Fusing Session-Typed Concurrent Programming into Functional Programming

CHUTA SANO, McGill University, Canada
DEEPAK GARG, Max Planck Institute for Software Systems, Germany
RYAN KAVANAGH, Université du Québec à Montréal, Canada
BRIGITTE PIENTKA, McGill University, Canada
BERNARDO TONINHO, Instituto Superior Técnico - University of Lisbon, Portugal

We introduce FuSes, a **Fu**nctional programming language that integrates **Ses**sion-typed concurrent process calculus code. A functional layer sits on top of a session-typed process layer. To generate and reason about open session-typed processes, the functional layer uses the contextual box modality extended with linear channel contexts. Due to the fundamental differences between the operational semantics of the functional layer and the concurrent semantics of processes, we bridge the two layers using a set of primitives to run and observe the behavior of closed processes within the functional layer. In addition, FuSes supports code analysis and manipulation of open session-typed process code. To showcase its benefit to programmers, we implement well-known optimizations, such as batch optimizations, as type-safe metaprograms over concurrent processes.

Our technical contributions include a type system for FuSes, an operational semantics, a proof of its type safety, and an implementation.

CCS Concepts: • **Theory of computation** → **Linear logic**; **Type theory**; • **Software and its engineering** → **Functional languages**; **Concurrent programming languages**; • **Computing methodologies** → **Concurrent programming languages**.

Additional Key Words and Phrases: linear logic, concurrency, session types, metaprogramming

ACM Reference Format:
Chuta Sano, Deepak Garg, Ryan Kavanagh, Brigitte Pientka, and Bernardo Toninho. 2025. Fusing Session-Typed Concurrent Programming into Functional Programming. *Proc. ACM Program. Lang.* 9, ICFP, Article 250 (August 2025), 28 pages. https://doi.org/10.1145/3747519

1 Introduction

Concurrent computation is an inherent part of modern software systems, making them challenging to reason about due to the non-deterministic nature of concurrency. One promising approach to establish static safety guarantees about concurrent computation is session-typed message-passing concurrency [Honda 1993]. In this setting, a session type specifies the protocol according to which processes communicate. This allows us to check statically that a given process correctly communicates over its channels and to catch many potential errors, such as mismatched messages or incorrect sequences. However, it has been challenging to integrate session-typed message-passing concurrency into statically-typed (functional) programming. This is largely due to the gap between

Authors' Contact Information: Chuta Sano, McGill University, Montreal, Canada, chuta.sano@mail.mcgill.ca; Deepak Garg, Max Planck Institute for Software Systems, Saarbruecken, Germany, dg@mpi-sws.org; Ryan Kavanagh, Université du Québec à Montréal, Montreal, Canada, kavanagh.ryan@uqam.ca; Brigitte Pientka, McGill University, Montreal, Canada, bpientka@cs.mcgill.ca; Bernardo Toninho, Instituto Superior Técnico - University of Lisbon, Lisbon, Portugal, bernardo.toninho@tecnico.ulisboa.pt.

ACM 2475-1421/2025/8-ART250
https://doi.org/10.1145/3747519

their underlying computational models: message-passing communication is inherently effectful, while ordinary (functional) programming is pure. How can these two computational paradigms interoperate safely and modularly?

One popular approach to incorporating session types into functional programming is through libraries [Imai et al. 2019; Kokke and Dardha 2021; Padovani 2017]. These libraries are ad-hoc and often exploit concurrent primitives in the functional language as well as unsafe features. This makes it hard to give general safety guarantees about the libraries and their implementations.

A more foundational approach to integrating session types into functional programming is GV [Gay and Vasconcelos 2010]. It combines functional programming with session types by extending the functional language with send and receive primitives, allowing functions to act as processes. However, the intertwined nature of pure functions and effectful session primitives makes GV hard to reason about.

In this paper, we present FuSes, a **Fu**nctional programming language that integrates **Ses**sion-typed concurrency.

FuSes is inspired by Session-typed Intuitionistic Linear Logic (SILL), a process calculus that modularly incorporates functional programming [Toninho et al. 2013]. In SILL, processes can send and receive functional values, and functions can generate open process code. In contrast, FuSes is a functional language that modularly incorporates session-typed processes. As in SILL, functions in FuSes can generate process code and additionally run process code and observe its effects. Furthermore, they can recursively analyze and manipulate process code, allowing programmers to write complex process code transformations and optimizations in a type-safe manner. FuSes therefore integrates session-typed processes by extending a core functional language with the ability to: (1) generate and compose session-typed process code; (2) run process code and observe its effects; and (3) recursively analyze session-typed process code.

(1) Type-Safe Code Generation and Composition in FuSes. The foundation of FuSes is a two-layered system with a functional layer that generates process code and a message-passing process layer that sends and receives functional values. Processes follow a client-server architecture where every process *offers* a single communication channel (server) while *using* multiple channels (clients). These channels are given session types in a linear type system so that processes can be type checked to ensure that they communicate correctly.

As in SILL, we use the contextual box modality [Nanevski et al. 2008] to capture open code which depends not only on the usual functional context but also a linear channel context. We therefore extend the variant of contextual box from SILL [Toninho et al. 2013], which does not capture any functional assumptions, to describe a session-typed process code that is open with respect to both values and channels.

To illustrate, we use the `box` construct to quote a process that is open with respect to some number `x` and a channel `a`. This process first receives a number from its client channel `a`, sends the sum of it with `x` across its offering channel `c`, and then continues:

```
box(x;a ⟶ | y = recv a; send c (x + y); ... | ⟶ c)
```

The assumptions on the left (`x ; a`) and the channel name on the right (`c`) *bind* variables of the appropriate kind for the underlying open code which we frame for clarity.

To compose a quoted process with other processes, we must first bind its code using a *let*-style eliminator:

```
letbox u = box(x; a ⟶ | y = recv a; send c (x + y); ... | ⟶ c) in ...
```

Here, `u` will be bound to `y = recv a; send c (x + y); ...`. To use `u` inside another quotation, we must splice it in by applying the appropriate substitutions to rename or instantiate `x` (the value), the input channel a, and the output channel c. For example, we can compose u with another process by spawning it in parallel. The two processes then communicate over the shared channel b where u is expected to send a number and the other process to receive a number:

```
box( ; a' → [b ← spawn u[5/x; a'/a, b/c] | let y = recv b; ...] → c)
```

(2) Observing Processes in FuSes. While running code tends to come for free in most metaprogramming systems by simply "running" the code and reifying its return value, this is not the case anymore when our programs are session-typed processes. A process does not "return" a single value but rather we execute it for its communication effects. Suppose we want to run a process that sends and receives some sequence of numbers:

```
box( ; → send c 1; x = recv c; send c (x+1); ... → c)
```

To observe this process, we must not only receive the first message, 1, but also continue to execute the process so that we can send some number and then receive another number. Furthermore, the execution of most session-typed processes is inherently non-deterministic. In FuSes, we use a standard operational semantics for the functional layer and use a multiset rewriting semantics for the process layer. In general, processes execute until they are stuck trying to communicate on their offering channels. The functional layer then resolves this stuck process by communicating with it. Afterwards, the functional layer binds not only the process it communicated with but also its environment as code so that it can continue to run and observe further communication from the process. This allows the functional layer to communicate with the process layer while preserving the semantics of both layers. We are not aware of prior work that has considered the interoperability of pure functional programming with concurrent processes, and we see our work as an important step towards interoperability of languages with fundamentally very different operational behaviours.

(3) Type-Safe Process Code Analysis in FuSes. Metaprogramming for functional programming is used to express runtime properties, perform code optimizations using domain-specific or even runtime knowledge, eliminate boilerplate code, etc. Since these techniques also apply to the concurrent setting, we extend FuSes with the ability to recursively analyze and traverse open process code.

While metaprogramming systems such as Moebius [Jang et al. 2022] explore code analysis for well-typed, open functional code, no work has done the same for open process code. A common technical challenge in supporting type-safe code analysis via pattern matching is accounting for the fact that types may be refined and contexts can grow as we recursively traverse code. To support recursive analysis of arbitrary code, contexts and types must therefore be first class.

In FuSes, the situation is even more complex; process code can depend on a functional and a channel context and on functional and session types. The channel context is notably linear, which means it can not only grow but also shrink and split as we traverse a process. To our knowledge, only LINCX [Georges et al. 2017] supports the analysis of open code with linear contexts, and moreover, its technique makes many simplifying assumptions that are incompatible with process code. Furthermore, processes can embed functional terms, which themselves can be open with respect to functional values. Thus, FuSes must support first-class contexts, types, and functional code all while dealing with the challenges associated with managing linear contexts. In particular, we allow top-level functions to be defined via pattern matching on well-typed session code and well-typed functional code.

In short, we make the following contributions in this paper.

(1) We present FuSes, a type-safe foundation for concurrent programming that separates "ordinary" functional programming and concurrent message-passing programming. In this setting,

open session-typed process code is first class and can be quoted and composed. We prove type preservation for this core.
(2) To enable the functions to run and observe session-typed process code, we give a multiset rewriting semantics for processes. We then extend the functional layer of FuSes with a collection of primitives that perform one synchronized communication with an executed process and then provide a code representation of the remaining process and its environment. We prove that this extension maintains type preservation, which mainly involves proving type preservation for the multiset rewriting semantics for processes.
(3) To support type-safe code transformations and code optimizations, we extend our system with recursion and analysis on session-typed process code (and necessarily contexts, types, and functional code) through top-level functions with pattern spines. The main technical contribution is extending our metatheory to account for these new first-class objects. To our knowledge, no existing metaprogramming system provides support for pattern matching on open session-typed process code.
(4) We implement in FuSes several examples of common concurrent code transformations including batch optimization, monitor insertion, and tracing. Our type system ensures that these optimizations are type-preserving, and therefore, these examples demonstrate the expressivity of our language.

2 FuSes by Example

To give motivation and intuition on the three components of FuSes, we explore a few examples in pseudocode, expanding the sketch given in the introduction.

2.1 Generating and Composing Session-Typed Concurrent Processes

To recap, we can quote open process code with the constructor `box(Γ; Δ → P → c:C)` which describes a suspended process `P` that offers a channel $c : C$ and is open with functional assumptions $\Gamma = \overline{x : \tau}$ and linear channels $\Delta = \overline{a : A}$. The type of this expression is of the form $\{\Gamma; \Delta \vdash c : C\}$. In the interest of readability, we do not annotate any of the assumptions with types in our examples unless they are needed.

For instance, `box(·;· → (send c 5; close c) → c)` denotes a suspended closed process that sends a natural number `5` and then terminates. The protocol over channel `c` is captured by the type `nat ∧ 1`, where the session type `1` denotes termination, so the type of this process code is `{·; · ⊢ c: nat ∧ 1}`.

To compose and refer to suspended processes on the functional layer, we introduce the eliminator `letbox u = M in N` that binds some suspended process code `M` to a globally accessible variable `u` that `N` can access.

Example 2.1.

```
letbox u = box(x; a → (y = recv a; wait a; send c (x + y); close c) → c) % Bind the process code
to u
in box(·;a' → (b ← spawn u[5/x; a'/a, b/c] ← a'             % Run u in parallel with a'
                | (x = recv b; wait b; close c)) → c)
```

Here we spawn the process `u` by passing a channel `a'` to it. The substitution `u[5/x; a'/a, b/c]` provides mappings for all free variables in `u`. This process runs in parallel with the process `x = recv b; wait b; close c`. Intuitively, this process receives a number from `u` over channel `b`, waits for `u` to terminate, and then terminates.

2.2 Running and Observing Code

As motivation, consider a function `gen_fib n` that returns a closed process code that concurrently computes the n-th Fibonacci number and sends it alongside its offering channel c of type `nat ∧ 1`.

```
gen_fib: nat → {·;· ⊢ c: nat ∧ 1}
gen_fib ⇒ λn.
  if n = 0 or n = 1 then
    box (·;· → send c 1; close c → c)    % base cases; return the trivial process outputting 1
  else
    letbox u = gen_fib (n-1) in
    letbox v = gen_fib (n-2) in
    box (·;· → (a ← spawn u[a/c]) |      % spawn process u with channel name a
               (b ← spawn v[b/c]) |      % spawn process v with channel name b
               (x = recv a;              % receive a nat on channel a
                y = recv b;              % receive a nat on channel b
                wait a;                  % wait for P to close channel a
                wait b;                  % wait for Q to close channel b
                send c (x + y);
                close c))
             → c)
```

In the base cases where `n = 0` or `n = 1`, the function outputs a trivial process code that sends the number `1` and then terminates. Otherwise, it first calls itself recursively twice to obtain `u` and `v`, which represent process code that output the `n-1` and `n-2` Fibonacci numbers, respectively. It then composes `u` and `v` to create a process that receives a number from both `u` and `v` and then sends its sum alongside its offering channel.

At this point, our metaprogramming system has only generated a suspended process that concurrently computes the number. If we want to actually compute the Fibonacci number, e.g.

```
fib: nat → nat
fib n ⇒
  letbox u = gen_fib n in
  ...
```

Then we are stuck because we would still need to run the concurrent process to extract an output.

To that end, we define a primitive `let (x1, ..., xn) = run M in N` that runs a closed (with respect to its channel context) process code `M` until it is stuck (i.e., attempting to communicate across its offered channel) and binds some tuple of data `x1, ..., xn` for `N` to use. The structure of this data is uniquely determined by the offering session type.

For instance, if we run a closed process of type `{· ; · ⊢ nat ∧ A}`, we expect the process to output a `nat` and then continue communicating according to `A`. We therefore think of running such a process as outputting a natural number alongside a process code that represents the remaining computation and communication. We can use the run primitive to execute the generated Fibonacci process and get its output number:

```
fib: nat → nat
fib n ⇒
  let (n, _) = run (gen_fib n) in
  n
```

We ignore the continuation process _ of type `{· ; · ⊢ 1}`, i.e., a process that eventually terminates. We run `gen_fib` to completion and only care about the number that the process outputs. However, when running processes with more complex interactions, we would use the continuation process in a monadic style. For example, if we run a process that sends, receives, and then sends a number:

```
let (n, u) = run(box(· ; · → (send c 1; x = recv c; send c x; close c) → c)) in ...
```

The continuation process variable `u` corresponds to the process code `x = recv c; ...`. Since this process expects to receive some value, running this process no longer gives a pair. Instead, it gives a process code `u'` that is open with respect to `x`, the value it receives:

```
let u' = run(box(· ; · → u → c)) in ...
```

`u'` now corresponds to the process code `send c x; ...` and again, is open with respect to a number `x`. To continue running this process, we must instantiate `x` by providing a substitution. As an example, we substitute the number `5`.

```
let (n', u'') = run(box(x ; · → u' → c)[5/x]) in ...
```

2.3 Analyzing Session-Typed Process Code

Our system supports code analysis for concurrent programs through top-level functions and pattern matching on code. A top-level function consists of a *spine*, or a list of patterns that each contains an implementation for some pattern fragment.

For example, consider a *deep-copy* function that traverses an arbitrary process and produces a copy of it. As a first attempt, the `copy` function takes as input some open process code `u:{?; ? ⊢ c:?}` and outputs a process code of the same type `{?; ? ⊢ c:?}`

```
copy : (u:{?;? ⊢ c:?}) → {?;? ⊢ c:?}
copy (?; ? → wait a; u → c) ⇒ ...
```

Since the process is open, we must make the contexts in which it lives first class. Therefore, the `copy` function must also abstract over some functional context `γ:ctx`, linear channel context `δ:lctx`, and the offering channel's session type `C:stp`:

```
copy : (γ:ctx) → (δ:lctx) → (C:stp) → (u:{γ;δ ⊢ C}) → {γ;δ ⊢ C}
```

This essentially means that function types are now dependent function types (Π-type). We name each input in the function type as is common in Π-types.

To implement the function `copy`, we write pattern spines that match simultaneously on the structure of all the inputs, i.e., functional context `γ:ctx`, linear channel context `δ:lctx`, the offering channel's session type `C:stp`, and most importantly the processes `u`. For example, for `u` to be of the form `close c`, its channel context must be empty (due to linearity) and its offering session type must be `1`:

```
copy γ · 1 (γ; ·→ close c → c) ⇒ box(γ; ·→ close c → c)
```

If the process is of the form `x = recv a; u'` where `a` is a client channel (i.e., not the offering channel), then its channel context must contain some `a:τ ∧ A`:

```
copy γ (δ, a:τ ∧ A) C (γ; δ, a → (x = recv a; u') → c) ⇒
  letbox v = copy (γ, x:τ) (δ, a:A) C u' in
  box(γ; δ, a → x = recv a; v → c)
```

Since we treat both functional and channel contexts as sets, the pattern `(δ, a:τ ∧ A)` matches any linear context containing some channel `a:τ ∧ A`. In the recursive call, we extend the functional context `(γ,x:τ)` since `u'` now depends on the received value `x`, and the type of channel `a` is now simply `A`.

The final interesting case is when the process is a parallel composition of two processes, i.e., `u` is of the form `a ← spawn u1 ← δ1 | u2`. In this case, the channel context `δ` is split into two contexts `δ1` and `δ2` where `δ1 ⋈ δ2` with `⋈` denoting a context merge:

```
copy γ (δ1 ⋈ δ2) C (γ; δ1 ⋈ δ2 → (a ← (spawn u1 ← δ1) | u2) → c) ⇒ ...
```

In this case, we would like to recursively copy both `u1` and `u2`, but the former is a bit problematic:

```
  letbox u1' = copy γ δ1 ? u1 in
```

Indeed, we do not know the session type of `u1's` offering channel. Therefore, we require type annotations on the offered channel `a` in the spawn construct to proceed:

```
copy γ (δ1 ⋈ δ2) C (γ; δ1 ⋈ δ2 → (a:A ← (spawn u1 ← δ1) | r2) → c) ⇒ ...
  letbox u1' = copy γ δ1 A u1 in
  letbox u2' = copy γ (δ2, a:A) C u2 in
  box(γ;δ1 ⋈ δ2 → ((a ← spawn u1' ← δ1) | u2') → c)
```

For the sake of conciseness, we will continue to omit types as much as possible in later examples and only include them when necessary.

3 Code Generation

We first introduce the core fragment of FuSes which contains a functional layer that captures process calculus syntax through a contextual type (Figure ??). We will expand this core fragment in Section 4 and Section 5 to support running process code and code analysis, respectively.

Session Types	A, B, C	$::=$	$1 \mid \tau \wedge A \mid \tau \supset A$
Functional Types	τ_1, τ_2	$::=$	$\tau_1 \longrightarrow \tau_2 \mid \{\Gamma; \Delta \vdash a : A\}$
Metalevel assumptions	$\mathcal{S}$	$::=$	$(\Gamma; \Delta \vdash a : A)$
Metalevel objects	$\mathcal{N}$	$::=$	$(\Gamma; \Delta.P :: c : C)$
Linear Context	Δ	$::=$	$\cdot \mid \Delta, a : A$
Functional Context	Γ	$::=$	$\cdot \mid \Gamma, x : \tau$
Global metacontext	Σ	$::=$	$\cdot \mid \Sigma, u : \mathcal{S}$
Metalevel Substitutions	θ	$::=$	$\cdot \mid \theta, \mathcal{N}$
Term-level Substitutions	σ	$::=$	$\cdot \mid \sigma, M/x$
Channel Renamings	ρ	$::=$	$\cdot \mid \rho, a/b$
Processes	P, Q	$::=$	$\mathsf{fwd}\ a\, b \mid \mathsf{close}\ a \mid \mathsf{wait}\ a; P \mid (a{:}A \leftarrow P \leftarrow \Delta \mid Q)$
			$\mid \mathsf{send}\ c\ M; P \mid x = \mathsf{recv}\ c; P \mid u[\sigma; \rho]$
Terms	M, N	$::=$	$x \mid \lambda x.M \mid M\ N \mid \mathsf{letbox}\ u = M\ \mathsf{in}\ N$
			$\mid \mathsf{box}(\Gamma; \Delta \vdash P :: c : C)$

Fig. 1. Syntax of the core fragment of FuSes.

3.1 Functional Layer of FuSes

We start with a simply-typed λ-calculus and extend its typing judgments with the context Σ which holds global assumptions of process code. We view Γ and Δ as unordered and Σ as an ordered context to anticipate our later developments which allow assumptions in Σ to depend on prior assumptions.

Type judgments of functional terms are of the form

$$\Sigma \mid \Gamma \Vdash M : \tau$$

with the standard rules of functions extended with Σ:

$$\frac{}{\Sigma \mid \Gamma, x : \tau \Vdash x : \tau}\ x \qquad \frac{\Sigma \mid \Gamma, x : \tau_1 \Vdash M : \tau_2}{\Sigma \mid \Gamma \Vdash \lambda x.M : \tau_1 \to \tau_2} \to I \qquad \frac{\Sigma \mid \Gamma \Vdash M : \tau_1 \to \tau_2 \quad \Sigma \mid \Gamma \Vdash N : \tau_1}{\Sigma \mid \Gamma \Vdash M\, N : \tau_2} \to E$$

To this system we add a contextual type $\{\Gamma; \Delta \vdash c : C\}$ to capture open process code that depends on some functional unrestricted context Γ and a linear channel context Δ to provide a communication channel c of type C.

Its introduction rule continues checking that the code is well-typed with respect to the process layer typing, which we define in the following section:

$$\frac{\Sigma \mid \Gamma; \Delta \vdash P :: (c : C)}{\Sigma \mid \Gamma' \Vdash \mathsf{box}(\Gamma; \Delta \vdash P :: c : C) : \{\Gamma; \Delta \vdash c : C\}}\ \mathrm{box}I$$

Note that in our core syntax we require full typing annotations on the contextual box. This is in contrast to the examples that we gave earlier, such as in Section 2.1, where we mostly omitted type annotations. This will make the overall development simpler, but the type annotations can usually be inferred in practice.

When we transition to process typing in the premise, we forget our functional context Γ′ and switch to the functional context Γ given in the box term. This ensures that the process is closed with respect to Γ and Δ. We pass Σ, the metacontext containing metavariables referring to processes, on to the process layer typing judgment to enable splicing of process code. This strict isolation of contexts is needed later when we consider processes that can receive (and send) functional values.

The metacontext Σ can be extended by eliminating a term of box type using a let-style eliminator:

$$\frac{\Sigma \mid \Gamma' \Vdash M : \{\Gamma; \Delta \vdash c : C\} \quad \Sigma, u : (\Gamma; \Delta \vdash c : C) \mid \Gamma' \Vdash N : \tau}{\Sigma \mid \Gamma' \Vdash \mathsf{letbox}\ u = M\ \mathsf{in}\ N : \tau}\ \mathrm{box}E$$

This rule moves a quoted process code M to the global metacontext, making it accessible under any boxes in N.

3.2 Process Layer of FuSes

As previewed in the previous section, process layer typing judgments check that a process P offers a channel $c : C$ while using channels in Δ, values in Γ, and metaassumptions in Σ.

$$\Sigma \mid \Gamma; \Delta \vdash P :: (c : C)$$

We take a very simple subset of (intuitionistic) session types consisting of the unit 1, corresponding to termination, value output $\tau \wedge A$, and value input $\tau \supset A$. We discuss in Section 7 how we can extend our system to the standard session type connectives ⊗, ⊸, ⊕, and &.

Typing rules in this layer correspond to proofs in the sequent calculus for intuitionistic linear logic where left rules correspond to interacting on a used channel as a *client*, while right rules correspond to interacting on the offered channel as a *server*. For each session type, the left and right rules must be dual to ensure that no communication mismatch occurs.

For instance, the right rule for the unit 1 corresponds to termination, which can be thought of as *sending* a termination signal, which we write as close c for some channel c. Therefore, the left rule must *receive* a termination signal and then proceed as some continuation process P, which we write as wait $a; P$ for some channel a.

$$\frac{\Sigma \mid \Gamma; \Delta \vdash P :: (c : C)}{\Sigma \mid \Gamma; \Delta, a : 1 \vdash \mathsf{wait}\ a; P :: (c : C)}\ 1L \qquad \frac{}{\Sigma \mid \Gamma; \cdot \vdash \mathsf{close}\ c :: (c : 1)}\ 1R$$

Analogously, the right rule for value output $\tau \wedge A$ must *send* a value of type τ and continue as type A, while the left rule must *receive* a value of type τ and continue using the channel as A. The right rule also refers to the expression layer typing to ensure that the sent value is well-typed.

$$\frac{\Sigma \mid \Gamma, x:\tau; \Delta, a:A \vdash P :: (c:C)}{\Sigma \mid \Gamma; \Delta, a:\tau \wedge A \vdash x = \mathsf{recv}\ a; P :: (c:C)} \wedge L \qquad \frac{\Sigma \mid \Gamma; \Delta \vdash P :: (c:C) \quad \Sigma \mid \Gamma \Vdash M:\tau}{\Sigma \mid \Gamma; \Delta \vdash \mathsf{send}\ c\ M; P :: (c:\tau \wedge C)} \wedge R$$

The rules for $\tau \supset A$ are dual; the right rule is a receive while the left rule is a send:

$$\frac{\Sigma \mid \Gamma; \Delta, a:A \vdash P :: (c:C) \quad \Sigma \mid \Gamma \Vdash M:\tau}{\Sigma \mid \Gamma; \Delta, a:\tau \supset A \vdash \mathsf{send}\ a\ M; P :: (c:C)} \supset L \qquad \frac{\Sigma \mid \Gamma, x:\tau; \Delta \vdash P :: (c:C)}{\Sigma \mid \Gamma; \Delta \vdash x = \mathsf{recv}\ c; P :: (c:\tau \supset C)} \supset R$$

The identity rule types a process that forwards communication between its two channels:

$$\frac{}{\Sigma \mid \Gamma; a:C \vdash \mathsf{fwd}\ a\, c :: (c:C)}\ \mathsf{id}$$

The cut rule corresponds to spawning a process offering a channel of the cut type in parallel and then continuing:

$$\frac{\Sigma \mid \Gamma; \Delta_1 \vdash P :: (a:A) \quad \Sigma \mid \Gamma; \Delta_2, a:A \vdash Q :: (c:C) \quad \Delta = \Delta_1 \bowtie \Delta_2}{\Sigma \mid \Gamma; \Delta \vdash a{:}A \leftarrow P \leftarrow \Delta_1 \mid Q :: (c:C)}\ \mathsf{spawn}$$

We make here explicit the context merge $\Delta_1 \bowtie \Delta_2$, which, for now, is defined in the obvious manner.

Finally, we can splice in process code u in Σ by providing a substitution σ between the functional contexts and a renaming ρ between the channel contexts, both defined in the following section:

$$\frac{u:(\Gamma'.\Delta' \vdash c:C) \in \Sigma \quad \Sigma \mid \Gamma \Vdash \sigma:\Gamma' \quad \Sigma \mid \Delta \Vdash \rho:\Delta'}{\Sigma \mid \Gamma; \Delta \vdash u[\sigma;\rho] :: (c:C)}\ P_{\mathsf{var}}$$

We do not explicitly rename the offered channel c since every process offers a single channel, so they can be implicitly α-renamed accordingly.

3.3 Substitutions and Renamings in FuSes

Our language defines three kinds of "substitutions": term substitutions σ; channel renamings ρ; and metasubstitutions θ. Thus far, the former two appear directly in the syntax of process code splicing $u[\sigma;\rho]$ to allow splicing of process code variables open under different functional and channel contexts. We define σ as a set of mappings between terms and variable names

$$\frac{}{\Sigma \mid \Gamma \Vdash \cdot : \cdot}\ \sigma_{\mathsf{empty}} \qquad \frac{\Sigma \mid \Gamma \Vdash M:\tau \quad \Sigma \mid \Gamma \Vdash \sigma : (\Gamma' \setminus (x:\tau)) \quad (x:\tau) \in \Gamma'}{\Sigma \mid \Gamma \Vdash \sigma, M/x : \Gamma'}\ \sigma_{\mathsf{dec}}$$

and ρ as a set of channel name mappings

$$\frac{}{\Sigma \mid \cdot \Vdash \cdot : \cdot}\ \rho_{\mathsf{empty}} \qquad \frac{\Sigma \mid \Delta \Vdash \rho : (\Delta' \setminus (a':A)) \quad (a':A) \in \Delta'}{\Sigma \mid \Delta, a:A \Vdash \rho, a/a' : \Delta'}\ \rho_{\mathsf{dec}}$$

Term-level substitutions $[\sigma]M$ and $[\sigma]P$ provide (capture-avoiding) instantiations for functional assumptions and are mutually defined for both terms and processes. For a given term-level substitution σ, we assume a lookup operation $\sigma(x)$ that returns some term M if $M/x \in \sigma$ and x otherwise.

For terms, the definition is standard:

$$[\sigma](M\ N) = [\sigma]M\ [\sigma]N \quad [\sigma]x = \sigma(x) \quad [\sigma](\lambda x.M) = \lambda x.[\sigma, x/x]M \quad \ldots$$

One notable case is a box; since the captured process code is open with respect to an unrelated functional context, the substitution must stop:

$$[\sigma]\mathsf{box}(\Gamma;\Delta \vdash P :: c:C) = \mathsf{box}(\Gamma;\Delta \vdash P :: c:C)$$

For processes, term substitutions recursively traverse the process in the usual way:

$$[\sigma]\mathsf{close}\ c = \mathsf{close}\ c \quad [\sigma](a{:}A \leftarrow P \leftarrow \Delta \mid Q) = a{:}A \leftarrow [\sigma]P \leftarrow \Delta \mid [\sigma]Q \quad \ldots$$

The only interesting case is when it encounters a value send, where it applies the substitution to the term that is being sent:

$$[\sigma](\mathsf{send}\ a\ M; P) = \mathsf{send}\ a\ [\sigma]M; [\sigma]P$$

On the other hand, channel renamings $[\rho]P$ are only defined for processes since functions do not depend on channel contexts. We assume a similar lookup function $\rho(a)$ that returns b if $b/a \in \rho$ and a otherwise. The definition of channel renaming is as one would expect. For instance,

$$[\rho](\mathsf{wait}\ a; P) = \mathsf{wait}\ \rho(a); [\rho]P \quad [\rho_1, \rho_2](a{:}A \leftarrow P \leftarrow \Delta \mid Q) = a{:}A \leftarrow [\rho_1]P \leftarrow \Delta \mid [\rho_2]Q$$

It is worth mentioning that although we declaratively assume a split ρ_1, ρ_2 in the second case, this can be easily computed by referencing Δ, the context for process P.

The metasubstitution θ is for now, also defined in a similar manner to term-level substitutions; it is a list of metaobjects, i.e., process code, that provides instantiations of assumptions in some meta-context Σ. The only difference is that θ is a list of metaobjects instead of a list of mappings because Σ is ordered; the domain can therefore be inferred. We defer the details of the metasubstitution to Section 5 where we extend the metalevel syntax to accommodate code analysis.

3.4 Example Derivations

We can type a boxed process that sends the number 5 (as given in Section 2.1) as follows, where we omit the derivation that 5 is a natural number:

$$\dfrac{\dfrac{\dfrac{}{\cdot \mid \cdot; \cdot \vdash \mathsf{close}\ c :: (c : 1)}\ 1R \qquad \dfrac{\dots}{\cdot \mid \cdot \Vdash 5 : \mathsf{nat}}}{\cdot \mid \cdot; \cdot \vdash \mathsf{send}\ c\ 5; \mathsf{close}\ c :: (c : \mathsf{nat} \wedge 1)}\ \wedge R}{\cdot \mid \cdot \Vdash \mathsf{box}(\cdot; \cdot \vdash \mathsf{send}\ c\ 5; \mathsf{close}\ c :: c : \mathsf{nat} \wedge 1) : \{\cdot; \cdot \vdash c : \mathsf{nat} \wedge 1\}}\ \mathsf{box}I$$

We also give a brief outline of the typing derivation for Example 2.1. We omit some typing annotations to save space and only recover them as necessary.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dots}{\cdot \mid x, y \Vdash x + y : \mathsf{nat}} \qquad \dfrac{}{\cdot \mid x, y; \cdot \vdash \mathsf{close}\ c :: (c : 1)}\ 1R}{\cdot \mid x, y; \cdot \vdash \mathsf{send}\ c\ x + y; \dots :: (c : \mathsf{nat} \wedge 1)}\ \wedge R}{\cdot \mid x : \mathsf{nat}, y : \mathsf{nat}; a : 1 \vdash \mathsf{wait}\ a; \dots :: c}\ 1L}{\cdot \mid x : \mathsf{nat}; a : \mathsf{nat} \wedge 1 \vdash y = \mathsf{recv}\ a; \dots :: c}\ \wedge L}{\cdot \mid \cdot \Vdash \mathsf{box}(x; a \vdash y = \mathsf{recv}\ a; \dots :: c) : \{x; a \vdash c\}}\ \mathsf{box}I \qquad \dfrac{\dfrac{\dfrac{\dots}{u \mid \cdot; a' \vdash u[5/x; a'/a] :: b}\ P_{\mathsf{var}} \qquad \dots}{u \mid \cdot; a' \vdash b \leftarrow u[5/x; a'/a] \leftarrow a' \mid \dots :: c}\ \mathsf{spawn}}{u \mid \cdot \Vdash \mathsf{box}(\cdot; a' \vdash \dots \mid \dots :: c) : \{\cdot; a' \vdash c\}}\ \mathsf{box}I}{\cdot \mid \cdot \Vdash \mathsf{letbox}\ u = \mathsf{box}(x; a \vdash y = \mathsf{recv}\ a; \dots :: c)\ \mathsf{in}\ \mathsf{box}(\cdot; a' \vdash \dots \mid \dots :: c) : \{\cdot; a' \vdash c\}}\ \mathsf{box}E$$

We point out that the premises in P_{var} that we omit check that u is a process code and that the substitutions $5/x$ and a'/a provide instantiations for the functional context $(x : \mathsf{nat})$ and channel context $(a : \mathsf{nat} \wedge 1)$ of u.

3.5 Operational Semantics and Type Preservation

We can now give a reduction semantics for our core functional terms using the judgment $M \Rightarrow_F M'$, and we write $\Rightarrow_F^*$ for its reflexive and transitive closure. Note that since process code is static for now, we have no operational semantics for the process layer.

The reduction rules for the simply-typed fragment of our functional layer is standard; we pick a call-by-value semantics for functions:

$$\dfrac{M \Rightarrow_F M'}{M\ N \Rightarrow_F M'\ N}\ \mathsf{app}_1 \qquad \dfrac{N \Rightarrow_F N'}{M\ N \Rightarrow_F M\ N'}\ \mathsf{app}_2 \qquad \dfrac{}{(\lambda x.M)\ v \Rightarrow_F [v/x]M}\ \mathsf{app}\beta$$

where v corresponds to values in the usual sense (variables, functions, and boxed processes).

When eliminating boxed processes via a letbox, we step the term until we see a box. After, we construct a metaobject corresponding to the underlying process code and substitute it for all occurrences of the free metavariable u. As a reminder, we omit the domain (the variable u) in the metasubstitution $[(\Gamma; \Delta.P :: c : C)]N$ since it can be inferred given that metalevel assumptions are ordered.

$$\frac{M \Rightarrow_F M'}{\mathsf{letbox}\ u = M\ \mathsf{in}\ N \Rightarrow_F \mathsf{letbox}\ u = M'\ \mathsf{in}\ N}\ \mathsf{lbox}$$

$$\frac{}{\mathsf{letbox}\ u = \mathsf{box}(\Gamma; \Delta \vdash P :: c : C)\ \mathsf{in}\ N \Rightarrow_F [(\Gamma; \Delta.P :: c : C)]N}\ \mathsf{lbox}\beta$$

Before we state type preservation, we introduce two key lemmas.

Lemma 3.1 (Term-level Substitution Property). *Let $\Sigma \mid \Gamma \vdash \sigma : \Gamma'$, then:*

- *If $\Sigma \mid \Gamma' \Vdash N : \tau'$, then $\Sigma \mid \Gamma \Vdash [\sigma]N : \tau'$.*
- *If $\Sigma \mid \Gamma'; \Delta \vdash P :: (c : C)$, then $\Sigma \mid \Gamma; \Delta \vdash [\sigma]P :: (c : C)$.*

The term-level substitution property states that both functional and process typing are preserved under some term-level substitution σ that provides instantiations for the appropriate functional context. Since our term-level substitutions are defined in the usual manner, the proof, which we omit, has no surprises. Metasubstitutions also enjoy an analogous property:

Lemma 3.2 (Metasubstitution Property). *Given some metasubstitution $\Sigma' \Vdash \theta : \Sigma$, the following hold:*

(1) *If $\Sigma \mid \Gamma \Vdash M : \tau$, then $\Sigma' \mid \Gamma \Vdash [\theta]M : \tau$.*
(2) *If $\Sigma \mid \Gamma; \Delta \vdash P :: (c : C)$, then $\Sigma' \mid \Gamma; \Delta \vdash [\theta]P :: (c : C)$.*
(3) *If $\Sigma \mid \Gamma \Vdash \sigma : \Gamma'$, then $\Sigma' \mid \Gamma \Vdash [\theta]\sigma : \Gamma'$.*

Its proof, just like for term-level substitutions, proceeds by simultaneous induction on the appropriate judgments for (1) to (3).

Theorem 3.3 (Preservation of functional layer). *If $\cdot \mid \cdot \Vdash M : \tau$ and $M \Rightarrow_F N$, then $\cdot \mid \cdot \Vdash N : \tau$*

Its proof proceeds by induction on the dynamics of the functional layer.

4 Running Processes and Operational Semantics

Since FuSes is a functional programming language first, we must give an interpretation of running process code within the functional layer. As demonstrated in Section 2.2, we do so by running the process and reifying the result of a single communication with the process. While we require process code to be closed with respect to its channel context so that we can execute it, it can contain free functional assumptions.

For instance, to run some process code of type $\{\Gamma; \cdot \vdash c : 1\}$, we can supply a substitution σ for the functional assumptions Γ. Since the offering protocol 1 is the trivial one where the process closes its offered channel, there is no meaningful output to bind. We nevertheless support running these processes for uniformity and to support future extensions to a process layer with effects.

$$\frac{\Sigma \mid \Gamma \Vdash N : \tau \quad \Sigma \mid \Gamma \Vdash M : \{\Gamma'; \cdot \vdash c : 1\} \quad \Sigma \mid \Gamma \Vdash \sigma : \Gamma'}{\Sigma \mid \Gamma \Vdash \mathsf{let}\ () = \mathsf{run}_1(M[\sigma])\ \mathsf{in}\ N : \tau}\ \mathsf{run}_1$$

Running a process of type $\{\Gamma'; \cdot \vdash c : \tau \wedge A\}$ results in binding a variable $x : \tau$ and a metalevel process variable $u : (\Gamma; \cdot \vdash c : A)$ to the value that is being sent and the continuation process, respectively. The continuation process no longer depends on the original functional context Γ' but instead on the context Γ of the substitution:

$$\frac{\Sigma, u : (\Gamma; \cdot \vdash c : A) \mid \Gamma, x : \tau \Vdash N : \tau' \quad \Sigma \mid \Gamma \Vdash M : \{\Gamma'; \cdot \vdash c : \tau \wedge A\} \quad \Sigma \mid \Gamma \Vdash \sigma : \Gamma'}{\Sigma \mid \Gamma \Vdash \mathsf{let}\ (x, u) = \mathsf{run}_\wedge(M[\sigma])\ \mathsf{in}\ N : \tau'}\ \mathsf{run}_\wedge$$

Processes of type $\{\Gamma'; \cdot \vdash c : \tau \supset A\}$ receive a value of type τ and then offer a channel of type A. We model this receiving of $x : \tau$ by making it free in the resulting continuation process u of type $(\Gamma, x : \tau; \cdot \vdash c : A)$. Intuitively, the process will "receive" a value for x when N runs u with an instantiation for x.

$$\frac{\Sigma, u : (\Gamma, x : \tau. \cdot \vdash c : A) \mid \Gamma \Vdash N : \tau' \quad \Sigma \mid \Gamma \Vdash M : \{\Gamma'; \cdot \vdash c : \tau \supset A\} \quad \Sigma \mid \Gamma \Vdash \sigma : \Gamma'}{\Sigma \mid \Gamma \Vdash \mathsf{let}\ u = \mathsf{run}_\supset(M[\sigma])\ \mathsf{in}\ N : \tau'}\ \mathsf{run}_\supset$$

Unlike in the examples, the subscripts in the run primitive denote the session type of the output and serve to syntactically distinguish each run construct to simplify the metatheory. Since process typing can be inferred, we may still adopt the generic `let _ = run(-)` syntax in an implementation.

4.1 Process Configuration

To describe the operational semantics of FuSes, we first introduce a multiset rewriting semantics for processes.

A configuration Ω is a multiset consisting of three predicates $\mathsf{proc}(\Delta; P; c)$, $\mathsf{sproc}(\Delta; P; c)$, and $\mathsf{code}(P; c)$. Unlike in other multiset formulations of session-typed processes, we track all of P's client channels and types Δ_P which is needed to support code suspension.

Intuitively, $\mathsf{proc}(\Delta; P; c)$ refers to a *live* process P offering a channel c using client channels Δ. The remaining two predicates $\mathsf{sproc}(\Delta; P; c)$ and $\mathsf{code}(P; c)$ refer to *currently suspending* and *suspended* processes respectively, which are used to capture the semantics of FuSes's run primitive. Suspended processes do not contain Δ because we ensure that they become closed. Configuration typing $\models \Omega :: \Delta$ asserts that the configuration Ω provides channels Δ and is given by the following rules:

$$\frac{}{\models \cdot :: \cdot}\ \Omega. \qquad \frac{\cdot \mid \cdot; \Delta_P \vdash P :: (c : C) \quad \models \Omega :: \Delta, \Delta_P}{\models \mathsf{proc}(\Delta_P; P; c), \Omega :: c : C, \Delta}\ \Omega_{\mathsf{proc}}$$

The two remaining predicates sproc and code are typed analogously to proc; the distinctions between the three predicates are only relevant in the operational semantics;

$$\frac{\cdot \mid \cdot; \Delta_P \vdash P :: (c : C) \quad \models \Omega :: \Delta, \Delta_P}{\models \mathsf{sproc}(\Delta_P; P; c), \Omega :: c : C, \Delta}\ \Omega_{\mathsf{sproc}} \qquad \frac{\cdot \mid \cdot; \cdot \vdash P :: (c : C) \quad \models \Omega :: \Delta}{\models \mathsf{code}(P; c), \Omega :: c : C, \Delta}\ \Omega_{\mathsf{code}}$$

As is typical, we require offered channel names to appear uniquely in a given multiset.

4.2 Operational Semantics and Type Preservation

We can now give a semantics for our process configurations $\Omega \Rightarrow_P \Omega'$, and we write $\Rightarrow_P^*$ for its reflexive and transitive closure. The multiset rewriting semantics for our process configuration is standard aside from how we deal with process suspension. We give two cases as examples.

If a process is sending across some channel a, and another process is receiving across the same channel a, then they can communicate:

$$\frac{M \Rightarrow_F^* v \quad v\ \mathsf{value}}{\mathsf{proc}(\Delta_1, c{:}\tau \wedge A; (x = \mathsf{recv}\ c; P); a), \mathsf{proc}(\Delta_2; (\mathsf{send}\ c\ M; Q); c) \ \Rightarrow_P\ \mathsf{proc}(\Delta_1, c{:}A; ([v/x]P); a), \mathsf{proc}(\Delta_2; Q; c)}$$

If a process is spawning a subprocess, then the result will be two process predicates:

$$\mathsf{proc}(\Delta_P \bowtie \Delta_Q; (a{:}A \leftarrow P \leftarrow \Delta_P \mid Q); c) \ \Rightarrow_P\ \mathsf{proc}(\Delta_Q; Q; c), \mathsf{proc}(\Delta_P; P; a) \quad (a\ \text{fresh})$$

These standard rules do not reference non-live processes sproc and code; we will come back to these predicates later.

4.2.1 Running Processes. When running processes from the functional layer, we first reduce the underlying term until we reach a box:

$$\frac{M \Rightarrow_F M'}{\mathsf{let}\ () = \mathsf{run}_1(M[\sigma])\ \mathsf{in}\ N \Rightarrow_F \mathsf{let}\ () = \mathsf{run}_1(M'[\sigma])\ \mathsf{in}\ N}\ \mathsf{run}_1$$

Each of the other run primitives has an analogous congruence rule.

We then defer to the operational semantics of the process layer by constructing a singleton configuration containing the process and run it until it attempts to communicate on its offered channel. For run_1, this is simple:

$$\frac{\mathsf{proc}(\cdot; P[\sigma]; c)\ \Rightarrow_P^*\ \mathsf{proc}(\cdot; \mathsf{close}\ c; c)}{\mathsf{let}\ () = \mathsf{run}_1(\mathsf{box}(\Gamma'; \cdot \vdash P :: c : 1)[\sigma])\ \mathsf{in}\ N \Rightarrow_F N}\ \mathsf{run}_1\beta$$

For processes offering more complex session types however, we must capture appropriate values and continuation processes depending on the session offered by the process. For instance, consider running some process P of type $\tau \wedge A$. We expect its singleton configuration to reach some multiset of the form $\mathsf{proc}(\Delta; (\mathsf{send}\ c\ M; Q); c), \Omega$, where Ω may contain other processes that were spawned by that execution. Since Q depends on processes in Ω, we must capture Ω (i.e., a process environment) so that we can continue running the process.

To keep the theory concise, we aim to capture the environment as process code. To do that, we signal $\mathsf{proc}(\Delta; (\mathsf{send}\ c\ M; Q); c)$ to suspend, which we model by replacing it with $\mathsf{sproc}(\Delta; Q; c)$. Note that when suspending, we have peeled off the action $\mathsf{send}\ c\ M$ to model the fact that we have observed this communication (in this case, the value to which M evaluates).

The idea then is that the semantics would force the entire configuration $\mathsf{sproc}(\Delta; Q; c), \Omega$ to reduce to some $\mathsf{code}(Q'; c)$, i.e., a closed process that represents Q and its environment Ω, that the functional layer can continue to run.

$$\frac{\mathsf{proc}(\cdot; P[\sigma]; c)\ \Rightarrow_P^*\ \mathsf{proc}(\Delta; (\mathsf{send}\ c\ M; Q); c), \Omega \quad M \Rightarrow_F^* v \quad \mathsf{sproc}(\Delta; Q; c), \Omega\ \Rightarrow_P^*\ \mathsf{code}(Q'; c)}{\mathsf{let}\ (x, u) = \mathsf{run}_\wedge(\mathsf{box}(\Gamma'; \cdot \vdash P :: c : \tau \wedge C)[\sigma])\ \mathsf{in}\ N \Rightarrow_F [(\cdot; \cdot.Q' :: c : C)][v/x]N}\ \mathsf{run}_\wedge\beta$$

The semantics for running a process of type $\tau \supset A$ proceeds in a similar manner.

$$\frac{\mathsf{proc}(\cdot; P[\sigma]; c)\ \Rightarrow_P^*\ \mathsf{proc}(\Delta; (x = \mathsf{recv}\ c; Q); c), \Omega \quad \mathsf{sproc}(\Delta; Q; c), \Omega\ \Rightarrow_P^*\ \mathsf{code}(Q'; c)}{\mathsf{let}\ u = \mathsf{run}_\supset(\mathsf{box}(\Gamma'; \cdot \vdash P :: c : \tau \supset C)[\sigma])\ \mathsf{in}\ N \Rightarrow_F [(x : \tau; \cdot.Q' :: c : C)]N}\ \mathsf{run}_\supset\beta$$

The multiset rewriting semantics involving process suspension is quite simple — a *currently suspending* process (sproc) will change any *live* client processes (proc) to *currently suspending* processes:

$$\mathsf{sproc}(\Delta; P; c), \mathsf{proc}(\Delta'; Q; a)\ \Rightarrow_P\ \mathsf{sproc}(\Delta; P; c), \mathsf{sproc}(\Delta'; Q; a) \quad (\text{if } a : - \in \Delta)$$

A *currently suspending* process will become a *suspended* process once all of its clients have suspended. In doing so, it composes all of its free channels with the code of its clients through a sequence of spawns.

$$\mathsf{sproc}(a_1{:}A_1, \ldots, a_n{:}A_n; P; c), \mathsf{code}(P_1; a_1), \ldots, \mathsf{code}(P_n; a_n)\ \Rightarrow_P\ \mathsf{code}\left(c, \begin{array}{l} \quad a_1{:}A_1 \leftarrow P_1 \leftarrow \cdot \\ \mid\ \ldots \\ \mid a_n{:}A_n \leftarrow P_n \leftarrow \cdot \\ \mid P \end{array}\right)$$

Note that a *currently suspending* process without any client channels will immediately become code according to this rule.

4.2.2 Type Preservation. Since we extend our functional language with run primitives, we extend our definition of term-level substitutions and metasubstitutions accordingly. Proving the corresponding substitution lemmas with the extended definitions poses no technical challenges.

The major change in type preservation comes from the fact that processes are no longer static. Our type preservation statement therefore also states that a well-typed configuration stays well typed:

Theorem 4.1 (Preservation of functional and process layer).

(1) *If* $\cdot \mid \cdot \Vdash M : \tau$ *and* $M \Rightarrow_F N$*, then* $\cdot \mid \cdot \Vdash N : \tau$.
(2) *If* $\models \Omega :: \Delta$ *and* $\Omega \Rightarrow_P \Omega'$*, then* $\models \Omega :: \Delta$.

The proof proceeds by simultaneous induction on the respective dynamics and highlights the advantage of our modular design. Most functional and process reduction cases are easy since they make no mention of the other layer. The cases where they do mention each other, such as $\mathsf{run}_\wedge\beta$, follow from an appeal to the induction hypothesis of the other layer.

5 Type-Safe Recursive Code Analysis

FuSes as presented thus far allows type-safe process code generation and observing the behavior of processes on the functional layer. However, we also want programmers to be able to optimize the process code that they have generated using domain-specific knowledge. For this reason, we add to FuSes the ability to recursively traverse and analyze process code in a type-safe manner as in Section 2.3.

Recall that $\mathrm{box}(\Gamma; \Delta \vdash P :: c : C)$ describes a session-typed process P that may refer to the functional context Γ, the input channels Δ, and the output channel $c : C$. As we recursively analyze P, both our functional context and our input channel context may change. Furthermore, the type of our output channel C does not remain fixed. We therefore must abstract over the functional context $\Gamma : \mathrm{ctx}$, the linear channel context $\Delta : \mathrm{lctx}$, and the offering channel's session type $C : \mathrm{stp}$ to support pattern matching on process code; hence function types turn into dependently typed functions. We force these abstractions over metaassumptions to only appear at the top-level to simplify later metatheoretic results; in principle we do not see any problems removing this restriction.

These functions are defined by a collection of branches where we simultaneously pattern match on all arguments to capture their dependencies. We hence consider *pattern spines*, i.e., a list of patterns, which we denote with θ. For example, when we pattern match on a process code of the form $\mathrm{box}(\Gamma; \Delta \vdash P :: c : C)$, we will also pattern match on Γ, Δ, and C. The particular pattern process P that we consider may also refine the functional context $\Gamma : \mathrm{ctx}$, the linear channel context $\Delta : \mathrm{lctx}$, or the offering channel's session type $C : \mathrm{stp}$. We describe a pattern spine using θ together with the pattern variables (metalevel assumptions) in Σ. Hence, a branch takes the form $\Sigma.\theta \mapsto M$ where M is the body. We therefore extend our metalevel assumptions and objects to make contexts, types, and open functional code first class in order to support code analysis:

Session Types	$A, B ::= 1 \mid \tau \wedge A \mid \tau \supset A \mid \alpha$
Functional Types	$\tau_1, \tau_2 ::= \tau_1 \rightarrow \tau_2 \mid \{\Gamma; \Delta \vdash a : A\} \mid \beta$
Metalevel assumptions	$\mathcal{S} ::= (\Gamma; \Delta \vdash a : A) \mid \mathsf{ctx} \mid \mathsf{lctx} \mid \mathsf{stp} \mid * \mid (\Gamma \vdash \tau)$
Metalevel objects	$\mathcal{N} ::= (\Gamma; \Delta . P :: c : C)] \mid \Gamma \mid \Delta \mid A \mid \tau \mid (\hat{\Gamma} . M)$
Generalized Linear Context	$\Delta ::= (\vec{\delta}; \mathcal{D})$
Set of Linear Assumptions	$\mathcal{D} ::= \cdot \mid \mathcal{D}, a : A$
Functional Context	$\Gamma ::= \cdot \mid \Gamma, x : \tau \mid \gamma$
Global metacontext	$\Sigma ::= \cdot \mid \Sigma, \phi : \mathcal{S}$
Pattern Spine	$\theta ::= \cdot \mid \theta, \mathcal{N}$
Branches	$\mathcal{B} ::= \Sigma . \theta \mapsto M$
Top-level Function Type Sig.	$\mathcal{T} ::= \cdot \mid \mathcal{T}, f : \Pi \Sigma . \tau$
Top-level Function Def. Sig.	$\mathcal{R} ::= \cdot \mid \mathcal{R}, f ::= \{\vec{\mathcal{B}}\}$

Types lctx and ctx correspond to linear channel contexts and functional contexts, stp and $*$ to session types and functional types, and $(\Gamma \vdash \tau)$ to an expression code of type τ that depends on Γ. Since we now have different metalevel assumptions, we will use the name ϕ to refer to a metalevel assumption in Σ. For readability, we continue to refer to metaassumptions denoting a process by u, v and introduce analogous conventions for metaassumptions denoting linear contexts by δ, functional contexts by γ, session types by α, functional types by β, and functional terms with r.

As shown above, we extend our now first-class objects so that they can refer to metaassumptions. Linear contexts must now allow for multiple metavariables to enable us to capture context splits during pattern matching. This is in contrast to functional contexts that, due to weakening, are never split, and so use at most one metavariable γ.

We therefore redefine the linear context Δ as a pair consisting of a list of linear context variables $\vec{\delta} = \delta_1, \ldots, \delta_n$ and a set of concrete channel name assumptions $\mathcal{D} = a_1 : A_1, \ldots, a_m : A_m$. This new definition simplifies later metatheory because we require $\vec{\delta}$ to be ordered while $\mathcal{D}$ can remain unordered. Changing process typing rules to accommodate this new definition of Δ is purely mechanical; left rules now refer to some channel in $\mathcal{D}$, any rules that require the context to be empty now also require $\vec{\delta}$ to be empty, and context merge concatenates the context variables and channel names pointwise. We continue to informally use commas to extend linear contexts.

We define top-level function types in a signature $\mathcal{T}$. Note that we only allow for dependent types on the top-level. The corresponding function definitions are in $\mathcal{R}$. As branches can refer to pattern variables, we must also extend the functional layer with the ability to refer to them. We can splice a metavariable r representing some open functional code through the syntax $r[\sigma]$ where σ is an appropriate term-level substitution. This is analogous to splicing $u[\sigma; \rho]$ for the process layer and is typed by:

$$\frac{r : (\Gamma' \vdash \tau) \in \Sigma \quad \Sigma \mid \Gamma \Vdash \sigma : \Gamma'}{\Sigma \mid \Gamma \Vdash r[\sigma] : \tau} \; M_{\mathsf{var}}$$

Before we explain top-level functions in more detail, we define metasubstitutions in FuSes with the extended metalevel objects.

5.1 Metasubstitution

Metasubstitutions $[\theta](-)$ simultaneously replace all occurrences of metavariables in $(-)$ by the objects in θ. We type metasubstitutions analogously to substitutions in dependently typed systems:

$$\frac{}{\Sigma \Vdash \cdot : \cdot}\ \theta_{\mathsf{empty}} \qquad \frac{\Sigma \Vdash \theta : \Sigma' \quad \Sigma \Vdash \mathcal{N} : [\theta]\mathcal{S}}{\Sigma \Vdash (\theta, \mathcal{N}) : (\Sigma', \phi : \mathcal{S})}\ \theta_{\mathsf{dec}}$$

The premise $\Sigma \Vdash \mathcal{N} : [\theta]\mathcal{S}$ defers to the appropriate judgments for each kind of $\mathcal{N}$. For instance, if $\mathcal{S} = \mathsf{lctx}$, then we check that $\mathcal{N}$ is a linear context according to its formation rules:

$$\frac{\Sigma \vdash \Delta\ \mathsf{lctx}}{\Sigma \Vdash \Delta : \mathsf{lctx}}\ \mathcal{N}_{\mathsf{lctx}}$$

Next, to define the metasubstitution operation $[\theta](-)$, we assume a lookup function

$$\theta(\phi) = \mathcal{N}$$

that finds a metaobject $\mathcal{N} \in \theta$ that corresponds to the metavariable ϕ in the domain of the substitution. Formally, it must satisfy the following:

Property 1 (Lookup Property). *If $\Sigma' \Vdash \theta : \Sigma$, then for all $\phi : S \in \Sigma$, there exists $\theta(\phi)$ such that $\Sigma' \Vdash \theta(\phi) : [\theta]S$.*

Because metacontexts Σ are ordered, we keep the domain of θ implicit for economical reasons and assume that it can be recovered when needed. In the following sections, we may sometimes make the domain of metasubstitutions explicit for clarity.

5.1.1 Types and Contexts. Metasubstitutions over types and contexts are mostly defined in the obvious manner. When encountering a type variable, we look it up, and when encountering other type constructors, we recursively apply the substitution to its components. For instance, we define metasubstitutions over session types as follows:

$$[\theta]\alpha \equiv \theta(\alpha) \qquad [\theta](\tau \wedge B) \equiv [\theta]\tau \wedge [\theta]B \qquad [\theta](\tau \supset B) \equiv [\theta]\tau \supset [\theta]B$$

Unfortunately, naïvely following this approach for linear contexts is unsound since they may contain repetitions of the same context variables. For instance, consider the metasubstitution $[(a : 1)/\delta](\delta, \delta; \cdot)$, where a naïve definition would result in an ill-formed context $(\cdot; a : 1, a : 1)$. Preventing these kinds of contexts is non-trivial, since a seemingly innocent metasubstitution such as $[\delta/\delta_1, \delta/\delta_2](\delta_1, \delta_2; \cdot)$ would result in a linear context with repeated context variables. Furthermore, the ability to describe a context with repeated context variables is useful. For example, we can write a metaprogram that takes some process code as input and constructs code that runs two copies of it in parallel.

Resolving these name conflicts through renaming individual channels is not a difficult task. However, this renaming must be applied consistently and therefore cannot be a purely local operation. In particular, for cases such as $[\theta]\mathrm{box}(\cdot; \Delta \vdash P :: c : C)$, we have to ensure that channel names in Δ remain in sync with P, meaning whatever renaming we choose for Δ must be applied to P.

We therefore declaratively assume an (ordered) list of renamings $\vec{\rho}$ which provides instantiations for the result of looking up each occurrence of a linear contextual variable in the domain such that no name conflicts occur, which we make explicit by extending the metasubstitution operator, $[\theta]_{\vec{\rho}}(-)$ for linear contexts. More precisely, we require $\vec{\rho}$ to

(1) $\Delta_i \vdash \rho_i : \theta(\delta_i)$ for all occurrences of a context variable δ_i in the domain of the metasubstitution, i.e., there must be a corresponding channel renaming for every occurrence of each context variable;

(2) cause no name conflicts when merged together, i.e., $\bigcap \mathsf{dom}(\rho_i) = \{\}$.

With this, we define metasubstitution for an empty linear context and channel extension in the obvious manner:

$$[\theta]_{(\cdot)} \cdot \equiv \cdot \quad [\theta]_{\vec{\rho}}(\vec{\delta}; \mathcal{D}, a : A) \equiv [\theta]_{\vec{\rho}}(\vec{\delta}; \mathcal{D}, a : [\theta]A)$$

The interesting case is when the linear context is of the form $\vec{\delta}, \delta'; \mathcal{D}$. Here, we perform a lookup on δ' and then apply the last renaming to its result:

$$[\theta]_{(\vec{\rho}, \rho')}(\vec{\delta}, \delta'; \mathcal{D}) \equiv [\theta]_{\vec{\rho}}(\vec{\delta}; \mathcal{D}) \bowtie [\rho']\theta(\delta')$$

Applying the respective renamings after each lookup ensures that the resulting context is well-formed by the second condition on $\vec{\rho}$.

5.1.2 Terms. Metasubstitutions on terms is mostly standard:

$$[\theta]r[\sigma] \equiv [[\theta]\sigma]\theta(r) \qquad [\theta]x \equiv x \qquad [\theta]\lambda x.M \equiv \lambda x.[\theta]M \qquad [\theta](M\ N) \equiv ([\theta]M)\ ([\theta]N)$$

One notable point is that when applying a metasubstitution to some code splicing $r[\sigma]$, we must not only perform a lookup on r but also continue recursively into the term-level substitution σ since they may also depend on metaassumptions.

For boxes, we declaratively assume a renaming $\vec{\rho}$ and apply it to both the binding linear context Δ and the process P; this ensures that names remain in sync after the metasubstitution:

$$[\theta]\mathsf{box}(\Gamma; \Delta \vdash P :: c : C) \equiv \mathsf{box}([\theta]\Gamma; [\theta]_{\vec{\rho}}\Delta \vdash [\theta]_{\vec{\rho}}P :: c : [\theta]C)$$

Since letbox $u = M$ in N binds a fresh metavariable u to the continuation term N, we must extend the metasubstitution with the identity substitution for u:

$$[\theta](\mathsf{letbox}\ u = M\ \mathsf{in}\ N) \equiv \mathsf{letbox}\ u = [\theta]M\ \mathsf{in}\ [\theta, u]N$$

And finally, metasubstitution for top-level function calls simply applies the substitution to the arguments.

$$[\theta](f[\theta']) \equiv f[[\theta]\theta']$$

5.1.3 Processes. Similarly to linear contexts, we must keep track of instantiations of each occurrence of context variables $\vec{\rho}$. Process terms that perform some communication over some channel pose no difficulties; we recursively apply the metasubstitution to its relevant components in the obvious manner.

$$[\theta]_{(\cdot)}\mathsf{fwd}\ a\,b \equiv \mathsf{fwd}\ a\,b \qquad [\theta]_{(\cdot)}\mathsf{close}\ a \equiv \mathsf{close}\ a \qquad [\theta]_{\vec{\rho}}(\mathsf{wait}\ a; P) \equiv \mathsf{wait}\ a; [\theta]_{\vec{\rho}}P$$

$$[\theta]_{\vec{\rho}}(\mathsf{send}\ a\ M; P) \equiv \mathsf{send}\ a\ [\theta]M; [\theta]_{\vec{\rho}}P \qquad [\theta]_{\vec{\rho}}(x = \mathsf{recv}\ a; P) \equiv x = \mathsf{recv}\ a; [\theta]_{\vec{\rho}}P$$

Since we ensure by typing that the two axioms fwd $a\,b$ and close a cannot depend on any context variables, the renaming $\vec{\rho}$ should also be empty, as denoted by $[\theta]_{(\cdot)}$.

When encountering a spawn, we must split our instantiations $\vec{\rho}$ accordingly. Note that while this split is written declaratively, it is very easy to compute in practice given that both the list of context variables within Δ and the list of instantiations $\vec{\rho}$ are ordered and required to "match."

$$[\theta]_{(\vec{\rho_P}, \vec{\rho_Q})}(a{:}A \leftarrow P \leftarrow \Delta \mid Q) \equiv a{:}[\theta]A \leftarrow [\theta]_{\vec{\rho_P}}P \leftarrow [\theta]_{\vec{\rho_P}}\Delta \mid [\theta]_{\vec{\rho_Q}}Q$$

Finally, splicing some process code $u[\sigma; \rho]$ requires some care. We must perform a lookup $\theta(u)$ and then apply the term substitution $[\theta]\sigma$ which we obtain by recursively applying the metasubstitution θ over σ.

$$[\theta]_{(\cdot)}u[\sigma; \rho] \equiv [\rho][[\theta]\sigma]\theta(u)$$

5.1.4 Term Substitutions and Metasubstitutions. Applying metasubstitutions over both term substitutions and metasubstitutions, which are both essentially lists of terms and metaobjects, simply amounts to traversing the list and applying the metasubstitutions in order. When traversing metasubstitutions, we must carry the renaming $\vec{\rho}$ since they could carry process code that need to remain in sync.

$$[\theta]\cdot \equiv \cdot \qquad [\theta](\sigma, M/x) \equiv [\theta]\sigma, ([\theta]M)/x$$

$$[\theta]_{\vec{\rho}}\cdot \equiv \cdot \qquad [\theta]_{\vec{\rho}}(\theta', \mathcal{N}) \equiv [\theta]_{\vec{\rho}}\theta', [\theta]_{\vec{\rho}}\mathcal{N}$$

5.2 Type-Checking Top-Level Function Definitions

We finally define the typing rules for top-level function definitions. We assume a signature $\mathcal{R}$ containing top-level function definitions and a fixed $\mathcal{T}$ containing type signatures of these top-level functions. For each function f in $\mathcal{R}$, we look up its corresponding type in $\mathcal{T}$. We then check that each branch of the function definition satisfies the specified type:

$$\frac{}{\Vdash \cdot \text{ wf}}\ \mathcal{R}_{\text{empty}} \qquad \frac{\Vdash \mathcal{R} \text{ wf} \quad f : \Pi\Sigma.\tau \in \mathcal{T} \quad \Vdash \mathcal{B}_i : \Pi\Sigma.\tau \text{ for all } \mathcal{B}_i \in \mathcal{B}}{\Vdash \mathcal{R}, f ::= \{\vec{\mathcal{B}}\} \text{ wf}}\ \mathcal{R}_{\text{dec}}$$

We type each branch $\Sigma_i.\theta_i \mapsto M_i$ by checking that M_i is well-typed under the metalevel assumptions Σ_i (i.e., the pattern variables) introduced in the pattern θ_i. We require θ_i to be a metasubstitution that maps the branch's metacontext Σ_i to the metaassumptions in the function's signature Σ:

$$\frac{\Sigma_i \Vdash \theta_i : \Sigma \quad \Sigma_i \mid \cdot \Vdash M_i : [\theta_i]\tau}{\Vdash \Sigma_i.\theta_i \mapsto M_i : \Pi\Sigma.\tau}\ \mathcal{B}$$

At this point, we do not consider any kind of coverage checking for process code patterns. We believe that this can be done following the ideas for coverage on contextual objects described in [Dunfield and Pientka 2008; Pientka and Abel 2015] and extending them to the linear setting.

We call these top-level functions from the functional layer using the syntax $f[\theta]$ where θ is a metasubstitution that instantiates the metacontext Σ' of the function definition.

$$\frac{\mathcal{T}(f) = \Pi\Sigma'.\tau \quad \Sigma \Vdash \theta : \Sigma'}{\Sigma \mid \Gamma' \Vdash f[\theta] : [\theta]\tau}\ \text{def}E$$

To reduce a top-level function call $f[\theta]$, we identify a particular branch $\Sigma_i.\theta_i \mapsto B_i$ alongside a substitution θ' that factors $\theta = \theta_i[\theta']$, and then we apply the metasubstitution θ' to M_i:

$$\frac{f ::= \{\vec{\mathcal{B}}\} \in \mathcal{R} \quad \mathcal{B}_i = \Sigma_i.\theta_i \mapsto M_i \quad \theta = \theta_i[\theta']}{f[\theta] \Rightarrow_F [\theta']M_i}\ \text{call}$$

For simplicity, we leave the runtime details of top-level functions open-ended. In particular, FuSes does not enforce that a given spine covers all patterns nor that branches do not overlap with each other.

5.3 Example

We revisit the deep copy example in Section 2.3. We first require that its type signature appear in $\mathcal{T}$, i.e.,

$$\mathcal{T}(\text{copy}) = \Pi(\gamma{:}\text{ctx}, \delta{:}\text{lctx}, C{:}\text{stp}, u : (\gamma; \delta \vdash c : C)).\{\gamma; \delta \vdash c : C\}.$$

We then check each of its definitions in $\mathcal{R}$ via $\mathcal{R}_{\text{dec}}$. For example, the base case of the copy function is given by the following:

$$\gamma{:}\text{ctx}.(\gamma, \cdot, 1, (\gamma; \cdot.\text{close } c :: c : 1)) \mapsto \text{box}(\gamma; \cdot \vdash \text{close } c :: c : 1)$$

where the singleton γ:ctx is its metalevel context, and the metasubstitution $\theta = (\gamma, \cdot, 1, (\gamma; \cdot.\mathsf{close}\ c :: c : 1))$ is viewed as the pattern.

This branch must then be checked against the type signature of copy:

$$\frac{\gamma{:}\mathsf{ctx} \Vdash \theta : (\gamma{:}\mathsf{ctx}, \delta{:}\mathsf{lctx}, C{:}\mathsf{stp}, u : (\gamma; \delta \vdash c : C)) \quad \gamma{:}\mathsf{ctx} \mid \cdot \Vdash \mathsf{box}(\gamma; \cdot \vdash \mathsf{close}\ c :: c : 1) : [\theta]\{\gamma; \delta \vdash c : C\}}{\Vdash \gamma{:}\mathsf{ctx}.\theta \mapsto \mathsf{box}(\gamma; \cdot \vdash \mathsf{close}\ c :: c : 1) : \Pi(\gamma{:}\mathsf{ctx}, \delta{:}\mathsf{lctx}, C{:}\mathsf{stp}, u : (\gamma; \delta \vdash c : C)).\{\gamma; \delta \vdash c : C\}}\ \mathcal{B}$$

The first premise type checks θ, which ensures that the pattern is valid, and the second premise checks that the body is well-typed with respect to the metasubstitution θ applied to the signature of copy. That is to say, $[\theta]\{\gamma; \delta \vdash c : C\} = \{\gamma; \cdot \vdash c : 1\}$, so we continue type checking the body against the type $\{\gamma; \cdot \vdash c : 1\}$.

The rule def$E$ handles any top-level function calls (in this case, a recursive call), such as

$$\gamma{:}\mathsf{ctx}, \delta{:}\mathsf{ctx}, C{:}\mathsf{stp}, u : (\gamma; \delta \vdash c : C).(\gamma, \delta, a : 1, C, (\gamma; \delta, a : 1.\mathsf{wait}\ a; u :: c : C))$$
$$\mapsto \mathsf{letbox}\ v = \mathsf{copy}[\gamma, \delta, C, u]\ \mathsf{in}\ \ldots$$

Type checking this branch would lead to a subderivation of the form:

$$\frac{\dfrac{\Sigma \Vdash (\gamma, \delta, C, u) : \gamma{:}\mathsf{ctx}, \delta{:}\mathsf{lctx}, C{:}\mathsf{stp}, u : (\gamma; \delta \vdash c : C)}{\Sigma \mid \cdot \Vdash \mathsf{copy}[\gamma, \delta, C, u] : [\gamma, \delta, C, u]\{\gamma; \delta \vdash c{:}C\}}\ \mathsf{def}E \quad \Sigma, v : (\gamma; \delta \vdash c : C) \mid \cdot \Vdash \ldots : \{\gamma; \delta, a{:}1 \vdash c{:}C\}}{\Sigma \mid \cdot \Vdash \mathsf{letbox}\ v = \mathsf{copy}[\gamma, \delta, C, u]\ \mathsf{in}\ \ldots : \{\gamma; \delta, a : 1 \vdash c : C\}}\ \mathsf{box}E$$

where $\Sigma = \gamma{:}\mathsf{ctx}, \delta{:}\mathsf{ctx}, C{:}\mathsf{stp}, u : (\gamma; \delta \vdash c : C)$. The premise in def$E$ ensures that the arguments to copy is a valid metasubstitution.

5.4 Metasubstitution Lemma and Type Preservation

The extensions that we present in this section mostly deals with the metasubstitution, and thus, our metasubstitution lemma becomes more complex; our term-level substitution property and type preservation remain mostly as is.

Lemma 5.1 (Metasubstitution Property for types and contexts). *Given some metasubstitution $\Sigma' \Vdash \theta : \Sigma$, the following hold:*

(1) *If* $\Sigma \vdash A$ stp, *then* $\Sigma' \vdash [\theta]A$ stp.
(2) *If* $\Sigma \vdash \tau\ *$, *then* $\Sigma' \vdash [\theta]\tau\ *$.
(3) *If* $\Sigma \vdash \Delta$ lctx, *then* $\Sigma' \vdash [\theta]_{\vec{\rho}}\Delta$ lctx *for some* $\vec{\rho}$.
(4) *If* $\Sigma \vdash \Gamma$ ctx, *then* $\Sigma' \vdash [\theta]\Gamma$ ctx.

Its proof proceeds by simultaneous induction on the appropriate formation judgments. Most of the cases fall under a lookup case such as $[\theta]\alpha \equiv \theta(\alpha)$ or a recursive case such as $[\theta](\tau \wedge A) \equiv [\theta]\tau \wedge [\theta]A$. In the former, we refer to the lookup property 1 and in the latter, we rebuild the appropriate object after applying induction hypothesis on its sub-objects.

Lemma 5.2 (Metasubstitution Property for terms, processes, and substitutions). *Given some metasubstitution $\Sigma' \Vdash \theta : \Sigma$, the following hold:*

(1) *If* $\Sigma \mid \Gamma \Vdash M : \tau$, *then* $\Sigma' \mid [\theta]\Gamma \Vdash [\theta]M : [\theta]\tau$.
(2) *If* $\Sigma \mid \Gamma; \Delta \vdash P :: (c : C)$, *then* $\Sigma' \mid [\theta]\Gamma; [\theta]_{\vec{\rho}}\Delta \vdash [\theta]_{\vec{\rho}}P :: (c : [\theta]C)$ *for some* $\vec{\rho}$.
(3) *If* $\Sigma \mid \Gamma \Vdash \sigma : \Gamma'$, *then* $\Sigma' \mid [\theta]\Gamma \Vdash [\theta]\sigma : [\theta]\Gamma'$.
(4) *If* $\Sigma \Vdash \theta' : \Sigma''$, *then* $\Sigma' \Vdash [\theta]\theta' : \Sigma''$.

Note the use of the same renaming $\vec{\rho}$ in (2), which applies to both Δ and P. This ensures that the two remain in sync with regards to channel names after the metasubstitution operation. The proof proceeds by simultaneous induction on the appropriate typing judgments.

The only change in type preservation is the addition of a new reduction case, call, for the functional layer.

6 Implementation and Further Examples

To show the practicality of FuSes and to allow us to write larger examples, we have implemented a prototype for it [Sano et al. 2025]. In particular, our artifact implements type checking for FuSes and simulates its operational semantics.

To develop a usable front-end, we infer type annotations. For example, type annotations for contextual objects can be inferred, and similarly, empty contexts in the spawn construct are optional. We also add some basic primitives such as integers, Booleans, common binary operations, and conditionals.

Our type-checker uses a bidirectional typing approach $\Sigma \mid \Gamma \vdash e \Rightarrow \tau$ and $\Sigma \mid \Gamma \vdash e \Leftarrow \tau$ to type expressions. Since type annotations are optional, our process typing judgment $\Sigma \mid \Gamma ; \Delta \vdash P :: c : C$ also outputs some Γ', Δ', P', and C', potentially containing newly inferred typing information.

Our implementation can type-check and run all the examples in this section.

6.1 Further Examples

The examples that we present previously are mostly for illustrative purposes to demonstrate features of our language. In this section, we use the surface language syntax used in Section 2 to implement more interesting and realistic metaprogramming operations; as mentioned, these are all implemented (with some modifications) and can be ran with our prototype.

6.1.1 Batch Optimization. In concurrent programming, a batch optimization refers to "batching" sequences of message transmissions into one. In our setting, this amounts to converting processes of the form `send c M; send c N; ...` to `send c <M, N>; ...` where `<M, N>` is the usual functional product type. We define a function `batch:(C:stp, C':stp, u:(·;· ⊢ c:C)). {·;· ⊢ c:C'}` [1] that performs this transformation over the offering channel of an arbitrary process.

```
batch 1 1 (·;· → close c → c:1) ⇒
  box(·;· → close c → c:1)
% the interesting/main case
batch (t1 ∧ t2 ∧ C) ( (t1 X t2) ∧ C') (·;· → send c M; send c N; u → c:t1 ∧ t2 ∧ C) ⇒
  letbox u' = batch C C' (·;· → u → c:C) in
  box(·;· → send c <M, N>; u' → c: (t1 X t2) ∧ C')
% examples of recursive cases
batch (t1 ⊃ C) (t1 ⊃ C') (·;· → x = recv c; u → c:C) ⇒
  letbox u' = batch C C' u in
  box(·;· → x = recv c; u' → c:C')
batch (t1 ∧ C) (t1 ∧ C') (·;· → send c M; u → c:t1 ∧ C) ⇒
  letbox u' = batch C C' (·;· → u → c:C) in
  box (·;· → send c M; u' → c:t1 ∧ C')
```

A similar function can be defined to batch transform a client channel.

```
batch_c 1 1 C (·; a:1 → wait a; u → c:C) ⇒
  box(·; a:1 → wait a; u → c:C)

batch_c (t1 ∧ t2 ∧ A) ((t1 X t2) ∧ A') C (·;a:(t1 ∧ t2 ∧ A) → send a M; send a N; u → c:C) ⇒
  letbox u' = batch_c A A' C (·;a:A → u → c:C) in
  box(·; a:((t1 X t2) → A') → send a <M, N>; u' → c:C)
...
```

[1]We assume the process is closed to save space. Supporting arbitrary open processes simply amounts to abstracting over the appropriate context variables and passing them on during recursive calls.

As currently written, both of these functions only batch two consecutive sends into one; for instance, if we have a process of the form

```
  send c M1; send c M2; send c M3; send c M4; ...
```

applying the transformation gives a process

```
  send c <M1, M2>; send c <M3, M4>; ...
```

because the function would recur on the continuation process `send c M3; ...` after it batches the sending of `M1` and `M2` together.

Of course, for any fixed n, it is easy to manually write all pattern that batches $2, 3, \ldots, n$ consecutive sends, but more generally, one way to ensure that all consecutive sends are batched would be to modify the recursive call in the main case such that it recurs on the entire transformed process instead of just the continuation:

```
batch (t1 ∧ t2 ∧ C) ( (t1 X t2) ∧ C') (·;· ⟶ send c M; send c N; u ⟶ c:t1 ∧ t2 ∧ C) ⇒
  batch ((t1 X t2) ∧ C) C' (·;· ⟶ send c <M, N>; u ⟶ c:C')
```

If `C` is of the form `t3 ∧ C''`, then the function would batch the two consecutive sends of `t1 X t2` and `t3` together, and so on.

Although this would work in theory, it is questionable whether an implementation of FuSes can automatically infer that this modified pattern would be terminating if we choose to implement a coverage and termination checking for top-level functions.

A safer approach would be to call the `batch` function multiple times; since each application would group two consecutive sends into one, we would only need to call the function $\log(n)$ times where n is the maximum number of consecutive sends within the process. Unfortunately, due to the nature of our pattern matching, it does not seem possible to write a function that would, in general, find n; we have no way of encoding such properties that speak of "all channels" in the context. It would however be possible to find n for fixed channels such as the offering channel.

6.1.2 Run-Time Monitoring. In process calculi, monitors [Gommerstadt et al. 2018] refer to processes that interface between two communicating parties, checking run-time invariants of messages exchanged between the two. For instance, a monitor of type `{x:nat; a:nat ∧ 1 ⊢ c:nat ∧ 1}` could ensure that the natural number received from `a` is greater than `x`:

```
let mon = box(x:nat; a:nat ∧ 1 ⟶
              x' = recv a;
              if x' > x, send c x'; fwd a c
              else abort(...) ⟶ c:nat ∧ 1)
```

In general, suppose we implement a monitor of some arbitrary protocol `A` that ensures some invariant that we would like to enforce on all internally spawned processes. We can achieve this by traversing the process that we would like to modify and check every spawn:

```
insert_mon : (A:stp, mon:(·; a:A ⊢ c:A), δ:lctx, C:stp, u:(·;δ ⊢ c:C)). {·;δ ⊢ c:C}
% main case: insert mon in between the spawn since type matches
insert_mon A mon (δ1 ⋈ δ2) C (·;(δ1 ⋈ δ2) ⟶ (a:A ⟵ spawn u ⟵ δ1) | v ⟶ c:C) ⇒
  box(·;(δ1 ⋈ δ2) ⟶
      letbox u' = insert_mon A mon δ1 A (·;δ1 ⟶ u ⟶ a:A) in
      letbox v' = insert_mon C mon δ2 C (·;δ2 ⟶ v ⟶ c:C) in
      (a':A ⟵ spawn u' ⟵ δ1) |
      (a:A ⟵ mon ⟵ a':A) |
      v' ⟶ c:C)
...
```

6.1.3 Tracing. Another useful debugging mechanism is *tracing*, where we continuously log messages that a process receives that are relevant to its execution behavior. In our process calculus setting, this amounts to logging choices, which we discuss in Section 7.1.

The idea is that given a process, we extend its channel context with a logging channel `log` of type `string → string → ...`, traverse the process, and then insert code that sends appropriate messages to `log`. We assume a primitive `chan_str(-)` that stringifies channels and a string concatenation operator `++` to generate relevant logging messages. We also do not fix any implementation details of this logging channel — one can implement one that does anything with the data it receives.

```
% T:stp is the type of the trace monitor. It is of the form string -> ... -> 1
tracing : (T:stp, δ:lctx, C:stp, u:(·; δ⊢ c:C)). {·; δ, log:T ⊢ c:C}
% case when the offering channel c receives the choice
tracing T δ (A & B) (·; δ→ case c of (inl → u , inr → v) → c:A & B) ⇒
  letbox u' = tracing T δ A (·; δ → u → c:A) in
  letbox v' = tracing T δ B (·; δ → v → c:B) in
  box(·; δ, log:(string → T) →
    case c of (inl → send log (chan_str(c) ++ "inl"); u',
               inr → send log (chan_str(c) ++ "inr"); v') → c:A & B)
% case when some client a:A+B in our context receives the choice
tracing T (δ, a:(A+B)) C (·; δ, a:(A + B) → case a of (inl → u, inr → v) → c:C) ⇒
  letbox u' = tracing T (δ, a:A) C (·; δ, a:A → u → c:C) in
  letbox v' = tracing T (δ, a:B) C (·; δ, a:B → v → c:C) in
  box(·; δ, log:(string → T), a:(A+B)→
    case a of (inl → send log (chan_str(c) ++ "inl"); u',
               inr → send log (chan_str(c) ++ "inr"); v') → c:C)
% an example of an axiom/leaf case - T must be set to 1, and we must insert a 'wait' to
% satisfy typing
tracing 1 · 1 (·; · → close c → c:1) ⇒
  box(·; log:1 → wait log; close c → c:1)
...
```

7 Extending the Process Layer

In this paper we focus our attention to investigating the design and techniques needed to combine functional programming with process calculus while keeping the two as separate as possible. Thus, our paper works over a very simple functional layer consisting only of functions $\tau_1 \to \tau_2$ and also a process calculus layer consisting only of termination 1, value output $\tau \wedge A$, and value input $\tau \supset A$.

In this section we focus on extending our process layer with additional constructs as evidence of the modularity of FuSes's design.

7.1 Internal and External Choice

The internal and external choices $A \oplus B$ and $A \mathbin{\&} B$ respectively denote protocols where either the server or client chooses to commit to either of two protocols. Operationally, $A \oplus B$ corresponds to the server sending either a "left" or "right" label and then transitioning to the protocol A or B respectively. The client must therefore receive a label and perform a case analysis on it to determine which protocol to follow.

$$\frac{\Delta, a : A \vdash P :: c : C \quad \Delta, a : B \vdash Q :: c : C}{\Delta, a : A \oplus B \vdash \mathsf{case}\ a\ (P, Q) :: c : C}\oplus L \qquad \frac{\Delta \vdash P :: c : A}{\Delta \vdash c[\mathsf{inl}]; P :: c : A \oplus B}\oplus R_1 \qquad \frac{\Delta \vdash P :: c : B}{\Delta \vdash c[\mathsf{inr}]; P :: c : A \oplus B}\oplus R_2$$

The typing rules for $A \mathbin{\&} B$ follows by reversing the role of the client and the server — the client must send a label while the server must case on the label and proceed accordingly. Extending these typing rules to our system is trivial — we merely add the metacontext Σ, the functional context Γ, and account for the differing structure of our linear context Δ as in Section 5.

To support running processes of this type, we must interpret the result of communicating with these session types in the functional setting. We can interpret a communication with the internal choice $A \oplus B$ as a case statement — running a process of type $\{\Gamma'; \cdot \vdash c : A \oplus B\}$ will produce *either* a process of type $\{\Gamma; \cdot \vdash c : A\}$ or $\{\Gamma; \cdot \vdash c : B\}$ where Γ is the current context.

$$\frac{\begin{array}{l}\Sigma, u : (\Gamma; \cdot \vdash c : A) \mid \Gamma \Vdash N_1 : \tau \\ \Sigma, v : (\Gamma; \cdot \vdash c : B) \mid \Gamma \Vdash N_2 : \tau \quad \Sigma \mid \Gamma \Vdash M : \{\Gamma'; \cdot \vdash c : A \oplus B\} \quad \Sigma \mid \Gamma \Vdash \sigma : \Gamma'\end{array}}{\Sigma \mid \Gamma \Vdash \mathsf{case}\ \mathsf{run}_\oplus(M[\sigma])\{\mathsf{inl} \to u.N_1 \mid \mathsf{inr} \to v.N_2\} : \tau}\ \mathsf{run}_\oplus$$

Similarly, we can interpret the external choice $A \mathbin{\&} B$ as a pair — running a process of type $\{\Gamma'; \cdot \vdash c : A \mathbin{\&} B\}$ should produce two processes, one offering A and the other offering B:

$$\frac{\Sigma, u{:}(\Gamma; \cdot \vdash c : A), v{:}(\Gamma; \cdot \vdash c : B) \mid \Gamma \Vdash N : \tau \quad \Sigma \mid \Gamma \Vdash M : \{\Gamma'; \cdot \vdash c : A \mathbin{\&} B\} \quad \Sigma \mid \Gamma \Vdash \sigma : \Gamma'}{\Sigma \mid \Gamma \Vdash \mathsf{let}\ (u, v) = \mathsf{run}_\&(M[\sigma])\ \mathsf{in}\ N : \tau}\ \mathsf{run}_\&$$

Finally, extending metasubstitution for these additional constructs is simple. For types $A \oplus B$ and $A \mathbin{\&} B$, we recursively apply the metasubstitution to their components A and B, and for processes $a[\mathsf{inl}]; P$, $a[\mathsf{inr}]; P$, and case a $(P,\ Q)$, we again recursively apply the metasubstitution to its component processes. Importantly, since both P and Q must use the same linear context in case a $(P,\ Q)$, we do not split our instantiations $\vec{\rho}$.

7.2 Channel Output and Input

Channel output and input $A \otimes B$ and $A \multimap B$ respectively denote protocols where either the server or client sends a channel that offers a communication of session type A. The interpretation for $A \multimap B$ can be obtained by reversing the role of client and server in $A \otimes B$ and vice versa, so we focus our attention to $A \otimes B$. There are two common (logically equivalent) presentations of the right rule for this connective:

$$\frac{\Delta \vdash P :: c : B}{\Delta, a : A \vdash \mathsf{send}\ c\ a; P :: c : A \otimes B}\ \otimes R_1 \qquad \frac{\Delta_1 \vdash P :: a : A \quad \Delta_2 \vdash Q :: c : B}{\Delta_1, \Delta_2 \vdash a : A \leftarrow P \leftarrow \Delta_1 \mid \mathsf{send}\ c\ a; Q :: c : A \otimes B}\ \otimes R_2$$

The first rule assumes a channel $a : A$ in the context whereas the second rule spawns a process P that outputs a session of type A alongside some fresh channel which it proceeds to send. Although these two rules are logically equivalent, the former formulation is problematic because we only want to run closed processes. Thus, we take the second variation.

The left rule corresponds to receiving a channel

$$\frac{\Delta, b : B, a : A \vdash P :: c : C}{\Delta, b : A \otimes B \vdash a = \mathsf{recv}\ b; P :: c : C}\ \otimes L$$

Both of these rules can be trivially extended with the two additional contexts Σ and Γ. The context split Δ_1, Δ_2 in $\otimes R_2$ must be made explicit as in our cut rule. The rules for $A \multimap B$ can be obtained by reversing the role of client and server, i.e., receiving a channel for the right rule and sending a channel for the left rule.

We can interpret a closed process offering a session of type $A \otimes B$ as a pair of processes offering a session of type A and B just like in the external choice $A \mathbin{\&} B$. This is unsurprising since logically speaking, $A \otimes B$ and $A \mathbin{\&} B$ are only distinguishable with linearity — they both collapse to a standard product in the functional layer that has weakening and contraction.

$$\frac{\Sigma, u{:}(\Gamma; \cdot \vdash c : A), v{:}(\Gamma; \cdot \vdash c : B) \mid \Gamma \Vdash N : \tau \quad \Sigma \mid \Gamma \Vdash M : \{\Gamma'; \cdot \vdash c : A \otimes B\} \quad \Sigma \mid \Gamma \Vdash \sigma : \Gamma'}{\Sigma \mid \Gamma \Vdash \mathsf{let}\ (u, v) = \mathsf{run}_\otimes(M[\sigma])\ \mathsf{in}\ N : \tau}\ \mathsf{run}_\otimes$$

Analogously, we think of running closed processes offering sessions of type $A \multimap B$ as obtaining an open process that depends on a channel of type A to offer a session of type B.

$$\frac{\Sigma, u : (\Gamma; a : A \vdash c : B) \mid \Gamma \Vdash N : \tau \quad \Sigma \mid \Gamma \Vdash M : \{\Gamma'; \cdot \vdash c : A \multimap B\} \quad \Sigma \mid \Gamma \Vdash \sigma : \Gamma'}{\Sigma \mid \Gamma \Vdash \mathsf{let}\ u = \mathsf{run}_{\multimap}(M[\sigma])\ \mathsf{in}\ N : \tau}\ \mathsf{run}_{\multimap}$$

7.3 Summary

In short, these examples reveal guiding principles to extend the process layer. First, we must take the typing rules for the appropriate construct and/or session type and extend it with Σ and Γ, most likely in the trivial manner. The extension to our formulation of the linear context Δ follows the strategy used in Section 5. Next, we must define a corresponding run primitive for the construct, which involves interpreting the result of a single round of communication. This amounts to capturing the relevant data in the process syntax as either values (for instance, in value output) or as metaassumptions such as continuation processes in a let-style syntax.

Finally, we must account for context splits during metasubstitutions $[\theta]_{\vec{\rho}}$; when branching on a choice, case a (P, Q), both P and Q use the same linear context, so we apply the instantiations $\vec{\rho}$ to both subprocesses without splitting. However in the case of $a : A \leftarrow P \leftarrow \Delta_1 \mid \mathsf{send}\ c\ a; Q$, the context is split between P and Q, so we must appropriately split $\vec{\rho}$ as in the metasubstitution case for our spawn construct (see Section 5.1).

We believe that extending our metatheoretic results with these additional connectives is straightforward given that none of these connectives interact with the metalevel layer nor the functional layer in any unique way; the additional cases they introduce would correspond exactly to standard proofs in a process calculus.

8 Related Work

Our paper is a confluence of several ideas and contributions. In particular, it builds on ideas from three different areas:

(1) Combining functional programming with session types
(2) Metaprogramming over linear type systems
(3) Typed heterogeneous metaprogramming

8.1 Combining Functional Programming with Session Types

A key aspect of FuSes is the fact that it tames a session-typed process calculus language within a functional programming setting. Because session types usually assumes a process calculus-like framework and the process semantics is inherently incompatible with (pure) functional programming semantics, is has been challenging to integrate session typed processes into a practical functional programming language.

SILL [Toninho et al. 2013], which our system is heavily inspired by, gives a logical foundation towards this combination by separating the sequent calculus-like process layer and the natural deduction-like functional layer. We follow their approach of embedding process calculus code through a contextual box monad in the functional layer.

The core fragment of FuSes as given in Section 3 captures the kind of function-process interactions that are available in SILL. The remaining developments of process code execution and observation (Section 4) and type-safe code analysis (Section 5) are all new features that SILL does not support.

Furthermore, a very notable design choice of SILL is that it is *process first* — it tames a functional programming language *within* a process calculus language, meaning it is not well-suited to be implemented as a functional programming language. On the contrary, FuSes is *functional first* —

functions can generate and run concurrent process code, making it a better suited foundation for a programming language.

Other variants of SILL [Balzer and Pfenning 2017; Das and Pfenning 2020] are orthogonal to our work since they tend to omit the contextual box and solely focus on the session-typed process layer.

GV [Gay and Vasconcelos 2010] and its extensions [Thiemann and Vasconcelos 2016], unlike FuSes, do not separate concurrency into any sort of process layer and instead implement session types directly within a functional programming language. Their approach can roughly be summarized as extending a core functional programming language with a linear type system and adding appropriate primitives that enable message-passing communication over channels and concurrent execution of functions.

In contrast, FuSes treats functional programming and concurrent programming separately. One notable result of this design choice is in supporting metaprogramming, which GV does not support. Our separation enables us to target our metaprogramming to the process layer. Metaprogramming support in GV on the other hand would require supporting both the process/concurrency aspects and the functional aspects of GV, making the development more complex.

The other advantage of our separation is that FuSes has a modular design, making metatheory relatively self-contained within each (functional/process) layer. For instance, our proof of type preservation can be separated into purely functional, purely process, and a mix of functional/process cases. The purely functional and process cases make no reference of the other layer and are essentially identical to proof cases of type preservation in a λ-calculus and a session-typed process calculus, respectively. Any complexities that arise from extending one layer would be isolated to the few constructs where the functional and process layers interact, making FuSes easier to scale, demonstrated by our extension of the process layer in Section 7.

And finally, various libraries [Imai et al. 2019; Kokke and Dardha 2021; Padovani 2017] implement variants of session types. These libraries necessarily rely on concurrent primitives in the implementation language and typically even unsafe features, making it challenging to establish any sort of safety guarantees. In contrast, FuSes is meant to be a foundational work; our separation between functional and concurrency allows for metatheory to be performed mostly in isolation.

8.2 Metaprogramming over Linear Type Systems

One key technical challenge that we address in our system is our need to support code generation, analysis, and execution for a linear type system.

To our knowledge there is very little work that aims to provide a foundation for metaprogramming with linear type systems through (linear) contextual types while preserving the distinction between (static) code and (meta)programs that compute. LINCX [Georges et al. 2017] is one such language. It extends the linear logical framework with contextual types and shares very similar technical challenges with our setting. However, they do not distinguish between an unrestricted context and a linear context for contextual objects like in our setting and instead use one context with tags. The key difference with our contexts is that they only allow one context variable in the context formation; they nevertheless partially recover context splits as is typical in linear systems through an algebra on subscripts. Their restriction greatly simplifies metasubstitutions since they do not have to deal with the kinds of name conflicts that FuSes faces, their system forces a bottom-up view of derivations and therefore requires upfront knowledge of all assumptions. They also cannot support merges of two identical context variables, which we believe is a fairly important and common paradigm that appears in metaprogramming for session types, such as running two copies of the same process in parallel.

Similarly, [Ângelo et al. 2024] develops a staged metaprogramming system that support linear contextual objects based on Moebius [Jang et al. 2022]. Their technical development is however limited to the linear λ-calculus, although they do give a brief outline on how to extend their system to support session types. This effectively amounts to repeating their technical developments over the system GV. Furthermore their system does not support code analysis or pattern matching, which, when combined with a linear type system, contributed most to the complexity of our system.

Finally, a major strength of FuSes is that it has an implementation. To our knowledge, no systems that support type-safe metaprogramming with recursive code analysis have been implemented.

8.3 Heterogeneous Metaprogramming

Another dimension of our paper is that it is closely related to (typed) heterogeneous metaprogramming systems since we may view FuSes as a heterogeneous metaprogramming language that combines two different programming paradigms.

For instance, in MetaHaskell [Mainland 2012], the author describes a heterogeneous system that supports code generation over multiple target languages including Linear MiniML. The ideas involved in supporting the linear type system in Linear MiniML is mostly in line with the core of FuSes in Section 3. However, there seems to be an explicit design choice for the metaprogramming language to be agnostic to the syntax of the target language for the sake of generality, meaning features such as code analysis or execution cannot be possible in their system.

Systems such as Beluga [Pientka and Cave 2015] or Cocon [Pientka et al. 2019] can be seen as a form of heterogeneous metaprogramming systems in some sense as well. Both of these systems take (flavors of) logical framework LF as its target language and build a richer computation layer based on contextual type theory which supports generation and pattern matching of contextual LF objects. However, none of these frameworks support linearity or session types directly; one would have to encode linearity by adopting techniques such as the one introduced in [Crary 2010] or even model a linear context as some data structure and also model the operational semantics. Recently, this framework has also been used to implement session-typed processes in LF where Sano et al. use an explicit linearity predicate to enforce linearity constraints within an intuitionistic setting to write proofs about well-typed session types at Beluga's computation layer [Sano et al. 2023]. More importantly, languages such as Beluga [Pientka and Cave 2015] or Cocon [Pientka et al. 2019] clearly separate the syntax of code from programs that generate and analyze code. This means the semantics of session-typed processes cannot be fully integrated into the functional layer; the best one can do is simulating the behavior. In FuSes, we can observe the native behavior of processes from the functional layer.

9 Conclusion

In this paper, we present FuSes, a functional programming language that enables the generation, execution, and analysis of session-typed process calculus code. We support code generation and composition by enriching the functional language with a contextual box type that has both a linear channel and a functional context. Standard techniques such as code splicing can be done naturally through a `letbox` construct that binds process code to a global metacontext accessible from the process layer. Next, we allow functions to run and observe process code through a series of run primitives. Finally, we support type-safe code analysis through top-level functions that can abstract over metalevel assumptions.

We demonstrate the expressive power of FuSes through an extensive collection of examples of code transformations and optimizations, thereby proving that these manipulations are type safe. We implement FuSes and all examples. Section 7 illustrates the extensibility afforded by our modular design.

9.1 Future Work

We view this paper primarily as setting a logical foundation for both metaprogramming over session types and incorporating concurrency to functional programming. Thus, there are many interesting directions that we would like to further pursue.

Implementation. Since we describe the system in a declarative manner, there are many parts of the system that we had to reformulate algorithmically in our implementation. The current implementation uses a backtracking algorithm to establish a metasubstitution between the arguments given in a top-level function call $f[\theta]$ and patterns in a particular branch, which is quite expensive. In practice, we may want to restrict the kinds of patterns that programmers can write, which would greatly simplify the pattern matching algorithm. Similarly, we do not check that a top-level function covers all its cases nor make only well-founded recursive calls. It will be interesting to understand the exact tradeoffs between the kind of guarantees that the language can offer and the expressivity of the language under certain constraints on patterns.

Equirecursive session types. While our modular design makes extending the process layer far more tractable, as evidenced by Section 7, it is unclear how to extend our language with recursive session types and processes. Indeed, standard presentations of recursive session types are equirecursive, and their non-syntax-directed nature makes pattern matching difficult. While we can simulate a finite unfolding of recursive session types and processes through recursion in the function layer, it would be useful from a practical standpoint to integrate recursion directly to the process layer of FuSes.

Shared memory. In this paper, we focus on metaprogramming over session types due to its well-understood logical foundations. However, shared-memory concurrency has also recently been given similar logical foundations [Pruiksma and Pfenning 2022]. It would be interesting to understand whether our approach also works in the shared memory setting, given its prevalence over message-passing concurrency.

Data-Availability Statement

The software containing the implementation of FuSes alongside the implementation of case studies is available on Zenodo [Sano et al. 2025].

Acknowledgments

This work was funded in part by a Natural Sciences and Engineering Research Council of Canada (Discovery Grant grant 206263), Fonds de recherche du Quebec - Nature et technologies (grant number 253521), Tomlinson Doctoral Fellowship awarded to the first author, the Natural Sciences and Engineering Research Council of Canada (RGPIN-2024-04777), and by national funds through FCT, Fundação para a Ciência e a Tecnologia, under project UIDB/50021/2020 (DOI:10.54499/UIDB/50021/2020).

We also thank the anonymous reviewers for their valuable comments and feedback.

References

Stephanie Balzer and Frank Pfenning. 2017. Manifest Sharing with Session Types. In *International Conference on Functional Programming (ICFP)*. ACM, 37:1–37:29. doi:10.1145/3110281 Extended version available as Technical Report CMU-CS-17-106R, June 2017..

Karl Crary. 2010. Higher-order Representation of Substructural Logics. In *Proceedings of the 15th International Conference on Functional Programming (ICFP 2010)*, P.Hudak and S.Weirich (Eds.). ACM, Baltimore, Maryland, 131–142. doi:10.1145/1863543.1863565

Ankush Das and Frank Pfenning. 2020. Rast: Resource-Aware Session Types with Arithmetic Refinements. In *5th International Conference on Formal Structures for Computation and Deduction (FSCD 2020)*, Z. Ariola (Ed.). LIPIcs 167, 33:1–33:17. doi:10.4230/LIPIcs.FSCD.2020.33 System description.

Jana Dunfield and Brigitte Pientka. 2008. Case Analysis of Higher-Order Data. *Electronic Notes in Theoretical Computer Science* 228 (2008), 69–84. doi:10.1016/j.entcs.2008.12.117 Proceedings of the International Workshop on Logical Frameworks and Metalanguages: Theory and Practice (LFMTP 2008).

Simon J. Gay and Vasco T. Vasconcelos. 2010. Linear Type Theory for Asynchronous Session Types. *Journal of Functional Programming* 20, 1 (Jan. 2010), 19–50. doi:10.1017/S0956796809990268

Aina Linn Georges, Agata Murawska, Shawn Otis, and Brigitte Pientka. 2017. LINCX: A Linear Logical Framework with First-Class Contexts. In *26th European Symposium on Programming (ESOP 2017) (Lecture Notes in Computer Science (LNCS 20201))*, Hongseok Yang (Ed.). 530–555. doi:10.1007/978-3-662-54434-1_20

Hannah Gommerstadt, Limin Jia, and Frank Pfenning. 2018. Session-Typed Concurrent Contracts. In *European Symposium on Programming (ESOP'18)*, A. Ahmed (Ed.). Springer LNCS 10801, Thessaloniki, Greece, 771–798. doi:10.1016/j.jlamp.2021.100731

Kohei Honda. 1993. Types for Dyadic Interaction. In *4th International Conference on Concurrency Theory (CONCUR 1993)*, E. Best (Ed.). Springer LNCS 715, 509–523. doi:10.1007/3-540-57208-2_35

Keigo Imai, Nobuko Yoshida, and Shoji Yuen. 2019. Session-ocaml: A session-based library with polarities and lenses. *Sci. Comput. Program.* 172 (2019), 135–159. doi:10.1016/j.scico.2018.08.005

Junyoung Jang, Samuel Gélineau, Stefan Monnier, and Brigitte Pientka. 2022. Moebius: metaprogramming using contextual types: the stage where system f can pattern match on itself. *Proc. ACM Program. Lang.* 6, POPL, Article 39 (jan 2022), 27 pages. doi:10.1145/3498700

Wen Kokke and Ornela Dardha. 2021. *Deadlock-free session types in linear Haskell.* Association for Computing Machinery, New York, NY, USA, 1–13. doi:10.1145/3471874.3472979

Geoffrey Mainland. 2012. Explicitly heterogeneous metaprogramming with MetaHaskell. *SIGPLAN Not.* 47, 9 (sep 2012), 311–322. doi:10.1145/2398856.2364572

Aleksandar Nanevski, Frank Pfenning, and Brigitte Pientka. 2008. Contextual Modal Type Theory. *Transactions on Computational Logic* 9, 3 (2008). doi:10.1145/1352582.1352591

Luca Padovani. 2017. A simple library implementation of binary sessions. *Journal of Functional Programming* 27 (2017), e4. doi:10.1017/S0956796816000289

Brigitte Pientka and Andreas Abel. 2015. Structural Recursion over Contextual Objects. In *13th International Conference on Typed Lambda Calculi and Applications (TLCA'15)*, Thorsten Altenkirch (Ed.). Leibniz International Proceedings in Informatics (LIPIcs) of Schloss Dagstuhl, 273–287. doi:10.4230/LIPIcs.TLCA.2015.273

Brigitte Pientka, Andreas Abel, Francisco Ferreira, David Thibodeau, and Rébecca Zucchini. 2019. Cocon: Computation in Contextual Type Theory. *CoRR* abs/1901.03378 (2019). arXiv:1901.03378 http://arxiv.org/abs/1901.03378

Brigitte Pientka and Andrew Cave. 2015. Inductive Beluga: Programming Proofs. In *25th International Conference on Automated Deduction (CADE 2015)*, A. Felty and A. Middeldorp (Eds.). Springer LNCS 9195, Berlin, Germany, 272–281. doi:10.1007/978-3-319-21401-6_18

Klaas Pruiksma and Frank Pfenning. 2022. Back to futures. *J. Funct. Program.* 32 (2022), e6. doi:10.1017/S0956796822000016

Chuta Sano, Deepak Garg, Ryan Kavanagh, Brigitte Pientka, and Bernardo Toninho. 2025. *Fusing Session-typed Concurrent Programming into Functional Programming - Implementation.* doi:10.5281/zenodo.15643008

Chuta Sano, Ryan Kavanagh, and Brigitte Pientka. 2023. Mechanizing Session-Types using a Structural View: Enforcing Linearity without Linearity. *Proc. ACM Program. Lang.* 7, OOPSLA2 (2023), 374–399. doi:10.1145/3622810

Peter Thiemann and Vasco T. Vasconcelos. 2016. Context-Free Session Types. In *Proceedings of the 21st International Conference on Functional Programming (ICFP)*. ACM, Nara, Japan, 462–475. doi:10.1145/2951913.2951926

Bernardo Toninho, Luís Caires, and Frank Pfenning. 2013. Higher-Order Processes, Functions, and Sessions: A Monadic Integration. In *Proceedings of the European Symposium on Programming (ESOP'13)*, M. Felleisen and P. Gardner (Eds.). Springer LNCS 7792, Rome, Italy, 350–369. doi:10.1007/978-3-642-37036-6_20

Pedro Ângelo, Atsushi Igarashi, and Vasco Vasconcelos. 2024. Linear Contextual Metaprogramming and Session Types. *Electronic Proceedings in Theoretical Computer Science* 401 (04 2024), 1–10. doi:10.4204/EPTCS.401.1

Received 2025-02-27; accepted 2025-06-27