathsf{lin}} M : A}$

$$\frac{}{x : A \vdash_{\mathsf{lin}} x : A} \qquad \frac{\Gamma_1 \vdash_{\mathsf{lin}} L : (A_1 + A_2) \qquad \Gamma_2, x : A_1 \vdash_{\mathsf{lin}} M : B \qquad \Gamma_2, y : A_2 \vdash_{\mathsf{lin}} N : B}{\Gamma_1, \Gamma_2 \vdash_{\mathsf{lin}} \mathbf{case}\ L\ \{\mathbf{inl}\ x \mapsto M, \mathbf{inr}\ y \mapsto N\} : B}$$

Sequential Merge $\boxed{\Delta_1 \bowtie_{\mathsf{lin}} \Delta_2 = \Delta;\ \mathscr{C}}$

$$\frac{}{\cdot \bowtie_{\mathsf{lin}} \cdot = \cdot;\ \emptyset} \qquad \frac{\Delta_1 \bowtie_{\mathsf{lin}} \Delta_2 = \Delta;\ \mathscr{C} \qquad x \notin \mathsf{dom}(\Delta_2)}{(\Delta_1, x : \tau) \bowtie_{\mathsf{lin}} \Delta_2 = (\Delta, x : \tau);\ \mathscr{C}} \qquad \frac{\Delta_1 \bowtie_{\mathsf{lin}} \Delta_2 = \Delta;\ \mathscr{C} \qquad x \notin \mathsf{dom}(\Delta_1)}{\Delta_1 \bowtie_{\mathsf{lin}} (\Delta_2, x : \tau) = (\Delta, x : \tau);\ \mathscr{C}}$$

Branching Merge $\boxed{\Delta_1 \asymp_{\mathsf{lin}} \Delta_2 = \Delta;\ \mathscr{C}}$

$$\frac{}{\cdot \asymp_{\mathsf{lin}} \cdot = \cdot;\ \emptyset} \qquad \frac{\Delta_1 \asymp_{\mathsf{lin}} \Delta_2 = \Delta;\ \mathscr{C}}{(\Delta_1, x : \tau) \asymp_{\mathsf{lin}} (\Delta_2, x : \sigma) = (\Delta, x : \tau);\ \mathscr{C} \cup \{\tau \doteq \sigma\}}$$

Check $\boxed{\mathsf{check}_{\mathsf{lin}}(x, \Delta) = \tau;\ \mathscr{C}}$

$$\mathsf{check}_{\mathsf{lin}}(x, \Delta) = \begin{cases} (\Delta(x), \emptyset) & \text{if } x \in \mathsf{dom}(\Delta) \\ \text{undefined} & \text{otherwise} \end{cases}$$

Figure 7 Parameters for Linear Co-contextual Type Systems

Affine

$$\frac{x : A \in \Gamma}{\Gamma \vdash_{\mathsf{aff}} x : A} \qquad \frac{\Gamma_1 \vdash_{\mathsf{aff}} L : (A_1 + A_2) \qquad \Gamma_2, x : A_1 \vdash_{\mathsf{aff}} M : B \qquad \Gamma_2, y : A_2 \vdash_{\mathsf{aff}} N : B}{\Gamma_1, \Gamma_2 \vdash_{\mathsf{aff}} \mathbf{case}\ L\ \{\mathbf{inl}\ x \mapsto M, \mathbf{inr}\ y \mapsto N\} : B}$$

$$\begin{array}{rcl} \Delta_1 \bowtie_{\mathsf{aff}} \Delta_2 & \triangleq & \Delta_1 \bowtie_{\mathsf{lin}} \Delta_2 \\ \Delta_1 \asymp_{\mathsf{aff}} \Delta_2 & \triangleq & \Delta_1 \asymp_{\mathsf{unr}} \Delta_2 \\ \mathsf{check}_{\mathsf{aff}}(x, \Delta) & \triangleq & \mathsf{check}_{\mathsf{unr}}(x, \Delta) \end{array}$$

Relevant

$$\frac{}{x : A \vdash_{\mathsf{rel}} x : A} \qquad \frac{\Gamma_1 \vdash_{\mathsf{rel}} L : (A_1 + A_2) \qquad \Gamma_2, x : A_1 \vdash_{\mathsf{rel}} M : B \qquad \Gamma_2, y : A_2 \vdash_{\mathsf{rel}} N : B}{\Gamma_1 \oplus \Gamma_2 \vdash_{\mathsf{rel}} \mathbf{case}\ L\ \{\mathbf{inl}\ x \mapsto M, \mathbf{inr}\ y \mapsto N\} : B}$$

$$\begin{array}{rcl} \Delta_1 \bowtie_{\mathsf{rel}} \Delta_2 & \triangleq & \Delta_1 \bowtie_{\mathsf{unr}} \Delta_2 \\ \Delta_1 \asymp_{\mathsf{rel}} \Delta_2 & \triangleq & \Delta_1 \asymp_{\mathsf{lin}} \Delta_2 \\ \mathsf{check}_{\mathsf{rel}}(x, \Delta) & \triangleq & \mathsf{check}_{\mathsf{lin}}(x, \Delta) \end{array}$$

Figure 8 Representative Rules and Parameters for Affine and Relevant Type Systems

branching merge operator to permit a variable to be used in one branch but not another. We use the unrestricted check to allow a variable to be unused.

2.4.4 Relevant

Relevant type systems restrict weakening but not contraction, ensuring that a variable must be used *at least once*. Figure 8 (bottom) shows the declarative rules and co-contextual parameters used to implement a relevant co-contextual type system.

The declarative typing rules we make use of an auxiliary environment combination judgement $\Gamma_1 \oplus \Gamma_2 = \Gamma$ that joins two type environments Γ_1 and Γ_2 into an environment Γ, where each variable in Γ may occur in Γ_1, Γ_2, or both.

Definition 1 (Unrestricted environment combination $\Gamma = \Gamma_1 \oplus \Gamma_2$)**.**

$$\frac{}{\cdot = \cdot \oplus \cdot} \quad \frac{\Gamma = \Gamma_1 \oplus \Gamma_2}{\Gamma, x : A = \Gamma_1, x : A \oplus \Gamma_2, x : A} \quad \frac{\Gamma = \Gamma_1 \oplus \Gamma_2 \qquad x \notin \mathit{dom}(\Gamma_2)}{\Gamma, x : A = \Gamma_1, x : A \oplus \Gamma_2} \quad \frac{\Gamma = \Gamma_1 \oplus \Gamma_2 \qquad x \notin \mathit{dom}(\Gamma_1)}{\Gamma, x : A = \Gamma_1 \oplus \Gamma_2, x : A}$$

We define the sequential merge as the unrestricted merge in order to allow variables to be used more than once, and use the linear branching merge and check function to ensure that variables are used at least once. The sequential merge operator represents control over contraction, while the combined presence of variable check and branching merge allows control over weakening.

2.5 Metatheory

We can now show that the co-contextual versions of our systems are sound and complete with respect to their declarative equivalents. Soundness ensures that everything typable in the co-contextual systems is also typable in the declarative systems, i.e., that the co-contextual systems do not accept ill-typed terms. Completeness ensures that everything typable in the declarative systems is typable in the algorithmic systems, i.e., that the co-contextual systems do not reject well-typed terms. We elide the proofs in the main body of the paper; they can be found in the extended version [36].

Since our algorithmic systems make use of constraints, we firstly define the notion of a type substitution that maps type variables to types.

Definition 2 (Substitution)**.** *A* substitution Ξ *is a mapping* $\{X_i \mapsto A_i\}_{i \in I}$ *from type variables to closed types. We* apply *a substitution* $\Xi = \{X_i \mapsto A_i\}_{i \in I}$ *to a type* τ*, written* $\Xi(\tau)$ *by replacing every free occurrence of each* X_i *in* τ *with* A_i*, and extend application of a substitution to constraints, constraint sets, and type environments.*

Because a substitution only substitutes in *closed* types, it does not matter whether we apply substitutions sequentially or simultaneously.

A *solution* for a given constraint set is a substitution mapping type variables to types such that all equality constraints hold.

Definition 3 (Solution)**.** *A substitution* Ξ *is a* solution *for a constraint set* $\mathscr{C} = \{\tau_i \doteq \sigma_i\}_{i \in I}$ *if* $\Xi(\tau_i) = \Xi(\sigma_i)$ *for all* $i \in I$*.*

Let us write $\mathsf{tv}(\tau)$ to denote the set of all type variables contained in a type τ. A *covering solution* for a co-contextual typing derivation is a solution that assigns a type to every type variable in the returned type, environment, and constraint set in the given derivation.

Definition 4 (Covering solution)**.**

Typing *A substitution* Ξ *is a* covering solution for a derivation $M : \tau \triangleright_m \Delta \,;\, \mathscr{C}$ *if* Ξ *is a solution for* $\mathscr{C}$ *and* $\mathsf{tv}(\tau) \cup \mathsf{tv}(\Delta) \subseteq \mathit{dom}(\Xi)$*.*

Merge Operations *A substitution* Ξ *is a* covering solution for a derivation $\Delta_1 \bowtie_m \Delta_2 = \Delta \,;\, \mathscr{C}$ *or* $\Delta_1 \asymp_m \Delta_2 = \Delta \,;\, \mathscr{C}$ *if* Ξ *is a solution for* $\mathscr{C}$ *and* $\mathsf{tv}(\Delta) \subseteq \mathit{dom}(\Xi)$*.*

Check *A substitution* Ξ *is a* covering solution for a derivation $\mathit{check}(\Delta, x) = \tau \,;\, \mathscr{C}$ *if* Ξ *is a solution for* $\mathscr{C}$ *and* $\mathsf{tv}(\tau) \cup \mathsf{tv}(\Delta) \subseteq \mathit{dom}(\Xi)$*.*

The following useful result shows that given a co-contextual judgement $M : \tau \triangleright \Delta ; \mathscr{C}$, the variables in an output context are precisely the free variables of M. Recall that judgements without a mode annotation hold for all structural modes.

Lemma 1 (Co-contextual variables)**.** *If $M : \tau \triangleright \Delta ; \mathscr{C}$, then $\mathit{dom}(\Delta) = \mathit{fv}(M)$.*

2.5.1 Soundness

To prove soundness, the main auxiliary results we need to show soundness of the merge operators with respect to their declarative counterparts.

If the linear sequential merge of two environments $\Delta_1 \bowtie_{\mathsf{lin}} \Delta_2$ is defined, producing merged environment Δ, and we have a covering solution for the derivation, then the solution applied to the Δ environment is equal to the disjoint combination of the solution applied on Δ_1 and Δ_2. On the other hand, since unrestricted environment combination allows duplicated uses of a variable, the disjoint combination of the two environments will not necessarily be defined, and so we must use the unrestricted environment combination operator.

Lemma 2 (Soundness (Sequential merge))**.**

Linear *Given a covering solution Ξ of $\Delta_1 \bowtie_{\mathit{lin}} \Delta_2 = \Delta$; $\mathscr{C}$, it holds that $\Xi(\Delta) = \Xi(\Delta_1)$, $\Xi(\Delta_2)$.*

Unrestricted *Given a covering solution Ξ of $\Delta_1 \bowtie_{\mathit{unr}} \Delta_2 = \Delta$; $\mathscr{C}$, it holds that $\Xi(\Delta) = \Xi(\Delta_1) \oplus \Xi(\Delta_2)$.*

The linear branching merge operator ensures that variables are used consistently across branches, so applying a covering solution will yield equal environments. In contrast, the unrestricted branching merge operator permits usage in either of the branches, or both. Again we address this by combining the resulting environments using the unrestricted environment combination operator.

Lemma 3 (Soundness (Branching merge))**.**

Linear *Given a covering solution Ξ of $\Delta_1 \curlywedge_{\mathit{lin}} \Delta_2 = \Delta$; $\mathscr{C}$, it holds that $\Xi(\Delta) = \Xi(\Delta_1) = \Xi(\Delta_2)$.*

Unrestricted *Given a covering solution Ξ of $\Delta_1 \curlywedge_{\mathit{unr}} \Delta_2 = \Delta$; $\mathscr{C}$, there exist Γ_1 and Γ_2 such that $\Xi(\Delta_1) = \Gamma_1$, $\Xi(\Delta_2) = \Gamma_2$, and $\Xi(\Delta_1) \oplus \Xi(\Delta_2) = \Gamma_1 \oplus \Gamma_2$.*

Finally, we can show that our family of co-contextual systems are sound with respect to their declarative counterparts.

Theorem 1 (Algorithmic Soundness)**.** *For $m \in \{\mathsf{lin}, \mathsf{unr}, \mathsf{aff}, \mathsf{rel}\}$, given a covering solution Ξ of $M : \tau \triangleright_m \Delta ; \mathscr{C}$, it holds that $\Xi(\Delta) \vdash_m M : \Xi(\tau)$.*

2.5.2 Completeness

Next, we show that each system in our family of co-contextual type systems is *complete* with respect to its declarative counterpart: everything typable in each declarative system is typable in its corresponding algorithmic system.

Again, we need some lemmas to reason about context merges. In the linear case, given a disjoint split of contexts Γ_1, Γ_2 (the components of which are some algorithmic

contexts under a covering solution), the union of the solutions produces a new covering solution for the sequential merge of the algorithmic contexts.

In the unrestricted case, the joined context can not be partitioned with a split, and the declarative system may contain more unrestricted free variables compared to the co-contextual system, which is minimal w.r.t. free variables (Lemma 1). We therefore require that $\Gamma_1 \subseteq \Gamma \wedge \Gamma_2 \subseteq \Gamma$. This provides us with a solution for a combined environment that is a subset of the original declarative context.

Lemma 4 (Completeness (Sequential merge))**.**

Linear *If* $\Gamma = \Gamma_1, \Gamma_2$ *with* $\Xi_1(\Delta_1) = \Gamma_1 \wedge \Xi_2(\Delta_2) = \Gamma_2$ *for some* $\Xi_1, \Xi_2, \Delta_1, \Delta_2$*, with* $\mathit{dom}(\Xi_1) \cap \mathit{dom}(\Xi_2) = \emptyset$*, then* $\Xi = \Xi_1 \cup \Xi_2$ *is a covering solution of the derivation* $\Delta_1 \bowtie_{\mathit{lin}} \Delta_2 = \Delta$*;* $\mathscr{C}$ *such that* $\Xi(\Delta) = \Gamma$*.*

Unrestricted *If* $\Xi_1(\Delta_1) \subseteq \Gamma \wedge \Xi_2(\Delta_2) \subseteq \Gamma$ *for some* $\Xi_1, \Xi_2, \Delta_1, \Delta_2$*, with* $\mathit{dom}(\Xi_1) \cap \mathit{dom}(\Xi_2) = \emptyset$*, then* $\Xi = \Xi_1 \cup \Xi_2$ *is a covering solution of* $\Delta_1 \bowtie_{\mathit{unr}} \Delta_2 = \Delta$*;* $\mathscr{C}$*, such that* $\Xi(\Delta) \subseteq \Gamma$*.*

A branching merge in the linear system occurs in the algorithmic system when we are typing two subterms that are typable under the same declarative environment. In the unrestricted case, we cannot rely on the unrestricted contexts being equal to their algorithmic counterparts under covering solution. We show the algorithmic output context is a subset of the declarative branch context.

Lemma 5 (Completeness (Branching merge))**.**

Linear *If* $\Xi_1(\Delta_1) = \Gamma \wedge \Xi_2(\Delta_2) = \Gamma$ *for some* $\Xi_1, \Xi_2, \Delta_1, \Delta_2$*, with* $\mathit{dom}(\Xi_1) \cap \mathit{dom}(\Xi_2) = \emptyset$*, then* $\Xi = \Xi_1 \cup \Xi_2$ *is a solution of* $\Delta_1 \asymp_{\mathit{lin}} \Delta_2 = \Delta$*;* $\mathscr{C}$*, such that* $\Xi(\Delta) = \Gamma$*.*

Unrestricted *If* $\Xi_1(\Delta_1) \subseteq \Gamma \wedge \Xi_2(\Delta_2) \subseteq \Gamma$ *for some* $\Xi_1, \Xi_2, \Delta_1, \Delta_2$*, with* $\mathit{dom}(\Xi_1) \cap \mathit{dom}(\Xi_2) = \emptyset$*, then* $\Xi = \Xi_1 \cup \Xi_2$ *is a covering solution of* $\Delta_1 \asymp_{\mathit{unr}} \Delta_2 = \Delta$*;* $\mathscr{C}$*, such that* $\Xi(\Delta) \subseteq \Gamma$*.*

With these intermediate results defined, we are finally in a position to state algorithmic completeness: if a term is typable in the declarative system for a given structural mode, then it is also typable in the corresponding co-contextual algorithmic system.

Theorem 2 (Algorithmic Completeness)**.** *For* $m \in \{\mathit{lin}, \mathit{unr}, \mathit{aff}, \mathit{rel}\}$*, given* $\Gamma \vdash_m M : A$*, it holds that there exists a covering solution* Ξ *for* $M : \tau \triangleright_m \Delta \,;\, \mathscr{C}$*, such that* $\Xi(\Delta) \subseteq \Gamma \wedge \Xi(\tau) = A$*.*

3 Mixed Linearity and Session Types

Mainstream programming languages use a mixture of resourceful and non-resourceful data. In fact, a purely linear language without usage of exponentials [11] is not capable of typing terms like $\lambda x.x + x$, due to its double usage of the variable x, which severely impairs expressiveness. In this section we show how to extend our co-contextual systems with a mixture of linear and unrestricted data.

To incorporate unrestricted data, we begin by describing a new predicate $\mathrm{un}(A)$ that classifies types as unrestricted.

Mixed

Check function

$$\mathsf{check}_{\mathsf{mix}}(x,\Delta) = \begin{cases} \Delta(x);\emptyset & \text{if } x \in \mathsf{dom}(\Delta) \\ X;\{\mathfrak{U}(X)\} & \text{otherwise, with fresh } X \end{cases}$$

Sequential merge function $\boxed{\Delta_1 \bowtie_{\mathsf{mix}} \Delta_2 = \Delta;\ \mathscr{C}}$

$$\textsc{Base}\ \frac{}{\cdot \bowtie_{\mathsf{mix}} \cdot = \cdot;\ \emptyset} \qquad \textsc{Match}\ \frac{\Delta_1 \bowtie_{\mathsf{mix}} \Delta_2 = \Delta;\ \mathscr{C}}{\Delta_1, x:\tau_1 \bowtie_{\mathsf{mix}} \Delta_2, x:\tau_2 = \Delta, x:\tau_1;\ \mathscr{C} \cup \{\tau_1 \doteq \tau_2, \mathfrak{U}(\tau_1)\}}$$

$$\textsc{Left}\ \frac{\Delta_1 \bowtie_{\mathsf{mix}} \Delta_2 = \Delta;\ \mathscr{C} \qquad x \notin \mathsf{dom}(\Delta_2)}{\Delta_1, x:\tau \bowtie_{\mathsf{mix}} \Delta_2 = \Delta, x:\tau;\ \mathscr{C}} \qquad \textsc{Right}\ \frac{\Delta_1 \bowtie_{\mathsf{mix}} \Delta_2 = \Delta;\ \mathscr{C} \qquad x \notin \mathsf{dom}(\Delta_1)}{\Delta_1 \bowtie_{\mathsf{mix}} \Delta_2, x:\tau = \Delta, x:\tau;\ \mathscr{C}}$$

Branching merge function $\boxed{\Delta_1 \asymp_{\mathsf{mix}} \Delta_2 = \Delta;\ \mathscr{C}}$

$$\textsc{Base}\ \frac{}{\cdot \asymp_{\mathsf{mix}} \cdot = \cdot;\ \emptyset} \qquad \textsc{Match}\ \frac{\Delta_1 \asymp_{\mathsf{mix}} \Delta_2 = \Delta;\ \mathscr{C}}{\Delta_1, x:\tau_1 \asymp_{\mathsf{mix}} \Delta_2, x:\tau_2 = \Delta, x:\tau_1;\ \mathscr{C} \cup \{\tau_1 \doteq \tau_2\}}$$

$$\textsc{Left}\ \frac{\Delta_1 \asymp_{\mathsf{mix}} \Delta_2 = \Delta;\ \mathscr{C} \qquad x \notin \mathsf{dom}(\Delta_2)}{\Delta_1, x:\tau \asymp_{\mathsf{mix}} \Delta_2 = \Delta, x:\tau;\ \mathscr{C} \cup \{\mathfrak{U}(\tau)\}} \qquad \textsc{Right}\ \frac{\Delta_1 \asymp_{\mathsf{mix}} \Delta_2 = \Delta;\ \mathscr{C} \qquad x \notin \mathsf{dom}(\Delta_1)}{\Delta_1 \asymp_{\mathsf{mix}} \Delta_2, x:\tau = \Delta, x:\tau;\ \mathscr{C} \cup \{\mathfrak{U}(\tau)\}}$$

Figure 9 Parameters for Mixed Co-contextual Type Systems

Definition 5 (Unrestricted type predicate $\mathsf{un}(A)$)**.**

$$\frac{}{\mathsf{un}(C)} \qquad \frac{}{\mathsf{un}(A \to B)} \qquad \frac{\mathsf{un}(A) \qquad \mathsf{un}(B) \qquad \odot \in \{\times, +\}}{\mathsf{un}(A \odot B)} \qquad \frac{\forall x:A \in \Gamma\,.\,\mathsf{un}(A)}{\mathsf{un}(\Gamma)}$$

Consider typing the term $(\lambda x^{\mathsf{Int}}.(\lambda y^{\mathsf{Int}}.x)\,x)\,1$ in the co-contextual setting. Although we might usually expect integers to be unrestricted, in the inner application of $\lambda y.x$ to x, we note x is used twice, on both the argument and function side of the application. In contrast, our current rule says that the component contexts of an application ought to be disjoint, i.e. (Γ_1, Γ_2). To extend the declarative system to allow integers to be unrestricted, we replace the disjoint context concatenation $(,)$ with mixed environment combination $(+)$, following Gay and Vasconcelos [10].

Definition 6 (Mixed environment combination $\Gamma_1 + \Gamma_2 = \Gamma$)**.**

$$\frac{}{\cdot + \cdot = \cdot} \qquad \frac{\Gamma_1 + \Gamma_2 = \Gamma \qquad \mathsf{un}(A)}{\Gamma_1, x:A + \Gamma_2, x:A = \Gamma, x:A} \qquad \frac{\Gamma_1 + \Gamma_2 = \Gamma \qquad x \notin \mathit{dom}(\Gamma_2)}{\Gamma_1, x:A + \Gamma_2 = \Gamma, x:A} \qquad \frac{\Gamma_1 + \Gamma_2 = \Gamma \qquad x \notin \mathit{dom}(\Gamma_1)}{\Gamma_1 + \Gamma_2, x:A = \Gamma, x:A}$$

We give a partial definition of the mixed declarative type system below, showcasing T-Var and T-Case. The full version is provided in the extended version [36].

$$\textsc{T-Var}\ \frac{\mathsf{un}(\Gamma)}{\Gamma, x:A \vdash_{\mathsf{mix}} x:A} \qquad \textsc{T-Case}\ \frac{\Gamma_1 \vdash_{\mathsf{mix}} L:(A_1 + A_2) \qquad \Gamma_2, x:A_1 \vdash_{\mathsf{mix}} M:B \qquad \Gamma_2, y:A_2 \vdash_{\mathsf{mix}} N:B}{\Gamma_1 + \Gamma_2 \vdash_{\mathsf{mix}} \mathbf{case}\ L\ \{\mathbf{inl}\ x \mapsto M, \mathbf{inr}\ y \mapsto N\}:B}$$

Syntax

Closed types	A, B	$::=$	$C \mid S \mid A \to B \mid (A \times B) \mid (A + B)$
Closed session types	S	$::=$	$!A.S \mid ?A.S \mid \mathsf{end}_? \mid \mathsf{end}_!$
Additional terms	L, M, N	$::=$	$\cdots \mid \mathbf{send}\, M\, N \mid \mathbf{receive}\, M \mid \mathbf{fork}\, M \mid \mathbf{wait}\, M$

Declarative Session Typing Rules $\boxed{\Gamma \vdash_{\mathsf{mix}} M : A}$

$$\textsc{T-Send}\ \frac{\Gamma_1 \vdash_{\mathsf{mix}} M : A \qquad \Gamma_2 \vdash_{\mathsf{mix}} N : !A.S}{\Gamma_1 + \Gamma_2 \vdash_{\mathsf{mix}} \mathbf{send}\, M\, N : S} \qquad \textsc{T-Fork}\ \frac{\Gamma \vdash_{\mathsf{mix}} M : S \to \mathsf{end}_!}{\Gamma \vdash_{\mathsf{mix}} \mathbf{fork}\, M : \overline{S}}$$

$$\textsc{T-Wait}\ \frac{\Gamma \vdash_{\mathsf{mix}} M : \mathsf{end}_?}{\Gamma \vdash_{\mathsf{mix}} \mathbf{wait}\, M : \mathsf{Unit}} \qquad \textsc{T-Receive}\ \frac{\Gamma \vdash_{\mathsf{mix}} M : ?A.S}{\Gamma \vdash_{\mathsf{mix}} \mathbf{receive}\, M : (A \times S)}$$

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

$$\overline{!A.S} = ?A.\overline{S} \qquad \overline{?A.S} = !A.\overline{S} \qquad \overline{\mathsf{end}_!} = \mathsf{end}_? \qquad \overline{\mathsf{end}_?} = \mathsf{end}_!$$

Figure 10 Declarative typing rules for the mixed GV fragment

Figure 9 shows how to extend the co-contextual system to support unrestricted data. We introduce a new constraint $\mathfrak{U}(\tau)$ that requires that τ is unrestricted, and is introduced when a variable with type τ is used in a way that would violate linearity.

Specifically, the check meta-function (used at binding sites) generates a fresh type variable X and a constraint $\mathfrak{U}(\tau)$ if a bound variable has not been used; the Match case of the sequential merge generates an unrestrictedness constraint if a variable is used multiple times; and the Left and Right cases of the branching merge generate an unrestrictedness constraint if a variable has only been used in one branch.

3.1 The GV Language

Figure 10 shows how to extend the base language with GV's session typing constructs. The **send** $M\ N$ construct involves sending a term of type A over a session endpoint of type $!A.S$, returning an endpoint of type S. The **receive** M construct involves receiving from an endpoint of type $?A.S$, returning a pair of the received value (of type A) and the continuation (of type S). The **wait** M construct eliminates a terminated session of type $\mathsf{end}_?$, and **fork** M establishes a session by spawning M as a parallel thread and supplying it with a fresh endpoint, returning an endpoint with dual session type. GV does not include offer and select types since they can be encoded using sums [6].

3.2 Algorithmic GV

Figure 11 shows the co-contextual rules for implementing GV. We introduce a constraint $\mathscr{S}(\tau)$ that asserts that τ is a session type S, and a *syntactic duality* type $\tau^{\perp}$ to represent the dual of type τ.

Sending data In TC-Send, terms M and N represent the payload and session endpoint respectively. We recursively check both subterms. We instantiate a fresh type variable X and constrain the expected type of the session endpoint using the constraint $\{\sigma \doteq !\tau.X\}$, and indicate that the continuation is a session with $\mathscr{S}(X)$.

Syntax

Open types	τ, σ	$::=$	$C \mid X \mid (\tau \times \sigma) \mid (\tau + \sigma) \mid \tau \to \sigma$ $\mid\ !\tau.\sigma \mid ?\tau.\sigma \mid \mathsf{end}_? \mid \mathsf{end}_! \mid \tau^\perp$
Additional terms	M, N	$::=$	$\cdots \mid \mathbf{send}\, M\, N \mid \mathbf{receive}\, M \mid \mathbf{fork}\, M \mid \mathbf{wait}\, M$
Open type environments	Δ	$::=$	$\cdot \mid (\Delta, x : \tau)$

Algorithmic Session Typing Rules $\boxed{M : \tau \triangleright_{\mathsf{mix}} \Delta\,;\, \mathscr{C}}$

TC-Send
$$\frac{M : \tau \triangleright_{\mathsf{mix}} \Delta_1\,;\, \mathscr{C}_1 \qquad N : \sigma \triangleright_{\mathsf{mix}} \Delta_2\,;\, \mathscr{C}_2 \qquad \Delta_1 \bowtie_{\mathsf{mix}} \Delta_2 = \Delta;\ \mathscr{C}_3 \qquad X \text{ is fresh}}{\mathbf{send}\, M\, N : X \triangleright_{\mathsf{mix}} \Delta\,;\, \mathscr{C}_1 \cup \mathscr{C}_2 \cup \mathscr{C}_3 \cup \{\sigma \doteq\, !\tau.X, \mathscr{S}(X)\}}$$

TC-Wait
$$\frac{M : \tau \triangleright_{\mathsf{mix}} \Delta\,;\, \mathscr{C}}{\mathbf{wait}\, M : \mathsf{Unit} \triangleright_{\mathsf{mix}} \Delta\,;\, \mathscr{C} \cup \{\tau \doteq \mathsf{end}_?\}}$$

TC-Fork
$$\frac{M : \tau \triangleright_{\mathsf{mix}} \Delta\,;\, \mathscr{C} \qquad X \text{ is fresh}}{\mathbf{fork}\, M : X^\perp \triangleright_{\mathsf{mix}} \Delta\,;\, \mathscr{C} \cup \{\tau \doteq X \to \mathsf{end}_!, \mathscr{S}(X)\}}$$

TC-Receive
$$\frac{M : \tau \triangleright_{\mathsf{mix}} \Delta\,;\, \mathscr{C} \qquad X \text{ is fresh} \qquad Y \text{ is fresh}}{\mathbf{receive}\, M : (X \times Y) \triangleright_{\mathsf{mix}} \Delta\,;\, \mathscr{C} \cup \{\tau \doteq\, ?X.Y, \mathscr{S}(Y)\}}$$

Figure 11 Algorithmic typing rules for the mixed GV fragment

We replace the environment combination $\Gamma_1 + \Gamma_2$ with a mixed sequential merge $\Delta_1 \bowtie_{\mathsf{mix}} \Delta_2 = \Delta;\ \mathscr{C}_3$.

Receiving data When receiving data, T-Receive generates fresh type variables X and Y to represent the payload and continuation types respectively. The rule typechecks subterm M and constrains its type to $?X.Y$.

Forking sessions Rule TC-Fork generates a fresh type variable X to represent the session endpoint type. It constrains the type of the subterm to $X \to \mathsf{end}_!$, and returns the syntactically dual type $X^\perp$.

Waiting on sessions Finally, TC-Wait typechecks term M and constrains it to type $\mathsf{end}_?$.

3.3 Metatheory

We can now show that our extended co-contextual system with session types is once again sound and complete with respect to the declarative presentation of GV. Mixed algorithmic types introduce syntactically dual types $\tau^\perp$. We introduce $\mathsf{simp}(-)$ as a means of replacing closed algorithmic types with their semantic duals.

Definition 7 (Conversion of syntactic duality ($\mathsf{simp}(\tau)$))**.** *$simp(\tau)$ is a partial recursive function defined on a closed algorithmic type τ as follows:*

$$simp((!\sigma.\rho)^\perp) = ?\sigma.simp(\rho^\perp) \qquad simp((?\sigma.\rho)^\perp) = !\sigma.simp(\rho^\perp)$$
$$simp(end_!{}^\perp) = end_? \qquad simp(end_?{}^\perp) = end_!$$
$$simp((\tau \times \sigma)) = (simp(\tau) \times simp(\sigma)) \qquad simp((\tau + \sigma)) = (simp(\tau) + simp(\sigma))$$
$$simp(\tau \to \sigma) = simp(\tau) \to simp(\sigma)$$

Syntactic duals of non-session dual types, e.g., $Int^\perp$ and $X^\perp$, are undefined. Our metatheory does not require us to apply $\mathsf{simp}(-)$ to an open type.

Lemma 6. *If τ is closed, and $\mathsf{simp}(\tau^{\perp})$ is defined, then $\mathsf{simp}(\tau^{\perp}) = \overline{\tau}$.*

Applying $\mathsf{simp}(\tau)$ on a closed type has no effect.

Lemma 7. *For any closed type A, $\mathsf{simp}(A) = A$.*

We extend application of a substitution (Definition 2) to act on the underlying types of the new constraints $\mathfrak{U}(\tau)$ and $\mathscr{S}(\tau)$ (e.g., $\Xi(\mathfrak{U}(X)) = \mathfrak{U}(\Xi(X))$). We next generalise solutions to support equality up-to duality, allowing the resolution of constraints like $(!\mathsf{Bool}.\mathsf{end}_!)^{\perp} \doteq ?\mathsf{Bool}.\mathsf{end}_?$.

Definition 8 (Equality up to duality)**.** *For two algorithmic types τ_1, τ_2, equality up to duality $\tau_1 \cong \tau_2$ is the reflexive, symmetric, and transitive closure of the following rules:*

$$
\begin{array}{rcllrcl}
(!\sigma_1.\rho_1)^{\perp} & \cong & ?\sigma_2.\rho_2 & \text{if } \sigma_1 \cong \sigma_2 \wedge {\rho_1}^{\perp} \cong \rho_2 & \mathsf{end}_!{}^{\perp} & \cong & \mathsf{end}_? \\
(?\sigma_1.\rho_1)^{\perp} & \cong & !\sigma_2.\rho_2 & \text{if } \sigma_1 \cong \sigma_2 \wedge {\rho_1}^{\perp} \cong \rho_2 & \mathsf{end}_?{}^{\perp} & \cong & \mathsf{end}_! \\
?\sigma_1.\rho_1 & \cong & (!\sigma_2.\rho_2)^{\perp} & \text{if } \sigma_1 \cong \sigma_2 \wedge \rho_1 \cong {\rho_2}^{\perp} & \mathsf{end}_! & \cong & \mathsf{end}_?{}^{\perp} \\
!\sigma_1.\rho_1 & \cong & (?\sigma_2.\rho_2)^{\perp} & \text{if } \sigma_1 \cong \sigma_2 \wedge \rho_1 \cong {\rho_2}^{\perp} & \mathsf{end}_? & \cong & \mathsf{end}_!{}^{\perp}
\end{array}
$$

Furthermore, compound types (pair, product, function) are equal up to duality if they are equal up to duality in their components. Two session types (non-dual) are equal up to duality if their actions and continuations are equal up to duality. For all other τ_1, τ_2, $\tau_1 \cong \tau_2 \iff \tau_1 = \tau_2$.

We extend the definition of a solution to incorporate unrestrictedness constraints, session constraints, and equality up to duality.

Definition 9 (Solution (mixed))**.** *A closed substitution Ξ is a* solution *for a constraint set $\mathscr{C}$ if:*

- *For all equality constraints $\tau \doteq \sigma$ in $\mathscr{C}$, $\Xi(\tau) \cong \Xi(\sigma)$ holds.*
- *For all unrestrictedness constraints $\mathfrak{U}(\tau)$ in $\mathscr{C}$, $\mathrm{un}(\Xi(\tau))$ holds.*
- *For all session constraints $\mathscr{S}(\tau)$ in $\mathscr{C}$, $\Xi(\tau)$ is a session type S.*

Soundness All contexts now have an unrestricted part, which is denoted with the $\mathrm{un}(-)$ predicate in statements. The soundness theorem now applies $\mathsf{simp}(-)$ to derive semantically dual types from syntactic dual types.

Lemma 8 (Soundness (Mixed merges))**.**

Sequential *If $\Delta_1 \bowtie_{mix} \Delta_2 = \Delta$; $\mathscr{C}$ and Ξ is a covering solution for the derivation, then $\Xi(\Delta) = \Xi(\Delta_1) + \Xi(\Delta_2)$.*

Branching *If $\Delta_1 \asymp_{mix} \Delta_2 = \Delta$; $\mathscr{C}$ and Ξ is a covering solution for the derivation, then there exists a Γ such that $\Xi(\Delta) = \Xi(\Delta_1) + \Gamma = \Xi(\Delta_2) + \Gamma$.*

With these intermediate results in hand, we can show algorithmic soundness for the mixed-linearity system.

Theorem 3 (Algorithmic soundness (mix))**.** *Given a covering solution Ξ of $M : \tau \triangleright_{mix} \Delta \,;\, \mathscr{C}$, it holds that $\Xi(\Delta) \vdash_{mix} M{:}\mathsf{simp}(\Xi(\tau))$.*

Completeness The lemmas for completeness now explicitly include descriptions of the unrestricted components of contexts using the $\mathrm{un}(-)$ predicate, and the completeness theorem applies $\mathsf{simp}(-)$ to transform syntactic duals into semantic duals.

Lemma 9 (Completeness (Mixed merges))**.**

Sequential *If* $\Gamma = \Gamma_1 + \Gamma_2$*, with* $\Xi_1(\Delta_1) \subseteq \Gamma_1 \wedge \Xi_2(\Delta_2) \subseteq \Gamma_2$*, and where* $\mathrm{un}(\Gamma_1 \setminus \mathit{dom}(\Delta_1)) \wedge \mathrm{un}(\Gamma_2 \setminus \mathit{dom}(\Delta_2))$ *for some* $\Delta_1, \Delta_2, \Xi_1, \Xi_2$*, with* $\mathit{dom}(\Xi_1) \cap \mathit{dom}(\Xi_2) = \emptyset$*, then there exists a covering solution* $\Xi = \Xi_1 \cup \Xi_2$ *for* $\Delta_1 \bowtie_{mix} \Delta_2 = \Delta$*;* $\mathscr{C}$ *such that* $\Xi(\Delta) \subseteq \Gamma \wedge \mathrm{un}(\Gamma \setminus \mathit{dom}(\Delta))$*.*

Branching *Given* Γ *such that* $\Xi_1(\Delta_1) \subseteq \Gamma$ *where* $\mathrm{un}(\Gamma \setminus \mathit{dom}(\Delta_1))$*, and* $\Xi_2(\Delta_2) \subseteq \Gamma$ *with* $\mathrm{un}(\Gamma \setminus \mathit{dom}(\Delta_2))$ *for some* $\Xi_1, \Xi_2, \Delta_1, \Delta_2$*, with* $\mathit{dom}(\Xi_1) \cap \mathit{dom}(\Xi_2) = \emptyset$*, then there exists a covering solution* $\Xi = \Xi_1 \cup \Xi_2$ *of* $\Delta_1 \asymp_{mix} \Delta_2 = \Delta$*;* $\mathscr{C}$*, such that* $\Xi(\Delta) \subseteq \Gamma \wedge \mathrm{un}(\Gamma \setminus \mathit{dom}(\Delta))$*.*

Theorem 4 (Algorithmic completeness (mix))**.** *If* $\Gamma \vdash_{mix} M : A$ *then there exists a covering solution* Ξ *of the derivation* $M : \tau \triangleright_{mix} \Delta \,;\, \mathscr{C}$*, such that* $\Xi(\Delta) \subseteq \Gamma \wedge \mathit{simp}(\Xi(\tau)) = A \wedge \mathrm{un}(\Gamma \setminus \mathit{dom}(\Delta))$*.*

4 Implementation

We have implemented a parametric co-contextual typechecker that supports the type system in Figure 5 (generalised co-contextual typing), as well as Figure 11 (GV) using OCaml. We implemented syntactic sugar for let-expressions and if-expressions. The full typechecker including unification was written in 766 lines of code. The core typechecking algorithm covering Figure 5 and Figure 11 has 186 lines of code.

We provide a small example demonstrating typechecking in practice, and a large one demonstrating a TCP handshake protocol. First we show the small example, followed by the results of typechecking.

```
1  let s = fork (\s . send true s) in let (b, s) = receive s in (wait s); b
```

```
1  [Typecheck results]
2  Term type:
3  <Bool>
4  Under constraints:
5  {S(X2), S(X3), S(X6), Unit = Unit, X0 = X1, X1 = !Bool.X2, X4 = ?X5.X6,
6  X4 = ~(X3), X7 = end?, (X5 * X6) = (X8 * X7), (X0 -> X2) = (X3 -> end!)}
```

Here, ~X refers to syntactic duality $X^{\perp}$. We use a mild extension of Robinson's algorithm [28] to solve constraint sets; the full algorithm can be be found in the extended version [36]. The solution of the constraint set above is: $\{X_0 \mapsto X_1, X_1 \mapsto \mathsf{!Bool}.X_2, X_4 \mapsto \mathsf{?}X_5.X_6, X_7 \mapsto \mathsf{end?}, X_3 \mapsto (\mathsf{?}X_5.X_6)^{\perp}, X_5 \mapsto X_8, X_6 \mapsto \mathsf{end?}, X_2 \mapsto \mathsf{end!}, X_8 \mapsto \mathsf{Bool}\}$.

The output type is shown as Bool since the typechecker applied the solution. To verify that type variables X_2, X_3, X_6 are session types we apply the solution.

We continue with a larger example. In the extended version we provide the extended output of the typechecker, showing the output type, as well as the constraints generated [36].

Listing 1 Example GV implementation of TCP handshake.

```
let server = (\handshake .
  let (syn_seq, handshake) =
    receive handshake in
  let syn_ack_seq = 10 in
  let syn_ack_ack = (syn_seq + 1) in
  let syn_ack_pair =
    (syn_ack_seq, syn_ack_ack) in
  let handshake =
    send syn_ack_pair handshake in
  let (ack_pair, handshake) =
    receive handshake in
  let (ack_ack, ack_seq) = ack_pair in
  if ack_ack == (syn_ack_seq + 1)
    then
      if ack_seq == syn_ack_ack
        then handshake
        else failure handshake
    else failure handshake) in
let client = fork server in
let client = send 42 client in
let (syn_ack_pair, client) = receive client in
let (syn_ack_seq, syn_ack_ack) =
  syn_ack_pair in
let ack_pair =
  (syn_ack_ack, syn_ack_seq + 1) in
let fail_pair = (999, 999) in
if syn_ack_ack == (42 + 1)
  then
    let client = send ack_pair client in
    (wait client); true
  else
    let client = send fail_pair client in
    (wait client); false
```

5 Discussion and Related Work

This section positions our approach with respect to local and global approaches to type inference, before discussing related work.

5.1 Local and Global Type Inference

Co-contextual typing is a form of *global* constraint-based type inference. In contrast, *local* type inference approaches such as bidirectional typing by using information from adjacent language constructs [26] rather than using type variables and constraints. A known issue of global type inference is the imprecision of error locations, and this is a disadvantage of our co-contextual approach.

Backwards bidirectional typing, introduced by Zeilberger [41], and further discussed elaborated by Dunfield and Krishnaswami [7], is a co-contextual formulation of bidirectional typing where contexts are produced, rather than used as parameters. Its primary feature, compared to a co-contextual system like ours, is that type inference is local, rather than global. Consequently, constraints are not required. However, the expressiveness of backwards bidirectional typing remains an open question, and the requirement for type annotations may hinder usability. We leave a thorough investigation of backwards bidirectional typing for implementing session types as future work.

5.2 Related Work

The original formulation of co-contextual type systems [8] is parametric in a single merge function, which is similar to our sequential merge. When we only consider unrestricted types, sequential and branching merges coincide, but our generalisation to include both sequential and branching merges, as well as including a check function, allows us to describe substructural type systems. Kuci, Erdweg, Bracevac, Bejleri, and Mezini [19] investigate a co-contextual formulation of Featherweight Java [16], in turn demonstrating how co-contextual typing can be used in the object-oriented paradigm. It would be interesting to see whether our generalisation scales to the object-oriented setting. Our work proves soundness and completeness with respect to a declarative system, whereas [8] proves this with respect to a standard type inference system.

Jim [17] discusses *principal typings* in depth, later elaborated on by Wells [34]. A principal typing is a typing derivation that contains all possible typing information for a polymorphic expression, and is similar to co-contextual typing as it is derived only from a typable term as opposed to a typing environment. However, the work focuses on theoretical aspects of principal typings rather than an algorithmic presentation.

Quantitative Type Theory (QTT) [2, 23] tracks variable usage through a usage semiring. QTT combines substructural type theory with dependent type theory by distinguishing referencing resources in types (which is computationally irrelevant to linearity) and computations. This model is still contextual.

Turner, Wadler, and Mossin [29] show a call-by-need simply typed λ-calculus capable of tracking usage based on type annotations like *single-use* (linear, or *one*) and *multi-use* (alternatively *many* or ω), which was given a stronger analysis in Mogensen [24]. Wansbrough and Peyton Jones [33] extend this to polymorphic types and algebraic datatypes. In the setting of process calculi, Igarashi and Kobayashi [15] shows similar usage tracking of channels within a concurrent calculus used for code optimisation along with a type reconstruction algorithm. A notable linear system in practical use is Linear Haskell [3].

Granule [25] is a programming language with *graded modal types*, where types can be graded by elements of a usage semiring, e.g. the *none-one-many* semiring. Granule's implementation uses a bidirectional type checker with a standard input environment and an output environment that records usage information. An advantage of their approach is that Granule supports local type inference on types and can express precise constraints on variable usage (e.g., that a variable should be used precisely three times). The use of a semiring for grades also allows users to model features like information flow security. However, solving usage constraints requires the use of a stronger solver than a simple unification algorithm. Furthermore Granule's algorithmic type system is not proved to be sound and complete with respect to their declarative type system.

Links [5] use linear types to implement session types. Their type system is based around ML-style type inference and, like Granule, is implemented using a standard input environment and an output environment that records variable usage. To support the integration of polymorphism, linearity, session types, and row typing [32], Links' core language [21] includes a kinding system that associates type variables with a

kind (i.e., whether a type variable refers to a type, row, or *presence* variable used in row typing) along with a linearity annotation and a *restriction* (basic type, effect type, or a session type). The latter has parallels with our session constraint.

Wood and Atkey [37] use a posemiring over usage annotations, which in turn lends itself to producing a vector of usages over a context. This allows for fine-grained usage monitoring. Zackon, Sano, Momigliano, and Pientka [38] introduces 'CARVe' that (like QTT) allows annotations of variables with usage information, and gives tooling to reason about context splits and merges. The primary focus of this work is to aid in mechanisation of proofs involving substructural systems, rather than focusing on the provision of an algorithmic type system for λ-calculi, and does not use type inference.

Walker [31] provides a formulation of a linear type system that uses a judgement with both input and output contexts, i.e. $\Gamma_{\text{in}} \vdash M \dashv \Gamma_{\text{out}}$. The system is *subtractive* in that its variable rule removes a variable once it is used. Although the system is simple and does not require any constraints, it lacks error locality. For example, the expression $(\lambda x^{\text{Int}}.(x, x))\,5$ is ill-typed due to its reuse of x, but an implementation of the the type system as described would report an unbound variable error.

Allais [1] formalises *leftover typing*, a more refined subtractive system for the multiplicative-additive fragment of linear logic in Agda. Their typing judgement again has two contexts, but subtracts from usage information as opposed to removing a variable. Zalakain and Dardha [40] adapts leftover typing for the π-calculus, also in a dependently typed metalanguage. Closest to our work is a co-contextual formulation of linear typing for the π-calculus in Agda [39]. Their work provides a roadmap for showing soundness and completeness of their inference algorithm but the proofs are not complete. Furthermore we work in the setting of the λ-calculus which has a richer structure (e.g., nested subexpressions rather than variable literals).

Session types were introduced by Honda [12] and Honda, Vasconcelos, and Kubo [13]. We base our work on the GV session-typed λ-calculus [20, 30], which was based on the LAST session-typed language by Gay and Vasconcelos [10]. GV is inspired by logically-grounded session-typed process calculi from Caires and Pfenning [4].

6 Conclusion

Concurrent, and distributed applications are in need of *correct-by-construction* approaches that lend themselves to catching communication errors at compile time. We have generalised an approach—previously used mainly for incremental and parallelised typechecking—to various substructural typing disciplines, which lend themselves to the encoding of behavioural types. In doing so, we provided an extended unification algorithm that solves the constraints produced by our typechecking system. To show an example behavioural type implementation, we provided an implementation of the GV system in a system that allows a mixture of linear and unrestricted substructural modes. This system is capable of rejecting unrestricted, that is, unsafe usage of sessions using linear typing, while allowing other data types to be used in a standard unrestricted manner.

Future work *Mailbox types* [9] are a type system for incoming message queues known as mailboxes. Mailbox type systems rely not only on combining *usages* of variables, but also types themselves. The algorithmic type system for the Pat programming language [9] relies on backwards bidirectional typing in order to merge *types* as well as usages—indeed it is difficult to see how a type system that is not co-contextual type system could achieve this. We are currently investigating whether type merging can be used to allow an imperative interface to session types without requiring complex alias control, and how to use co-contextual typing to implement an algorithmic typechecker.

Another avenue for future work is investigating how further language features could be integrated in our framework: as an example, we believe the presented type systems can be extended with parametric polymorphism akin to Kuci [18].

It would also be interesting to investigate co-contextual implementations of other behaviourally typed systems (such as typestate for object-oriented programming) using the co-contextual methodology.

Acknowledgements We gratefully acknowledge support from a University of Glasgow College of Science and Engineering Studentship, which supported Weston's PhD studies, and EPSRC grant EP/T014628 (STARDUST).

References

[1] Guillaume Allais. "Typing with Leftovers: A Mechanization of Intuitionistic Multiplicative-Additive Linear Logic". In: *23rd International Conference on Types for Proofs and Programs*. Edited by Andreas Abel, Fredrik Nordvall Forsberg, and Ambrus Kaposi. Volume 104. Jan. 8, 2019, pages 1–22. ISBN: 978-3-95977-071-2. DOI: 10.4230/LIPIcs.TYPES.2017.1.

[2] Robert Atkey. "Syntax and Semantics of Quantitative Type Theory". In: *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*. LICS '18. Oxford, United Kingdom: Association for Computing Machinery, 2018, pages 56–65. ISBN: 978-1-4503-5583-4. DOI: 10.1145/3209108.3209189.

[3] Jean-Philippe Bernardy, Mathieu Boespflug, Ryan R. Newton, Simon Peyton Jones, and Arnaud Spiwack. "Linear Haskell: Practical Linearity in a Higher-Order Polymorphic Language". In: *Proc. ACM Program. Lang.* 2 (POPL Dec. 2017). DOI: 10.1145/3158093.

[4] Luís Caires and Frank Pfenning. "Session Types as Intuitionistic Linear Propositions". In: *CONCUR 2010 - Concurrency Theory*. Edited by Paul Gastin and François Laroussinie. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pages 222–236. ISBN: 978-3-642-15375-4. DOI: 10.1007/978-3-642-15375-4_16.

[5] Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. "Links: Web Programming Without Tiers". In: *Formal Methods for Components and Objects*. Edited by Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem-Paul de Roever. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pages 266–296. ISBN: 978-3-540-74792-5. DOI: 10.1007/978-3-540-74792-5_12.

[6] Ornela Dardha, Elena Giachino, and Davide Sangiorgi. "Session Types Revisited". In: *Proceedings of the 14th Symposium on Principles and Practice of Declarative Programming*. PPDP '12. Leuven, Belgium: Association for Computing Machinery, 2012, pages 139–150. ISBN: 978-1-4503-1522-7. DOI: 10.1145/2370776.2370794.

[7] Jana Dunfield and Neel Krishnaswami. "Bidirectional Typing". In: *ACM Computing Surveys* 54.5 (May 2021). ISSN: 0360-0300. DOI: 10.1145/3450952.

[8] Sebastian Erdweg, Oliver Bračevac, Edlira Kuci, Matthias Krebs, and Mira Mezini. "A Co-Contextual Formulation of Type Rules and Its Application to Incremental Type Checking". In: *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. OOPSLA 2015. New York, NY, USA: Association for Computing Machinery, 2015, pages 880–897. ISBN: 978-1-4503-3689-5. DOI: 10.1145/2814270.2814277.

[9] Simon Fowler, Duncan Paul Attard, Franciszek Sowul, Simon J. Gay, and Phil Trinder. "Special Delivery: Programming with Mailbox Types". In: *Proc. ACM Program. Lang.* 7.ICFP (Aug. 2023). DOI: 10.1145/3607832.

[10] Simon J. Gay and Vasco T. Vasconcelos. "Linear Type Theory for Asynchronous Session Types". In: *Journal of Functional Programming* 20.1 (2010), pages 19–50. DOI: 10.1017/S0956796809990268.

[11] Jean-Yves Girard. "Linear Logic". In: *Theoretical Computer Science* 50 (1987), pages 1–102. DOI: 10.1016/0304-3975(87)90045-4.

[12] Kohei Honda. "Types for Dyadic Interaction". In: *CONCUR'93*. Edited by Eike Best. Berlin, Heidelberg: Springer Berlin Heidelberg, 1993, pages 509–523. ISBN: 978-3-540-47968-0. DOI: 10.1007/3-540-57208-2_35.

[13] Kohei Honda, Vasco T. Vasconcelos, and Makoto Kubo. "Language Primitives and Type Discipline for Structured Communication-Based Programming". In: *Programming Languages and Systems*. Edited by Chris Hankin. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pages 122–138. ISBN: 978-3-540-69722-0. DOI: 10.1007/BFb0053567.

[14] Kohei Honda, Nobuko Yoshida, and Marco Carbone. "Multiparty Asynchronous Session Types". In: *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '08. San Francisco, California, USA: Association for Computing Machinery, 2008, pages 273–284. ISBN: 978-1-59593-689-9. DOI: 10.1145/1328438.1328472.

[15] Atsushi Igarashi and Naoki Kobayashi. "Type-Based Analysis of Communication for Concurrent Programming Languages". In: *Static Analysis*. Edited by Pascal Van Hentenryck. Berlin, Heidelberg: Springer Berlin Heidelberg, 1997, pages 187–201. ISBN: 978-3-540-69576-9. DOI: 10.1007/BFb0032742.

[16] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. "Featherweight Java: A Minimal Core Calculus for Java and GJ". In: *ACM Trans. Program. Lang. Syst.* 23.3 (May 2001), pages 396–450. ISSN: 0164-0925. DOI: 10.1145/503502.503505.

[17] Trevor Jim. "What Are Principal Typings and What Are They Good For?" In: *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '96. St. Petersburg Beach, Florida, USA: Association for Computing Machinery, 1996, pages 42–53. ISBN: 0-89791-769-3. DOI: 10.1145/237721.237728.

[18] Edlira Kuci. "Co-Contextual Type Systems: Contextless Deductive Reasoning for Correct Incremental Type Checking". PhD thesis. Darmstadt: Darmstadt University of Technology, Germany, Feb. 2020. DOI: 10.25534/tuprints-00011419.

[19] Edlira Kuci, Sebastian Erdweg, Oliver Bracevac, Andi Bejleri, and Mira Mezini. "A Co-Contextual Type Checker for Featherweight Java". In: *31st European Conference on Object-Oriented Programming (ECOOP 2017)*. Edited by Peter Müller. Volume 74. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2017, 18:1–18:26. ISBN: 978-3-95977-035-4. DOI: 10.4230/LIPIcs.ECOOP.2017.18.

[20] Sam Lindley and J. Garrett Morris. "A Semantics for Propositions as Sessions". In: *Programming Languages and Systems*. Edited by Jan Vitek. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pages 560–584. ISBN: 978-3-662-46669-8. DOI: 10.1007/978-3-662-46669-8_23.

[21] Sam Lindley and J. Garrett Morris. "Lightweight Functional Session Types". In: *Behavioural Types*. River Publishers, 2017, pages 265–286. ISBN: 978-87-93519-82-4. DOI: 10.1201/9781003337331-12.

[22] Conor McBride. "Everybody's Got to Be Somewhere". In: *Electronic Proceedings in Theoretical Computer Science* 275 (July 2018), pages 53–69. ISSN: 2075-2180. DOI: 10.4204/eptcs.275.6.

[23] Conor McBride. "I Got Plenty o' Nuttin'". In: *A List of Successes That Can Change the World: Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*. Edited by Sam Lindley, Conor McBride, Phil Trinder, and Don Sannella. Cham: Springer International Publishing, 2016, pages 207–233. ISBN: 978-3-319-30936-1. DOI: 10.1007/978-3-319-30936-1_12.

[24] Torben Æ. Mogensen. "Types for 0, 1 or Many Uses". In: *Implementation of Functional Languages*. Edited by Chris Clack, Kevin Hammond, and Tony Davie. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pages 112–122. ISBN: 978-3-540-68528-9. DOI: 10.1007/BFb0055427.

[25] Dominic Orchard, Vilem-Benjamin Liepelt, and Harley Eades III. "Quantitative Program Reasoning with Graded Modal Types". In: *Proc. ACM Program. Lang.* 3 (ICFP July 2019). DOI: 10.1145/3341714.

[26] Benjamin C. Pierce and David N. Turner. "Local Type Inference". In: *ACM Trans. Program. Lang. Syst.* 22.1 (Jan. 2000), pages 1–44. ISSN: 0164-0925. DOI: 10.1145/345099.345100.

[27] Ohad Rau, Caleb Voss, and Vivek Sarkar. "Linear Promises: Towards Safer Concurrent Programming". In: *35th European Conference on Object-Oriented Programming (ECOOP 2021)*. Edited by Anders Møller and Manu Sridharan. Volume 194. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021, 13:1–13:27. ISBN: 978-3-95977-190-0. DOI: 10.4230/LIPIcs.ECOOP.2021.13.

[28] John A. Robinson. "A Machine-Oriented Logic Based on the Resolution Principle". In: *Journal of the ACM* 12.1 (Jan. 1965), pages 23–41. ISSN: 0004-5411. DOI: 10.1145/321250.321253.

[29] David N. Turner, Philip Wadler, and Christian Mossin. "Once upon a Type". In: *Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture*. Functional Programming Languages & Computer Architecture (La Jolla, CA, USA). 1995. DOI: 10.1145/224164.224168.

[30] Philip Wadler. "Propositions as Sessions". In: *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming*. ICFP '12. New York, NY, USA: Association for Computing Machinery, 2012, pages 273–286. ISBN: 978-1-4503-1054-3. DOI: 10.1145/2364527.2364568.

[31] David Walker. "Substructural Type Systems". In: *Advanced Topics in Types and Programming Languages*. Edited by Benjamin C. Pierce. The MIT Press, Dec. 23, 2004. ISBN: 978-0-262-28159-1. DOI: 10.7551/mitpress/1104.003.0003.

[32] Mitchell Wand. “Complete Type Inference for Simple Objects”. In: *Proceedings of the Symposium on Logic in Computer Science (LICS ’87), Ithaca, New York, USA, June 22-25, 1987*. IEEE Computer Society, 1987, pages 37–44.

[33] Keith Wansbrough and Simon Peyton Jones. “Once upon a Polymorphic Type”. In: *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL ’99. San Antonio, Texas, USA: Association for Computing Machinery, 1999, pages 15–28. ISBN: 1-58113-095-3. DOI: 10.1145/292540.292545.

[34] J. B. Wells. “The Essence of Principal Typings”. In: *Automata, Languages and Programming*. Edited by Peter Widmayer, Stephan Eidenbenz, Francisco Triguero, Rafael Morales, Ricardo Conejo, and Matthew Hennessy. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pages 913–925. ISBN: 978-3-540-45465-6. DOI: 10.1007/3-540-45465-9_78.

[35] Olivia Weston. *Ntextual*. Version 1.0. University of Glasgow, Sept. 27, 2026. DOI: 10.5281/zenodo.21852108.

[36] Olivia Weston and Simon Fowler. “Splitting Is Stressful but Merging Is Manageable: Co-Contextual Typing for Substructural and Session Types (Extended Version)”. In: 11.33 (Oct. 6, 2026). DOI: 10.5281/zenodo.23190205.

[37] James Wood and Robert Atkey. “A Framework for Substructural Type Systems”. In: *Programming Languages and Systems. ESOP 2022*. Edited by Ilya Sergey. Volume 13240. Lecture Notes in Computer Science. Cham: Springer, 2022, pages 376–402. ISBN: 978-3-030-99336-8. DOI: 10.1007/978-3-030-99336-8_14.

[38] Daniel Zackon, Chuta Sano, Alberto Momigliano, and Brigitte Pientka. “Split Decisions: Explicit Contexts for Substructural Languages”. In: *Proceedings of the 14th ACM SIGPLAN International Conference on Certified Programs and Proofs*. CPP ’25. Denver, CO, USA: Association for Computing Machinery, 2025, pages 257–271. ISBN: 979-8-4007-1347-7. DOI: 10.1145/3703595.3705888.

[39] Uma Zalakain and Ornela Dardha. *Co-Contextual Typing Inference for the Linear π-Calculus in Agda (Extended Abstract)*. Presented at TyDE. 2021. URL: https://icfp21.sigplan.org/details/TyDe-2021/5/Co-Contextual-Typing-Inference-for-the-Linear-Calculus-in-Agda-Extended-Abstract- (visited on 2026-06-16).

[40] Uma Zalakain and Ornela Dardha. “Pi with Leftovers: A Mechanisation in Agda”. In: *Formal Techniques for Distributed Objects, Components, and Systems*. Edited by Kirstin Peters and Tim A. C. Willemse. Cham: Springer International Publishing, 2021, pages 157–174. ISBN: 978-3-030-78089-0. DOI: 10.1007/978-3-030-78089-0_9.

[41] Noam Zeilberger. *Balanced Polymorphism and Linear Lambda Calculus*. Unpublished extended abstract for presentation given at TYPES 2015. Tallinn, 2015. URL: https://cs.ioc.ee/types15/abstracts-book/contrib35.pdf (visited on 2026-11-04).

About the authors

Olivia Weston o.weston.1@research.gla.ac.uk
Olivia Weston is a researcher in programming language theory with a focus on types and concurrent languages. She has a degree in Logic and Computation (MRes) from Swansea University under the supervision of Cécilia Pradic, and as of 2026 is studying towards a PhD in Computing Science from the University of Glasgow under the supervision of Simon Fowler.
https://orcid.org/0009-0000-1325-0533

Simon Fowler simon.fowler@glasgow.ac.uk
Simon Fowler is a Lecturer in Programming Language Foundations at the University of Glasgow. He was awarded his PhD from the University of Edinburgh in 2019. Simon's primary research interests are in programming abstractions and behavioural type systems.
https://orcid.org/0000-0001-5143-5475