LINCX: A Linear Logical Framework with First-class Contexts

Aina Linn Georges[1], Agata Murawska[⋆2], Shawn Otis[1], and Brigitte Pientka[1]

[1] McGill University, Montreal, QC, Canada,
{aina.georges, shawn.otis}@mail.mcgill.ca
bpientka@cs.mcgill.ca
[2] IT University of Copenhagen, Denmark,
agmu@itu.dk

Abstract. Linear logic provides an elegant framework for modelling stateful, imperative and concurrent systems by viewing a context of assumptions as a set of resources. However, mechanizing the meta-theory of such systems remains a challenge, as we need to manage and reason about mixed contexts of linear and intuitionistic assumptions.
We present LINCX, a contextual linear logical framework with first-class mixed contexts. LINCX allows us to model (linear) abstract syntax trees as syntactic structures that may depend on intuitionistic and linear assumptions. It can also serve as a foundation for reasoning about such structures. LINCX extends the linear logical framework LLF with first-class (linear) contexts and an equational theory of context joins that can otherwise be very tedious and intricate to develop. This work may be also viewed as a generalization of contextual LF that supports both intuitionistic and linear variables, functions, and assumptions.
We describe a decidable type-theoretic foundation for LINCX that only characterizes canonical forms and show that our equational theory of context joins is associative and commutative. Finally, we outline how LINCX may serve as a practical foundation for mechanizing the meta-theory of stateful systems.

1 Introduction

Logical frameworks make it easier to mechanize formal systems and proofs about them by providing a single meta-language with abstractions and primitives for common and recurring concepts, like variables and assumptions in proofs. This can have a major impact on the effort and cost of mechanization. By factoring out and abstracting over low-level details, it reduces the time it takes to mechanize formal systems, avoids errors in manipulating low-level operations, and makes the mechanizations themselves easier to maintain. It can also make an enormous difference when it comes to proof checking and constructing meta-theoretic proofs, as we focus on the essential aspect of a proof without getting bogged down in the quagmire of bureaucratic details.

⋆ Supported by grant 10-092309 from the Danish Council for Strategic Research to the *Demtech* project.

The contextual logical framework [20, 21], an extension of the logical framework LF [14], is designed to support a broad range of common features that are needed for mechanizations of formal systems. To model variables, assumptions and derivations, programmers can take advantage of higher-order abstract syntax (HOAS) trees; a context of assumptions together with properties about uniqueness of assumptions can be represented abstractly using first-class contexts and context variables [21]; single and simultaneous substitutions together with their equational theory are supported via first-class substitutions [7, 8]; finally, derivation trees that depend on a context of assumption can be precisely described via contextual objects [20]. This last aspect is particularly important. By encapsulating and representing derivation trees together with their surrounding context of assumptions, we can analyze and manipulate these rich syntactic structures via pattern matching, and can construct (co)inductive proofs by writing recursive programs about them [24, 6]. This leads to a modular and robust design where we cleanly separate the representation of formal systems and derivations from the (co)inductive reasoning about them.

Substructural frameworks such as the linear logical framework LLF [9] provide additional abstractions to elegantly model the behaviour of imperative operations such as updating and deallocating memory [30, 12] and concurrent computation (see for example session types [5]). However, it has been very challenging to mechanize proofs about LLF specifications as we must manage mixed contexts of unrestricted and linear assumptions. When constructing a derivation tree, we must often split the linear resources and distribute them to the premises relying on a *context join* operation, written as $\Psi = \Psi_1 \bowtie \Psi_2$. This operation should be commutative and associative. Unrestricted assumptions present in Ψ should be preserved in both contexts Ψ_1 and Ψ_2. The mix of unrestricted and restricted assumptions leads to an intricate equational theory of contexts that often stands in the way of mechanizing linear or separation logics in proof assistants and has spurred the development of specialized tactics [16, 2].

Our main contribution is the design of LINCX (read: "lynx"), a contextual linear logical framework with first-class contexts that may contain both intuitionistic and linear assumptions. On the one hand our work extends the linear logical framework LLF with support for first-class linear contexts together with an equational theory of context joins, contextual objects and contextual types; on the other we can view LINCX as a generalization of contextual LF to model not only unrestricted but also linear assumptions. LINCX hence allows us to abstractly represent syntax trees that depend on a mixed context of linear and unrestricted assumptions, and can serve as a foundation for mechanizing the meta-theory of stateful systems where we implement (co)inductive proofs about linear contextual objects by pattern matching following the methodology outlined by Cave and Pientka [6] and Thibodeau et.al. [29]. Our main technical contributions are:

1) A bi-directional decidable type system that only characterizes canonical forms of our linear LF objects. Consequently, exotic terms that do not represent legal objects from our object language are prevented. It is an inherent property

of our design that bound variables cannot escape their scope, and no separate reasoning about scope is required. To achieve this we rely on hereditary substitution to guarantee normal forms are preserved. Equality of two contextual linear LF objects reduces then to syntactic equality (modulo α-renaming).

2) Definition of first-class (linear) contexts together with an equational theory of context joins. A context in LINCX may contain both unrestricted and linear assumptions. This not only allows for a uniform representation of contexts, but also leads to a uniform representation of simultaneous substitutions. Context variables are indexed and their indices are freely built from elements of an infinite, countable set through a context join operation ($\bowtie$) that is associative, commutative and has a neutral element. This allows a canonical representation of contexts and context joins. In particular, we can consider contexts equivalent modulo associativity and commutativity. This substantially simplifies the meta-theory of LINCX and also directly gives rise to a clean implementation of context joins which we exploit in our mechanization of the meta-theoretic properties of LINCX.

3) Mechanization of LINCX *together with its meta-theory in the proof assistant* BELUGA *[23].* Our development takes advantage of higher-order abstract syntax to model binding structures compactly. We only model linearity constraints separately. We have mechanized our bi-directional type-theoretic foundation together with our equational theory of contexts. In particular, we mechanized all the key properties of our equational theory of context joins and the substitution properties our theory satisfies.

We believe that LINCX is a significant step towards modelling (linear) derivation trees as well-scoped syntactic structures that we can analyze and manipulate via case-analysis and implementing (co)inductive proofs as (co)recursive programs. As it treats contexts, where both unrestricted and linear assumptions live, abstractly and factors out the equational theory of context joins, it eliminates the need for users to explicitly state basic mathematical definitions and lemmas and build up the basic necessary infrastructure. This makes the task easier and lowers the costs and effort required to mechanize properties about imperative and concurrent computations.

2 Motivating Examples

To illustrate how we envision using (linear) contextual objects and (linear) contexts, we implement two program transformations on object languages that exploit linearity. We first represent our object languages in LINCX and then write recursive programs that analyze the syntactic structure of these objects by pattern matching. This highlights the role that contexts and context joins play.

2.1 Example: Code Simplification

To illustrate the challenges that contexts pose in the linear setting, we implement a program that translates linear Mini-ML expressions that feature let-expression

into a linear core lambda calculus. We define the linear Mini-ML using the linear type `ml` and our linear core lambda calculus using the linear type `lin` as our target language. We introduce a linear LF type together with its constructors using the keyword **Linear LF**.

```
Linear LF ml : type =
   | lam  : (ml -o ml) -o ml
   | app  : ml -o ml -o ml
   | letv : ml -o (ml -o ml) -o ml;
```

```
Linear LF lin: type =
| llam   : (lin -o lin) -o lin
| lapp  : lin -o lin -o lin
;
```

We use the linear implication `-o` to describe the linear function space and we model variable bindings that arise in abstractions and let-expressions using higher-order abstract syntax, as is common in logical frameworks. This encoding technique exploits the function space provided by LF to model variables. In linear LF it also ensures that bound variables are used only once.

Our goal is to implement a simple translation of Mini-ML expressions to the core linear lambda calculus by eliminating all let-expressions and transforming them into function applications. We thus need to traverse Mini-ML expressions recursively. As we go under an abstraction or a let-expression, our sub-expression will not, however, remain closed. We therefore model a Mini-ML expression together with its surrounding context in which it is meaningful. Our function `trans` takes a Mini-ML expression in a context γ, written as `[`γ $\vdash$ `ml]`, and returns a corresponding expression in the linear lambda calculus in a context δ, an object of type `[`δ $\vdash$ `lin]`. More precisely, there exists such a corresponding context δ. Due to linearity, the context of the result of translating a Mini-ML term has the same length as the original context. This invariant is however not explicitly tracked.

We first define the structure of such contexts using context schema declarations. The tag `l` ensures that any declaration of type `ml` in a context of schema `ml_ctx` must be linear. Similarly, any declaration of type `lin` in a context of schema `core_ctx` must be linear.

```
schema ml_ctx   = l (ml);
schema core_ctx = l (lin);
```

To characterize the result of this translation, we define a recursive type:

```
inductive Result: type = Return : (δ:core_ctx) [δ ⊢ lin] → Result;
```

By writing round parenthesis in `(`δ`:core_ctx)` we indicate that we do not pass δ explicitly to the constructor `Return`, but it can always be reconstructed. It is merely an annotation declaring the schema of δ.

We now define a recursive function `trans` using the keyword **rec** (see Fig. 1). First, let us highlight some high level principles and concepts that we use. We write `[`Ψ $\vdash$ `N]` to describe an expression `N` that is meaningful in the context Ψ. For example, `[`γ $\vdash$ `lam ^ (`$\widehat{\lambda}$`x. M)]` denotes a term of type `ml` in the context γ where γ is a context variable that describes contexts abstractly. We call `M` a meta-variable. It stands for a `ml` term that may depend on the context γ`,x:ml`. In general, all meta-variables are associated with a stuck substitution, written `N[`σ`]` or `M[`σ`]`. We usually omit the substitution σ, if it is the identitiy substitution. One substitution that

```
rec trans : (γ:ml_ctx)[γ ⊢ ml] → Result =
fn e ⇒ case e of
| [x:ml ⊢ x] ⇒ Return [x:lin ⊢ x]

| [γ ⊢ lam ^ (λ̂x. M)] ⇒
   let Return [δ, x:lin ⊢ M'] = trans [γ, x:ml ⊢ M] in
    Return [δ ⊢ llam ^ (λ̂x. M)]

| [γ(1⋈2) ⊢ app ^ M ^ N] where M:[γ1 ⊢ ml] and N:[γ2 ⊢ ml] and γ(1⋈2) = γ1 ⋈ γ2⇒
  let Return [δ1 ⊢ M'] = trans [γ1 ⊢ M] in
  let Return [δ2 ⊢ N'] = trans [γ2 ⊢ N] in
     Return [δ(1⋈2) ⊢ lapp ^ M' ^ N'] where δ(1⋈2) = δ1 ⋈ δ2

| [γ(1⋈2) ⊢ let ^ M ^ (λ̂x. N)] where M:[γ1⊢ ml] and N:[γ2, x:ml ⊢ ml]
                                and γ(1⋈2) = γ1 ⋈ γ2 ⇒
  let Return [δ1 ⊢ M']       = trans [γ1 ⊢ M] in
  let Return [δ2, x:lin ⊢ N'] = trans [γ2, x:ml ⊢ N] in
    Return [δ(1⋈2) ⊢ lapp ^ (llam ^ (λ̂x. N')) ^ M'] where δ(1⋈2) = δ1 ⋈ δ2;
```

Fig. 1. Translation of linear ML-expressions to a linear core language

frequently arises in practice is the empty substitution that is written as `[]` and maps from the empty context to an unrestricted context Ψ. It hence acts as a weakening substitution.

Our simplification is implemented by pattern matching on `[γ ⊢ ml]` objects and specifying constraints on contexts. In the variable case, since we have a linear context, we require that `x` be the only variable in the context[3]. In the lambda case `[γ ⊢ lam ^(λ̂x.M)]` we write `^` for linear application and linear abstraction. We expect the type of `M` to be inferred as `[γ,x:ml ⊢ ml]`, since we interpret every pattern variable to depend on all its surrounding context unless otherwise specified. We now recursively translate `M` in the extended context `γ, x:ml`, unpack the result and rebuild the equivalent linear term. Note that we pattern match on the result translating `M` by writing `Result [δ, x:lin ⊢ M']`. However, we do not necessarily know that the output `core_ctx` context is of the same length as the input `ml_ctx` context and hence necessarily has the shape `[δ, x:lin]`, as we do not track this invariant explicitly. To write a covering program we would need to return an error, if we would encounter `Return [ ⊢ M']`, i.e. a closed term where δ is empty. We omit this case here.

The third and fourth cases are the most interesting ones, as we must split the context. When we analyze for example `[γ(1⋈2) ⊢ app ^ M ^N]`, then `M` has some type `[γ1 ⊢ ml]` and `N` has some type `[γ2 ⊢ ml]` where $\gamma_{(1\bowtie 2)} = \gamma_1 \bowtie \gamma_2$. We specify these type annotations and context constraints explicitly. Note that we overload the $\bowtie$ symbol in this example: when it occurs as a subscript it is part of the name, while when we write $\gamma_1 \bowtie \gamma_2$ it refers to the operation on contexts. Then we can simply recursively translate `M` and `N` and rebuild the final result where

[3] In case we have a mixed context, we could specify instead that the rest of the context is unrestricted, using the keywords **where** and `unr`.

we explicitly state $\delta_{1\bowtie 2} = \delta_1 \bowtie \delta_2$. We proceed similarly to translate recursively every let-expression into a function application.

Type checking verifies that a given object is well-typed modulo context joins. This is non-trivial. Consider for example $[\delta_{(1\bowtie 2)} \vdash \texttt{lapp ^ (llam ^(}\widehat{\lambda}\texttt{x. N')) ^ M'}]$ where $\delta_{(1\bowtie 2)} = \delta_1 \bowtie \delta_2$. Clearly, we should be able to type check such an example also if the user wrote $\delta = \delta_2 \bowtie \delta_1$. Hence we want our underlying type theory to reason about context constraints modulo associativity and commutativity.

As the astute reader will have noticed, we only allow one context variable in every context, i.e. writing $[\delta_1, \delta_2 \vdash \texttt{lapp ^ (llam ^ (}\widehat{\lambda}\texttt{x. N')) ^ M'}]$ is illegal. Furthermore, we have deliberately chosen the subscripts for our context variables to emphasize their encoding in our underlying theory. Note that all context variables that belong to the same tree of context splits have the same name, but differ in their subscripts. The context variables γ_1 and γ_2 are called *leaf-level context variables*. The context variable $\gamma_{(1\bowtie 2)}$ is their direct parent and sits at the root of this tree. One can think of the tree of context joins as an abstraction of the typing derivation. To emphasize this idea, let us consider the following deeply nested pattern: $[\gamma_{((11\bowtie 12)\bowtie 2)} \vdash \texttt{lapp ^ (lapp ^ (llam ^ (}\widehat{\lambda}\texttt{x. M)) ^ N') ^ K}]$ where $\texttt{M} : [\gamma_{11}, \texttt{x}\widehat{:}\ \texttt{ml} \vdash \texttt{ml}]$, $\texttt{N} : [\gamma_{12} \vdash \texttt{ml}]$, and $\texttt{K} : [\gamma_2 \vdash \texttt{ml}]$, and where we again encode the splitting of γ in its subscript. Our underlying equational theory of context joins treats $\gamma_{(11\bowtie(12\bowtie 2))}$ as equivalent to $\gamma_{((11\bowtie 12)\bowtie 2)}$ or $\gamma_{((12\bowtie 11)\bowtie 2)}$ as it takes into account commutativity and associativity. However, it may require us to generate a new intermediate node $\gamma_{(1\bowtie 21)}$ and eliminate intermediate nodes (such as $\gamma_{21\bowtie 22}$).

Fig. 2. Context Joins

Our encoding of context variables is hence crucial to allow the rearrangement of context constraints, but also to define what it means to instantiate a given context variable such as γ_{21} with a concrete context Ψ. If Ψ contains also unrestricted assumptions then instantiating γ_{21} will have a global effect, as unrestricted assumptions are shared among all nodes in this tree of context joins. This latter complication could possibly be avoided if we separate the context of intuitionistic assumptions and the context of linear assumptions. However, this kind of separation between intuitionistic and linear assumptions is not trivial in the dependently typed setting because linear assumptions may depend on intuitionistic assumptions.

This design of context variables and capturing their dependency is essential to LINCX and to the smooth extension of contextual types to the linear setting. As the leaf-level context variables uniquely describe a context characterized by a

tree of context joins, we only track the leaf-level context variables as assumptions while type checking an object, but justify the validity of context variables that occur as interior nodes through the leaf-level variables. We want to emphasize that this kind of encoding of context variables does not need to be exposed to programmers.

2.2 Example: CPS translation

As a second example, we implement the translation of programs into continuation passing style following Danvy and Filinski [11]. Concretely, we follow closely the existing implementation of type-preserving CPS translation in Beluga by Belanger et.al [1], but enforce that the continuations are used linearly, an idea from Berdine et.al [3]. Although context splits do not arise in this example, as we only have one linear variable (standing for the continuation) in our context, we include it, to showcase the mix and interplay of intuitionistic and linear function spaces in encoding program transformations.

Our source language is a simple language consisting of natural numbers, functions, applications and let-expressions. We only model well-typed expressions by defining a type `source` that is indexed by types `tp`.

```
Linear LF tp : type =
| nat   : tp
| arr   : tp → tp → tp
;
```

```
Linear LF source : tp → type =
| app   : source (arr S T) → source S → source T
| lam   : (source S → source T) → source (arr S T)
| z     : source nat
| s     : source nat → source nat;
```

In our target language we distinguish between expressions, characterized by the type `exp` and values, defined by the type `value`. Continuations take values as their argument and return an `exp`. We ensure that each continuation itself is used exactly once by abstracting `exp` over the linear function space.

```
Linear LF exp : type =
| kapp     : value (arr S T) → value S → (value T → exp) -o exp
| halt     : value S → exp
and value : tp → type =
| klam     : (value S → (value T → exp) -o exp) → value (arr S T)
| kz       : value nat
| ksuc     : value nat → value nat ;
```

We can now define our `source` and `value` contexts as unrestricted contexts by marking the schema element with the tag `u`.

```
schema sctx = u (source T);
schema vctx = u (value T);
```

To guarantee that the resulting expression is well-typed, we define a context relation `Ctx_Rel` to relate the `source` context to the `value` context (see Fig. 3). Notice that we explicitly state that the type `s` of a source and target expression is closed; it does not depend on γ or δ. To distinguish between objects that depend on their surrounding context and objects that do not, we associate every index and pattern variable with a substitution (the identity substitution by default); if we want to state that a given variable is closed, we associate it with the empty substitution `[]`.

We can now define the translation itself (see Fig. 3). The function `cpse` takes in a context relation `Ctx_Rel [γ] [δ]` and a source term of type `source S[]` that depends on context γ. It then returns the corresponding expression of type `exp`, depending on context δ extended by a continuation from `value S` to `exp`. The fact that the continuation is used only once in `exp` is enforced by declaring it linear in the context. The translation proceeds by pattern matching on the source term. We concentrate here on the interesting cases.

```
data Ctx_Rel: {γ:sctx}{δ:vctx} type =
Nil  : Ctx_Rel [] []
Cons : Ctx_Rel [γ] [δ] → Ctx_Rel [γ, x:source S[]] [δ, v:value S[]] ;


rec cpse:(γ:sctx)(δ:vctx)(S:[ ⊢ tp])
        Ctx_Rel [γ] [δ] → [γ ⊢ source S[]] → [δ, k̂:value S[] → exp ⊢ exp] =
fn r, e ⇒ case e of
| [γ ⊢ #p] ⇒
  let [δ⊢ #q] = lookup r [γ ⊢ #p] in
    [δ, k̂:value _ → exp ⊢ k #q]

| [γ ⊢ z] ⇒ let (r : Ctx_Rel [γ] [δ]) = r in [δ,k̂:value nat → exp ⊢ k kz]

| [γ ⊢ suc N] ⇒
  let [δ,k̂:value nat → exp ⊢ P] = cpse r [γ ⊢ N] in
  [δ,k̂:value nat → exp ⊢  P[λp. k (ksuc p)] ]

| [γ ⊢ lam λx. M] ⇒
   let [δ, v:value S[], k̂:value T[] → exp ⊢ P] = cpse [Cons r] [γ, x:source _ ⊢ M] in
     [δ, k^:value (arr S[] T[]) → exp ⊢ k (klam (λx.λ̂c. P))]

| [γ ⊢ app M N] ⇒
  let [δ, k1̂:value (arr S[] T[]) → exp ⊢ P] = cpse r [γ ⊢ M] in
  let [δ, k2̂:value S[]           → exp ⊢ Q] = cpse r [γ ⊢ N] in
    [δ,k̂:value T[] → exp ⊢ P[λf. Q[λx. kapp f x ^ k]]];
```

Fig. 3. CPS Translation

Parameter Variable If we encounter a variable from the context γ, written as `#p`, we look up the corresponding variable `#q` in the target context δ by using the context relation and we pass it to the continuation `k`. We omit here the definition of the lookup function which is straightforward. We use `_` where we believe that the omitted object can reasonably be inferred. Finally, we note that `k #q` is well-typed in the context δ, `k̂:value _ → exp`, as `k` is well-typed in the context that only contains the declaration `k̂:value _ → exp` and `#q` is well-typed in the context δ.

Constant `z` We first retrieve the target context δ to build the final expression by pattern matching on the context relation `r`. Then we pass `kz` to the continuation `k` in the context δ,`k̂:value nat → exp`. Note that an application `k kz` is well-typed in δ,`k̂:value nat → exp`, as `kz` is well-typed in δ, i.e. its unrestricted part.

Lambda Abstraction To convert functions, we extend the context γ and the context relation `r` and convert the term `M` recursively in the extended context

to obtain the target expression `P`. We then pass to the continuation `k` the value `klam` $\lambda x.\widehat{\lambda} c.P$.

Application Finally, let us consider the the source term `app M N`. We translate both `M` and `N` recursively to produce the target terms `P` and `Q` respectively. We then substitute for the continuation variable `k2` in `Q` a continuation consuming the local argument of an application. A continuation is then built from this, expecting the function to which the local argument is applied and substituted for `k1` in `P` producing a well-typed expression, if a continuation for the resulting type `S` is provided.

We take advantage of our built-in substitution here to reduce any administrative redexes. The term `(`λ`x. kapp f x ^ k)` that we substitute for references to `k2` in `Q` will be β-reduced wherever that `k2` appears in a function call position, such as the function calls introduced in the variable case. We hence reduce administrative redexes using the built-in (linear) LF application.

3 LINCX: A Linear Logical Framework with First-Class Contexts

Throughout this section we gradually introduce LINCX, a contextual linear logical framework with first-class contexts (i.e. context variables) that generalizes the linear logical framework LLF [9] and contextual LF [6]. Fig. 4 presents both contextual linear LF (see Sect. 3.1) and its meta-language (see Sect. 3.6).

3.1 Syntax of Contextual Linear LF

LINCX allows for linear types, written $A \multimap B$, and dependent types $\Pi x{:}A.B$ where x may be unrestricted in B. We follow recent presentations where we only describe canonical LF objects using hereditary substitution.

As usual, our framework supports constants, (linear) functions, and (linear) applications. We only consider objects in η-long β-normal form, as these are the only meaningful terms in a logical framework. While the grammar characterizes objects in β-normal form, the bi-directional typing rules will also ensure that objects are η-long. Normal canonical terms are either intuitionistic lambda abstractions, linear lambda abstractions, or neutral atomic terms. We define (linear) applications as neutral atomic terms using a spine representation [10], as it makes the termination of hereditary substitution easier to establish. For example, instead of $x\ M_1 \ldots M_n$, we write $x \cdot M_1;\ \ldots;\ M_n;\ \epsilon$. The three possible variants of a spine head are: a variable x, a constant c or a parameter variable closure $p[\sigma]$.

Our framework contains *ordinary bound variables* x which may refer to a variable declaration in a context Ψ or may be bound by either the unrestricted or linear lambda-abstraction, or by the dependent type $\Pi x{:}A.B$. Similarly to contextual LF, LINCX also allows two kinds of *contextual variables* as terms. First, the meta-variable u of type $(\Psi \vdash P)$ stands for a general LF object of

Contextual Linear LF

Kinds	K	$::=$	$\mathsf{type} \mid \Pi x{:}A.K$
Types	A, B	$::=$	$P \mid \Pi x{:}A.B \mid A \multimap B$
Atomic Types	P, Q	$::=$	$a \cdot S$
Heads	H	$::=$	$x \mid c \mid p[\sigma]$
Spines	S	$::=$	$\epsilon \mid M\,;S \mid M\,\hat{;}\,S$
Atomic Terms	R	$::=$	$H \cdot S \mid u[\sigma]$
Canonical Terms	M, N	$::=$	$R \mid \lambda x.M \mid \widehat{\lambda} x.M$
Variable Declarations	D	$::=$	$x{:}A \mid x\hat{:}A \mid x\check{:}A$
Contexts	Ψ, Φ	$::=$	$\cdot \mid \psi_m \mid \Psi, D$
Substitutions	σ, τ	$::=$	$\cdot \mid \mathsf{id}_\psi \mid \sigma, M$

Meta-Language

Meta-Variables	X	$::=$	$u \mid p \mid \psi_i$
Meta-Objects	C	$::=$	$\widetilde{\Psi}.R \mid \widetilde{\Psi}.H \mid \Psi$
Context Schema Elem.	E	$::=$	$\lambda(\overrightarrow{x_i{:}A_i}).A \mid \lambda(\overrightarrow{x_i{:}A_i}).\widehat{A}$
Context Schemata	G	$::=$	$E \mid G + E$
Context Var. Indices	m	$::=$	$\epsilon \mid i \mid m \bowtie n$
Meta Types	U	$::=$	$\Psi \vdash P \mid \Psi \vdash \#A \mid G$
Meta-Contexts	Δ	$::=$	$\cdot \mid \Delta, X : U$
Meta-Substitutions	Θ	$::=$	$\cdot \mid \Theta, C/X$

Fig. 4. Contextual Linear LF with first-class contexts

atomic type P and uses the variables declared in Ψ. Second, the parameter variable p of type $(\Psi \vdash \#A)$ stands for a variable object of type A from the context Ψ. These contextual variables are associated with a postponed substitution σ representing a closure. The intention is to apply σ as soon as we know what u (or p resp.) stands for.

The system has one mixed context Ψ containing both intuitionistic and linear assumptions: $x{:}A$ is an intuitionistic assumption in the context (also called *unrestricted assumption*), $x\hat{:}A$ represents a linear assumption and $x\check{:}A$ stands for its dual, an unavailable assumption. It is worth noting that we use $\widehat{\ }$ throughout the system description to indicate a linear object – be it term, variable, name etc. Similarly, $\check{}$ always denotes an unavailable resource.

In the simultaneous substitution σ, we do not make the domain explicit. Rather, we think of a substitution together with its domain Ψ; the i-th element in σ corresponds to the i-th declaration in Ψ. The expression id_ψ denotes the identity substitution with domain ψ_m for some index m; we write $\cdot$ for the empty substitution. We build substitutions using normal terms M. We must however be careful: note that a variable x is only a normal term if it is of base type. As

we push a substitution σ through a λ-abstraction $\lambda x.M$, we need to extend σ with x. The resulting substitution σ, x might not be well-formed, since x might not be of base type and, in fact, we do not know its type. This is taken care of in our definition of substitution, based on contextual LF [7]. As we substitute and replace a context variable with a concrete context, we unfold and generate an (η-expanded) identity substitution for a given context Ψ.

3.2 Contexts and Context Joins

Since linearity introduces context splitting, context maintenance is crucial in any linear system. When we allow for first-class contexts, as we do in LINCX, it becomes much harder: we now need to ensure that, upon instantiation of the context variables, we do not accidentally join two contexts sharing a linear variable. To enforce this in LINCX, we allow for at most one (indexed) context variable per context and use indices to abstractly describe splitting. This lets us generalize the standard equational theory for contexts based on context joins to include context variables.

As mentioned above, contexts in LINCX are mixed. Besides linear and intuitionistic assumptions, we allow for unavailable assumptions following the approach of Schack-Nielsen and Schürmann [27], in order to maintain symmetry when splitting a context: if $\Psi = \Psi_1 \bowtie \Psi_2$, then Ψ_1 and Ψ_2 both contain all the variables declared in Ψ; however, if Ψ_1 contains a linear assumption $x\hat{:}A$, Ψ_2 will contain its unavailable counterpart $x\check{:}A$ (and vice-versa).

To account for context splitting in the presence of context variables, we index the latter. The indices are freely built from elements of an infinite, countable set $\mathcal{I}$, through a join operation ($\bowtie$). It is associative and commutative, with ϵ as its neutral element. In other words, $(\mathcal{I}^*, \bowtie, \epsilon)$ is a (partial) free commutative monoid over $\mathcal{I}$. For our presentation it is important that no element of the monoid is invertible, that is if $m \bowtie n = \epsilon$ then $m = n = \epsilon$. In the process of joining contexts, it is crucial to ensure that each linear variable is used only once: we do not allow a join of $\Psi, x\hat{:}A$ with $\Phi, x\hat{:}A$. To express the fact that indices m and n share no elements of $\mathcal{I}$ and hence the join of ψ_m with ψ_n is meaningful, we use the notation $m \perp n$. In fact we will overload $\bowtie$, changing it into a partial operation $m \bowtie n$ that fails when $m \not\perp n$. This is because we want the result of joining two context variables to continue being a correct context upon instantiation. We will come back to this point in Sect. 3.6, when discussing meta-substitution for context variables.

To give more intuition, the implementation of the indices in our formalization of the system is using binary numbers, where $\mathcal{I}$ contains powers of 2, $\bowtie$ is defined as a binary *OR* and $\epsilon = 0$. $m \perp n$ holds when m and n use different powers of 2 in their binary representation. We can also simply think of indices m as sets of elements from $\mathcal{I}$ with $\bowtie$ being $\cup$ for sets not sharing any elements.

The only context variables tracked in the meta-context Δ are the *leaf-level* context variables ψ_i. We require that these use elements of the carrier set $i \in \mathcal{I}$ as indices. To construct context variables for use in contexts, we combine leaf-level context variables using $\bowtie$ on indices. Consider again the tree describing

the context joins (see Fig. 2). In this example, we have the leaf-level context variables γ_1, γ_{21}, and γ_{22}. These are the only context variables we track in the meta-context Δ. Using a binary encoding we would use the subscripts 100, 010 and 001 instead of 1, 21, and 22.

Rules of constructing a well-formed context (Fig. 5) describe four possible initial cases of context construction. First, the empty context, written simply as $\cdot$, is well-formed. Next, there are two possibilities why a context denoted by a context variable ψ_i is well-formed. If the context variable ψ_i is declared in the meta-context Δ, then it is well-formed and describes a leaf-variable. To guarantee that also context variables that describe intermediate nodes in our context tree are well-formed, we have a composition rule that allows joining two well-formed context variables using $\bowtie$ operation on indices; the restriction we make on $\bowtie$ ensures that they do not share any leaf-level variables. ψ_ϵ forms a well-formed context as long as there is some context variable ψ_i declared in Δ. This is an abstraction that allows us to describe the intuitionistic variables of a context. Finally, the last case for context extensions is straightforward.

$\boxed{\Delta \vdash \Psi\ \mathsf{ctx}}$ $\quad \Psi$ is a valid context under meta-context Δ

$$\frac{}{\Delta \vdash \cdot\ \mathsf{ctx}} \qquad \frac{\psi_i \in \mathsf{dom}(\Delta)}{\Delta \vdash \psi_\epsilon\ \mathsf{ctx}} \qquad \frac{\psi_i \in \mathsf{dom}(\Delta)}{\Delta \vdash \psi_i\ \mathsf{ctx}} \qquad \frac{\Delta \vdash \psi_k\ \mathsf{ctx} \quad \Delta \vdash \psi_l\ \mathsf{ctx} \quad m = k \bowtie l}{\Delta \vdash \psi_m\ \mathsf{ctx}}$$

$$\frac{\Delta \vdash \Psi\ \mathsf{ctx} \quad \Delta; \overline{\Psi} \vdash A\ \mathsf{type} \quad D \in \{x{:}A, x{:}\hat{A}, x{:}\check{A}\}}{\Delta \vdash \Psi, D\ \mathsf{ctx}}$$

Fig. 5. Well-formed contexts

In general we write Γ for contexts that do not start with a context variable and Ψ, Γ for the extension of context Ψ by the variable declarations of Γ.

When defining our inference rules, we will often need to access the *intuitionistic part* of a context. Much like in linear LF [9], we introduce the function $\overline{\Psi}$ which is defined as follows:

$\boxed{\overline{\Psi}}$ Intuitionistic part of Ψ

$$\begin{aligned}
\overline{\cdot} &= \cdot \\
\overline{\psi_m} &= \psi_\epsilon \\
\overline{\Psi, x{:}A} &= \overline{\Psi}, x{:}A \\
\overline{\Psi, x{:}\hat{A}} &= \overline{\Psi}, x{:}\check{A} \\
\overline{\Psi, x{:}\check{A}} &= \overline{\Psi}, x{:}\check{A}
\end{aligned}$$

Note that this function does not remove any variable declarations from Ψ, it simply makes them unavailable. Further, when applying this function to a context variable, it drops all the indices, indicating access to only the shared part

of the context variable. After we instantiate ψ_m with a concrete context, we will apply the operation. Extracting the intuitionistic part of a context is hence simply postponed.

Further, we define notation $\mathsf{unr}(\Psi)$ to denote an unrestricted context, i.e. a context that only contains unrestricted assumptions; while $\overline{\Psi}$ drops all linear assumptions, $\mathsf{unr}(\Psi)$ simply verifies that Ψ is a purely intuitionistic context. In other words, $\mathsf{unr}(\Psi)$ holds if and only if $\overline{\Psi} = \Psi$. We omit here its (straightforward) judgmental definition.

$$\boxed{\Psi = \Psi_1 \bowtie \Psi_2} \qquad \text{Context } \Psi \text{ is a join of } \Psi_1 \text{ and } \Psi_2$$

$$\frac{}{\cdot = \cdot \bowtie \cdot} \qquad \frac{m = k \bowtie l}{\psi_m = \psi_k \bowtie \psi_l}$$

$$\frac{\Psi = \Psi_1 \bowtie \Psi_2}{\Psi, x{:}A = \Psi_1, x{:}A \bowtie \Psi_2, x{:}A} \qquad \frac{\Psi = \Psi_1 \bowtie \Psi_2}{\Psi, x{:}\check{A} = \Psi_1, x{:}\check{A} \bowtie \Psi_2, x{:}\check{A}}$$

$$\frac{\Psi = \Psi_1 \bowtie \Psi_2}{\Psi, x{:}\hat{A} = \Psi_1, x{:}\hat{A} \bowtie \Psi_2, x{:}\check{A}} \qquad \frac{\Psi = \Psi_1 \bowtie \Psi_2}{\Psi, x{:}\hat{A} = \Psi_1, x{:}\check{A} \bowtie \Psi_2, x{:}\hat{A}}$$

Fig. 6. Joining contexts

The rules for joining contexts (see Fig. 6) follow the approach presented by Schack-Nielsen in his PhD dissertation [26], but are generalized to take into account context variables. Because of the monoid structure of context variable indices, the description can be quite concise while still preserving the desired properties of this operation. For instance the expected property $\Psi = \Psi \bowtie \overline{\Psi}$ follows, on the context variable level, from ϵ being the neutral element of $\bowtie$. Indeed, for any ψ_m, we have that $\psi_m = \psi_m \bowtie \psi_\epsilon$.

It is also important to note that, thanks to the determinism of $\bowtie$, context joins are unique. In other words, if $\Psi = \Psi_1 \bowtie \Psi_2$ and $\Phi = \Psi_1 \bowtie \Psi_2$, $\Psi = \Phi$. On the other hand, context splitting is non-deterministic: given a context Ψ we have numerous options of splitting it into Ψ_1 and Ψ_2, since each linear variable can go to either of the components.

We finish this section by describing the equational theory of context joins. We expect joining contexts to be a commutative and associative operation, and the unrestricted parts of contexts in the join should be equal. Further, it is always possible to extend a valid join with a ground unrestricted context, and $\overline{\Psi}$ can always be joined with Ψ without changing the result.

Lemma 1 (Theory of context joins).

1. *(Commutativity) If* $\Psi = \Psi_1 \bowtie \Psi_2$ *then* $\Psi = \Psi_2 \bowtie \Psi_1$.
2. *(Associativity* $_1$*) If* $\Psi = \Psi_1 \bowtie \Psi_2$ *and* $\Psi_1 = \Psi_{11} \bowtie \Psi_{12}$ *then there exists a context* Ψ_0 *s.t.* $\Psi = \Psi_{11} \bowtie \Psi_0$ *and* $\Psi_0 = \Psi_{12} \bowtie \Psi_2$.
3. *(Associativity* $_2$*) If* $\Psi = \Psi_1 \bowtie \Psi_2$ *and* $\Psi_2 = \Psi_{21} \bowtie \Psi_{22}$ *then there exists a context* Ψ_0 *s.t.* $\Psi_0 = \Psi_1 \bowtie \Psi_{21}$ *and* $\Psi = \Psi_0 \bowtie \Psi_{22}$.

4. *If* $\Psi = \Psi_1 \bowtie \Psi_2$ *then* $\overline{\Psi} = \overline{\Psi_1} = \overline{\Psi_2}$.
5. *If* $\mathsf{unr}(\Gamma)$ *and* $\Psi = \Psi_1 \bowtie \Psi_2$ *then* $\Psi, \Gamma = \Psi_1, \Gamma \bowtie \Psi_2, \Gamma$.
6. *For any* Ψ, $\Psi = \Psi \bowtie \overline{\Psi}$.

We will need these properties to prove lemmas about typing and substitution, specifically for the cases that call for specific context joins.

3.3 Typing for Terms and Substitutions

We now describe the bi-directional typing rules of LINCX terms (see Fig. 7). All typing judgments have access to the meta-context Δ, context Ψ, and to a fixed well-typed signature Σ where we store constants c together with their types and kinds. LINCX objects may depend on variables declared in the context Ψ and a fixed meta-context Δ which contains contextual variables such as meta-variables u, parameter variables p, and context variables. Although the rules are bi-directional, they do not give a direct algorithm, as we need to split a context Ψ into contexts Ψ_1 and Ψ_2 such that $\Psi = \Psi_1 \bowtie \Psi_2$ (see for example the rule for checking $H \cdot S$ against a base type P). This operation is in itself non-deterministic, however since our system is linear there is only one split that makes the components (for example H and S in $H \cdot S$) typecheck.

Typing rules presented in Fig. 7 are, perhaps unsurprisingly, a fusion between contextual LF and linear LF. As in contextual LF, the typing for meta-variable closures and parameter variable closures is straightforward. A meta-variable $u : (\Psi \vdash P)$ represents an open LF object (a "hole" in a term). As mentioned earlier it has, associated with it, a postponed substitution σ, applied as soon as u is made concrete. Similarly, a parameter variable $p : (\Psi \vdash \#A)$ represents an LF variable – either an unrestricted or linear one.

As in linear LF, we have two lambda abstraction rules (one introducing intuitionistic, the other linear assumptions) and two corresponding variable cases. Moreover, we ensure that types only depend on the unrestricted part of a context when checking that two types are equal. As we rely on hereditary substitutions, this equality check ends up being syntactic equality. Similarly, when we consider a spine $M\,;S$ and check it against the dependent type $\Pi x{:}A.B$, we make sure that M has type A in the unrestricted context before continuing to check the spine S against $[M/x]_A B$. When we encounter a spine $M\,\hat{;}\,S$ and check it against the linear type $A \multimap B$ in the context Ψ, we must show that there exists a split s.t. $\Psi = \Psi_1 \bowtie \Psi_2$ and then check that the term M has type A in the context Ψ_1 and the remaining spine S is checked against B to synthesize a type P.

Finally, we consider the typing rules for substitutions, presented in Fig. 8. We exercise care in making sure the range context in the base cases, i.e. where the substitution is empty or the identity, is unrestricted. This guarantees weakening and contraction for unrestricted contexts.

The substitution σ, M is well-typed with domain $\Phi, x{:}A$ and range Ψ, if σ is a substitution from Φ to the context Ψ and in addition M has type $[\sigma]_\Phi A$ in the unrestricted context $\overline{\Psi}$. The substitution σ, M is well-typed with domain $\Phi, x\hat{:}A$ and range Ψ, if there exists a context split $\Psi = \Psi_1 \bowtie \Psi_2$ s.t. σ is a substitution

$\boxed{\Delta;\Psi \vdash M \Leftarrow A}$ Term M checks against type A

$$\frac{\Delta;\Psi, x{:}A \vdash M \Leftarrow B}{\Delta;\Psi \vdash \lambda x.M \Leftarrow \Pi x{:}A.B} \qquad \frac{\Delta;\Psi, x{:\!\!\wedge}A \vdash M \Leftarrow B}{\Delta;\Psi \vdash \widehat{\lambda} x.M \Leftarrow A \multimap B}$$

$$\frac{u : (\Phi \vdash P) \in \Delta \quad \Delta;\Psi \vdash \sigma \Leftarrow \Phi \quad \Delta;\overline{\Psi} \vdash [\sigma]_{\overline{\Phi}} P = Q}{\Delta;\Psi \vdash u[\sigma] \Leftarrow Q}$$

$$\frac{\Delta;\Psi_1 \vdash H \Rightarrow A \quad \Delta;\Psi_2 \vdash S > A \Rightarrow P \quad \Delta;\overline{\Psi} \vdash P = Q \quad \Psi = \Psi_1 \bowtie \Psi_2}{\Delta;\Psi \vdash H \cdot S \Leftarrow Q}$$

$\boxed{\Delta;\Psi \vdash H \Rightarrow A}$ Head H synthesizes a type A

$$\frac{c{:}A \in \Sigma \quad \mathsf{unr}(\Psi)}{\Delta;\Psi \vdash c \Rightarrow A} \qquad \frac{p : (\Phi \vdash \#A) \in \Delta \quad \Delta;\Psi \vdash \sigma \Leftarrow \Phi}{\Delta;\Psi \vdash p[\sigma] \Rightarrow [\sigma]_{\overline{\Phi}} A}$$

$$\frac{\mathsf{unr}(\Psi) \quad x{:}A \in \Psi}{\Delta;\Psi \vdash x \Rightarrow A} \qquad \frac{\mathsf{unr}(\Psi_1) \quad \mathsf{unr}(\Psi_2)}{\Delta;\Psi_1, x{:\!\!\wedge}A, \Psi_2 \vdash x \Rightarrow A}$$

$\boxed{\Delta;\Psi \vdash S > A \Rightarrow P}$ Spine S synthesizes type P

$$\frac{\mathsf{unr}(\Psi)}{\Delta;\Psi \vdash \epsilon > P \Rightarrow P} \qquad \frac{\Delta;\overline{\Psi} \vdash M \Leftarrow A \quad \Delta;\Psi \vdash S > [M/x]_A B \Rightarrow P}{\Delta;\Psi \vdash M\,;S > \Pi x{:}A.B \Rightarrow P}$$

$$\frac{\Delta;\Psi_1 \vdash M \Leftarrow A \quad \Delta;\Psi_2 \vdash S > B \Rightarrow P \quad \Psi = \Psi_1 \bowtie \Psi_2}{\Delta;\Psi \vdash M \,\hat{;}\, S > A \multimap B \Rightarrow P}$$

Fig. 7. Typing rules for terms

with domain Φ and range Ψ_1 and M is a well-typed term in the context Ψ_2. The substitution σ, M is well-typed with domain $\Phi, x{:\!\!\wedge}A$ and range Ψ, if σ is a substitution from Φ to Ψ and for some context Ψ', $\overline{\Psi} = \overline{\Psi'}$, M is a well-typed term in the context Ψ'. This last rule, extending the substitution domain by an unavailable variable, is perhaps a little surprising. Intuitively we may want to skip the unavailable variable of a substitution. This would however mean that we have to perform not only context splitting, but also substitution splitting when defining the operation of simultaneous substitution. An alternative is to use an arbitrary term M to be substituted for this unavailable variable, as the typing rules ensure it will never actually occur in the term in which we substitute. When establishing termination of type-checking, it is then important that M type checks in a context that can be generated from the one we already have. We ensure this with a side condition $\overline{\Psi} = \overline{\Psi'}$. By enforcing that the unrestricted parts of Ψ and Ψ' are equal we limit the choices that we have for Ψ' deciding which linear variables to take (linear) and which to drop (unavailable), and deciding on the index of context variable.

$\boxed{\Delta; \Psi \vdash \sigma \Leftarrow \Phi}$ Substitution σ maps variables in Φ to variables in Ψ

$$\frac{\mathsf{unr}(\Psi)}{\Delta; \Psi \vdash \cdot \Leftarrow \cdot} \qquad \frac{\mathsf{unr}(\Gamma)}{\Delta; \psi_m, \Gamma \vdash \mathsf{id}_\psi \Leftarrow \psi_m}$$

$$\frac{\Delta; \Psi \vdash \sigma \Leftarrow \Phi \quad \Delta; \overline{\Psi} \vdash M \Leftarrow [\sigma]_{\overline{\Phi}} A}{\Delta; \Psi \vdash \sigma, M \Leftarrow \Phi, x{:}A}$$

$$\frac{\Delta; \Psi_1 \vdash \sigma \Leftarrow \Phi \quad \Delta; \Psi_2 \vdash M \Leftarrow [\sigma]_{\overline{\Phi}} A \quad \Psi = \Psi_1 \bowtie \Psi_2}{\Delta; \Psi \vdash \sigma, M \Leftarrow \Phi, x\hat{:}A}$$

$$\frac{\Delta; \Psi \vdash \sigma \Leftarrow \Phi \quad \overline{\Psi} = \overline{\Psi'} \quad \Delta; \Psi' \vdash M \Leftarrow [\sigma]_{\overline{\Phi}} A}{\Delta; \Psi \vdash \sigma, M \Leftarrow \Phi, x\check{:}A}$$

Fig. 8. Typing rules for substitutions

When considering an identity substitution id_ψ, we allow for some ambiguity: we can use any ψ_m for both the domain and range of id_ψ. Upon meta-substitution, all instantiations of ψ_m will have the same names and types of variables; the only thing differentiating them will be their status (intuitionistic, linear or unavailable). Since substitutions do not store information about the status of variables they substitute for (this information is available only in the domain and range), the constructed identity substitution will be the same regardless of the initial choice of ψ_m – it will however have a different type.

The observation above has a more general consequence, allowing us to avoid substitution splits when defining the operation of hereditary substitution: if a substitution in LINCX transforms context Φ to context Ψ, it does so also for their unrestricted fragments.

Lemma 2. *If* $\Delta; \Psi \vdash \sigma \Leftarrow \Phi$ *then* $\Delta; \overline{\Psi} \vdash \sigma \Leftarrow \overline{\Phi}$.

3.4 Hereditary Substitution

Next we will characterise the operation of hereditary substitution, which allows us to consider only normal forms in our grammar and typing rules, making the decidability of type-checking easy to establish.

As usual, we annotate hereditary substitutions with an approximation of the type of the term we substitute for to guarantee termination.

$$\text{Type approximations } \alpha, \beta ::= a \mid \alpha \to \beta \mid \alpha \multimap \beta$$

We then define the dependency erasure operator $(-)^-$ as follows:

$$\boxed{A^- = \alpha} \qquad \alpha \text{ is a type approximation of } A$$

$$\begin{aligned}(a \cdot S)^- &= a \\ (\Pi x{:}A.B)^- &= A^- \to B^- \\ (A \multimap B)^- &= A^- \multimap B^-\end{aligned}$$

We will sometimes tacitly apply the dependency erasure operator $(-)^-$ in the following definitions. Hereditary single substitution for LINCX is standard and closely follows [7], since linearity does not induce any complications. When executing the current substitution would create redexes, we proceed by hereditarily performing another substitution. This reduction operation is defined as:

$$\boxed{\mathsf{reduce}(M : \alpha, S) = N} \quad N \text{ is the result of reducing } M \text{ applied to the spine } S$$

$$\begin{aligned}\mathsf{reduce}(\lambda x.M : \alpha \to \beta, (N\,; S)) &= \mathsf{reduce}([N/x]_\alpha M : \beta, S) \\ \mathsf{reduce}(\widehat{\lambda} x.M : \alpha \multimap \beta, (N \mathbin{\hat{;}} S)) &= \mathsf{reduce}([N/x]_\alpha M : \beta, S) \\ \mathsf{reduce}(R : a, \epsilon) &= R \\ \mathsf{reduce}(M : \alpha, S) &= \bot\end{aligned}$$

Termination can be readily established:

Theorem 1 (Termination of hereditary single substitution).
The hereditary substitutions $[M/x]_\alpha(N)$ *and* $\mathsf{reduce}(M : \alpha, S)$ *terminate, either by failing or successfully producing a result.*

The following theorem provides typing for the hereditary substitution. We use J to stand for any of the forms of judgments defined above.

Theorem 2 (Hereditary single substitution property).

1. *If* $\Delta; \overline{\Psi} \vdash M \Leftarrow A$ *and* $\Delta; \Psi, x{:}A \vdash J$ *then* $\Delta; \Psi \vdash [M/x]_A J$.
2. *If* $\Delta; \Psi_1 \vdash M \Leftarrow A$, $\Delta; \Psi_2, x \hat{:} A \vdash J$ *and* $\Psi = \Psi_1 \bowtie \Psi_2$ *then* $\Delta; \Psi \vdash [M/x]_A J$
3. *If* $\Delta; \Psi_1 \vdash M \Leftarrow A$, $\Delta; \Psi_2 \vdash S > A \Rightarrow B$, $\Psi = \Psi_1 \bowtie \Psi_2$ *and* $\mathsf{reduce}(M : A^-, S) = M'$ *then* $\Delta; \Psi \vdash M' \Leftarrow B$

We can easily generalize hereditary substitution to simultaneous substitution. We focus here on the simultaneous substitution in a canonical terms (see Fig. 9). Hereditary simultaneous substitution relies on a lookup function that is defined below. Note that $(\sigma, M)_{\Psi, x \hat{:} A}(x) = \bot$, since we assume x to be unavailable in the domain of σ.

$$\boxed{\sigma_\Psi(x)} \qquad \text{Variable lookup}$$

$$\begin{aligned}(\sigma, M)_{\Psi, x:A}(x) &= M : A^- \\ (\sigma, M)_{\Psi, x \hat{:} A}(x) &= M : A^- \\ (\sigma, M)_{\Psi, y:A}(x) &= \sigma_\Psi(x) \qquad \text{where } y \neq x \\ (\sigma, M)_{\Psi, y \hat{:} A}(x) &= \sigma_\Psi(x) \qquad \text{where } y \neq x \\ \sigma_\Psi(x) &= \bot\end{aligned}$$

$\boxed{[\sigma]^{\widetilde{\Phi}}_{\Psi} M}$	Substitution of the variables of Ψ in a canonical term (leaving elements of $\widetilde{\Phi}$ unchanged)	
$[\sigma]^{\widetilde{\Phi}}_{\Psi}(\lambda y.N)$	$= \lambda y.N'$	where $[\sigma]^{\widetilde{\Phi},y}_{\Psi} N = N'$, choosing $y \notin \Psi, y \notin \mathsf{FV}(\sigma)$
$[\sigma]^{\widetilde{\Phi}}_{\Psi}(\widehat{\lambda} y.N)$	$= \widehat{\lambda} y.N'$	where $[\sigma]^{\widetilde{\Phi},\widetilde{y}}_{\Psi} N = N'$, choosing $y \notin \Psi, y \notin \mathsf{FV}(\sigma)$
$[\sigma]^{\widetilde{\Phi}}_{\Psi}(u[\tau])$	$= u[\tau']$	where $[\sigma]^{\widetilde{\Phi}}_{\Psi} \tau = \tau'$
$[\sigma]^{\widetilde{\Phi}}_{\Psi}(c \cdot S)$	$= c \cdot S'$	where $[\sigma]^{\widetilde{\Phi}}_{\Psi} S = S'$
$[\sigma]^{\widetilde{\Phi}}_{\Psi}(x \cdot S)$	$= \mathsf{reduce}(M : \alpha, S')$	where $\Psi = \Psi_1 \bowtie \Psi_2$ and $x \notin \widetilde{\Phi}$ and $\sigma_{\Psi_1}(x) = M : \alpha$ and $[\sigma]^{\widetilde{\Phi}}_{\Psi_2} S = S'$
$[\sigma]^{\widetilde{\Phi}}_{\Psi}(y \cdot S)$	$= y \cdot S'$	where $y \in \widetilde{\Phi}$ and $[\sigma]^{\widetilde{\Phi}}_{\Psi} S = S'$
$[\sigma]^{\widetilde{\Phi}}_{\Psi}(y \cdot S)$	$= y \cdot S'$	where $\widetilde{y} \in \widetilde{\Phi}$, and $[\sigma]^{\widetilde{\Phi}\setminus\widetilde{y}}_{\Psi} S = S'$
$[\sigma]^{\widetilde{\Phi}}_{\Psi}(p[\tau] \cdot S)$	$= p[\tau'] \cdot S'$	where $\Psi = \Psi_1 \bowtie \Psi_2$, and $\widetilde{\Phi} = \widetilde{\Phi}_1 \bowtie \widetilde{\Phi}_2$ and $[\sigma]^{\widetilde{\Phi}_1}_{\Psi_1} \tau = \tau'$ and $[\sigma]^{\widetilde{\Phi}_2}_{\Psi_2} S = S'$

Fig. 9. Simultaneous substitution

Unlike many previous formulations of contextual LF, we do not allow substitutions to be directly extended with variables. Instead, following Cave and Pientka's more recent approach [7], we require that substitutions must be extended with η-long terms, thus guaranteeing unique normal forms for substitutions. For this reason, we maintain a list of variable names and statuses which are not to be changed, $\widetilde{\Phi}$ in $[\sigma]^{\widetilde{\Phi}}_{\Psi}$. This list gets extended every time we pass through a lambda expression. We use it when substituting in $y \cdot S$ – if $y \in \widetilde{\Phi}$ or $\widehat{y} \in \widetilde{\Phi}$ we simply leave the head unchanged. It is important to preserve not only the name of the variable, but also its status (linear, intuitionistic or unavailable), since we sometimes have to perform a split on $\widetilde{\Phi}$. Such split works precisely like one on complete contexts, since types play no role in context splitting.

As simultaneous substitution is a transformation of contexts, it is perhaps not surprising that it becomes more complex in the presence of context splitting. Consider for instance the case where we push the substitution σ through an expression $p[\tau] \cdot S$. While σ has domain Ψ (and is ignoring variables from $\widetilde{\Phi}$) and $p[\tau] \cdot S$ is well-typed in (Ψ, Φ), the closure $p[\tau]$ is well-typed in a context (Ψ_1, Φ_1) and the spine S is well-typed in a context (Ψ_2, Φ_2) where $\Psi = \Psi_1 \bowtie \Psi_2$ and $\Phi = \Phi_1 \bowtie \Phi_2$. As a consequence, $[\sigma]^{\widetilde{\Phi}}_{\Psi} \tau$ and $[\sigma]^{\widetilde{\Phi}}_{\Psi} S$ would be ill-typed, however $[\sigma]^{\widetilde{\Phi_1}}_{\Psi_1} \tau$ and $[\sigma]^{\widetilde{\Phi_2}}_{\Psi_2} S$ will work well. Notice that it is only the domain of the substitution that we need to split, not the substitution itself.

Similarly to the case for hereditary single substitution, the theorem below provides typing for simultaneous substitution.

Theorem 3 (Simultaneous substitution property).
If $\Delta; \Psi \vdash J$ *and* $\Delta; \Phi \vdash \sigma \Leftarrow \Psi$ *then* $\Delta; \Phi \vdash [\sigma]_{\Psi} J$.

3.5 Decidability of Type Checking in Contextual Linear LF

In order to establish a decidability result for type checking, we observe that the typing judgments are syntax directed. Further, when a context split is necessary (e.g. when checking $\Delta, \Psi \vdash \sigma, M \Leftarrow \Phi, x\hat{:}A$), it is possible to enumerate all the possible correct splits (all Ψ_1, Ψ_2 such that $\Psi = \Psi_1 \bowtie \Psi_2$). For exactly one of them it will hold that $\Delta; \Psi_1 \vdash \sigma \Leftarrow \Phi$ and $\Delta; \Psi_2 \vdash M \Leftarrow [\sigma]_{\overline{\Phi}} A$. Finally, in the $\Delta, \Psi \vdash \sigma, M \Leftarrow \Phi, x\check{:}A$ case, thanks to explicit mention of all the variables (including unavailable ones), we can enlist all possible contexts Ψ' well-formed under Δ and such that $\overline{\Psi} = \overline{\Psi'}$.

Theorem 4 (Decidability of type checking). *Type checking is decidable.*

3.6 Lincx's Meta-Language

To use contextual linear LF as an index language in Beluga, we have to be able to lift Lincx objects to meta-types and meta-objects and the definition of the meta-substitution operation. We are basing our presentation on one for contextual LF [6].

Fig. 4 presents the meta-language of Lincx. Meta-objects are either contextual objects or contexts. The former may be instantiations to parameter variables $p : (\Psi \vdash \#A)$ or meta-variables $u : (\Psi \vdash P)$. These objects are written $\widetilde{\Psi}.R$ where $\widetilde{\Psi}$ denotes a list of variables obtained by dropping all the type information from the declaration, but retaining the information about variable status (intuitionistic, linear or unavailable).

$$
\begin{array}{ll}
\boxed{\widetilde{\Psi}} & \text{Name and status of variables from } \Psi \\
\widetilde{\cdot} & = \cdot \\
\widetilde{\psi_m} & = \psi_m \\
\widetilde{\Psi, x{:}A} & = \widetilde{\Psi}, x \\
\widetilde{\Psi, x\hat{:}A} & = \widetilde{\Psi}, \widehat{x} \\
\widetilde{\Psi, x\check{:}A} & = \widetilde{\Psi}, \check{x}
\end{array}
$$

Contexts as meta-objects are used to instantiate context variables $\psi_i : G$. When constructing those we must exercise caution, as we need to ensure that no linear variable is used in two contexts that are, at any point, joined. At the same time, instantiations for context variables differing only in the index (ψ_i and ψ_j) have to use precisely the same variable names and their unrestricted fragments have to be equal. It is also important to ensure that the constructed context is of a correct schema G. Schemas describe possible shapes of contexts, and each schema element can be either linear ($\lambda(\overrightarrow{x_i{:}A_i}).\widehat{A}$) or intuitionistic ($\lambda(\overrightarrow{x_i{:}A_i}).A$). This can be extended to also allow combinations of linear and intuitionistic schema elements.

We now give rules for a well-formed meta-context Δ (see Fig. 10). It is defined on the structure of Δ and is mostly straightforward. As usual, we assume the

$$\boxed{\vdash \Delta\ \mathsf{mctx}} \qquad \Delta \text{ is a valid meta-context}$$

$$\frac{}{\vdash \cdot\ \mathsf{mctx}} \qquad \frac{\vdash \Delta\ \mathsf{mctx} \quad \Delta; \overline{\Psi} \vdash P\ \mathsf{type}}{\vdash \Delta, u : (\Psi \vdash P)\ \mathsf{mctx}}$$

$$\frac{\vdash \Delta\ \mathsf{mctx} \quad \Delta; \overline{\Psi} \vdash A\ \mathsf{type}}{\vdash \Delta, p : (\Psi \vdash \#A)\ \mathsf{mctx}} \qquad \frac{\vdash \Delta\ \mathsf{mctx} \quad i \in \mathcal{I}}{\vdash \Delta, \psi_i : G\ \mathsf{mctx}}\ \star$$

Fig. 10. Well-formed meta-contexts

names we choose are fresh. The noteworthy case arises when we extend Δ with a context variable ψ_i. Because all context variables ψ_j will describe parts of the same context, we require their schemas to be the same. This side condition ($\star$) can be formally stated as: $\forall j.\psi_j \in \mathsf{dom}(\Delta) \to \psi_j : G \in \Delta$. Moreover, to avoid manually ensuring that indices of context variables do not cross, we require that leaf context variables use elements of the carrier set $i \in \mathcal{I}$ (i.e. they are formed without using the $\bowtie$ operation).

Typing of meta-terms is straightforward and follows precisely the schema presented in previous work.

$$\boxed{\Psi \perp_\psi \Theta} \qquad \text{Context } \Psi \text{ is linearly disjoint from the range of } \Theta \text{ for } \psi_j$$

$$\frac{}{\Psi \perp_\psi (\cdot)} \qquad \frac{\Psi \perp_\psi \Theta \quad \Psi' = \Psi \bowtie \Psi_j}{\Psi \perp_\psi (\Theta, \Psi_j/\psi_j)} \qquad \frac{\Psi \perp_\psi \Theta \quad X \neq \psi_j}{\Psi \perp_\psi (\Theta, C/X)}$$

$$\boxed{\Delta \vdash \Theta \Leftarrow \Delta'} \qquad \Theta \text{ has domain } \Delta' \text{ and range } \Delta$$

$$\frac{}{\Delta \vdash \cdot \Leftarrow \cdot} \qquad \frac{\Delta \vdash \Theta \Leftarrow \Delta' \quad \Delta \vdash \Psi_i \Leftarrow G \quad \Psi_i \perp_\psi \Theta}{\Delta \vdash \Theta, \Psi_i/\psi_i \Leftarrow \Delta', \psi_i : G}$$

$$\frac{\Delta \vdash \Theta \Leftarrow \Delta' \quad \Delta \vdash C \Leftarrow [\![\Theta]\!]_{\Delta'} U}{\Delta \vdash \Theta, C/X \Leftarrow \Delta', X : U}$$

Fig. 11. Typing rules for meta-substitutions

Because of the interdependencies when substituting for context variables, we diverge slightly from standard presentations of typing of meta-substitutions.

First, we do not at all consider single meta-substitutions, as they would be limited only to parameter and meta-variables. In the general case it is impossible to meaningfully substitute only one context variable, as this would break the invariant that all instantiations of context variables share variable names and the intuitionistic part of the context.

Second, the typing rules for the simultaneous meta-substitution (see Fig. 11) are specialized in the case of substituting for a context variable. When extending Θ with an instantiation Ψ_i for a context variable $\psi_i : G$, we first verify that context Ψ_i has the required schema G. We also have to check that Ψ_i can be joined with *any other* instantiation Ψ_j for context variable ψ_j already present in Θ (that is, $\Psi_i \perp_\psi \Theta$). This is enough to ensure the desired properties of meta-substitution for context variables.

We can now define the simultaneous meta-substitution. The operation itself is straightforward, as linearity does not complicate things on the meta-level. What is slightly more involved is the variable lookup function.

$\boxed{\Theta_\Delta(X)}$ Contextual variable lookup

$$
\begin{array}{lll}
(\Theta, \Psi/\psi_i)_{\Delta,\psi_i:G}(\psi_\epsilon) & = \overline{\Psi} & \\
(\Theta, \Psi/\psi_i)_{\Delta,\psi_i:G}(\psi_i) & = \Psi & \\
(\Theta, \Psi/\psi_i)_{\Delta,\psi_i:G}(\psi_m) & = \Phi & \text{where } \Phi = \Psi \bowtie \Psi' \text{ and } m = i \bowtie n \\
 & & \text{and } \Theta_\Delta(\psi_n) = \Psi' \\
(\Theta, \Psi/\psi_i)_{\Delta,\psi_i:G}(\psi_m) & = \Theta_\Delta(\psi_m) & \text{where } i \perp_\psi m \\
(\Theta, C/X)_{\Delta,X:U}(X) & = C : U & \\
(\Theta, C/Y)_{\Delta,Y:\_}(X) & = \Theta_\Delta(X) & \text{where } Y \neq X \\
\Theta_\Delta(X) & = \bot &
\end{array}
$$

On parameter and meta-variables it simply returns the correct meta-object, to which the simultaneous substitution from the corresponding closure is then applied. The lookup is a bit more complicated for context variables, since Θ only contains substitutions for leaf context variables ψ_i. For arbitrary ψ_m we must therefore deconstruct the index $m = i_1 \bowtie \cdots \bowtie i_k$ and return $\Theta_\Delta(\psi_{i_1}) \bowtie \cdots \bowtie \Theta_\Delta(\psi_{i_k})$. Finally, for ψ_ϵ we simply have to find any Ψ/ψ_i in Θ and return $\overline{\Psi}$ – the typing rules for Θ ensure that the choice of ψ_i is irrelevant, as the unrestricted part of the substituted context is shared.

Theorem 5 (Simultaneous meta-substitution property).
If $\Delta \vdash \Theta \Leftarrow \Delta'$ *and* $\Delta'; \Psi \vdash J$, *then* $\Delta; [\![\Theta]\!]_{\Delta'} \Psi \vdash [\![\Theta]\!]_{\Delta'} J$.

3.7 Writing Programs about LINCX Objects

We sketch here why LINCX is a suitable index language for writing programs and proofs. In [29], Thibodeau et.al describe several requirements for plugging in an index language into the (co)inductive foundation for writing programs and proofs about them. They fall into three different classes. We will briefly touch on each one.

First, it requires that the index domain satisfies meta-substitution properties that we also prove for LINCX. Second, comparing two objects should be decidable. We satisfy this criteria, since we only characterize $\beta\eta$-long canonical forms and equality reduces to syntactic equality. The third criterion is unification of index

objects. While we do not describe a unification algorithm for Lincx objects, we believe it is a straightforward extension of A. Schack-Nielsen and C. Schürmann's work [27]. Finally, we require a notion of coverage of Lincx objects which is a straightforward extension of B. Pientka and A. Abel's approach [22].

4 Mechanization of Lincx

We have mechanized key properties of our underlying theory in the proof assistant Beluga. In particular, we encoded the syntax, typing rules of Lincx together with single and simultaneous hereditary substitution operations in the logical framework LF relying on HOAS encodings to model binding. Our encoding is similar to C. Martens and K. Crary's [15] of LF in LF, but we also handle meta-variables and simultaneous substitutions. Since Beluga only intrinsically supports intuitionistic binding structures and contexts, linearity must be enforced separately. We do this through an explicit context of variable declarations, connecting each variable to a flag and a type. To model contexts with context variable indices we use a binary encoding. The implementation of Lincx in Beluga was crucial to arrive at our understanding of modelling context variables using commutative monoids.

As mentioned in Section 3.2, the context variable indices take context splitting into account by describing elements from a countably infinite set $\mathcal{I}$, along with a neutral element and a join operation that is commutative and associative. We implement these indices using binary strings, where ϵ is the empty string, and a string with a single positive bit represents a leaf-level variable. In other words, through this abstraction, every context variable in Δ is a binary string with a single positive bit. A. Schack-Nielsen [26] uses a similar encoding for managing flags for linear, unrestricted, and unavailable assumptions in concrete contexts. Our encoding lifts these ideas to modelling context variables. We then implement the $\bowtie$ operation as a binary OR operation which fails when the two strings have a common positive (for instance a join between 001 and 011 would fail). The following describes the join of M and N, forming K.

```
LF bin_or : bin → bin → bin → type =
   | bin_or_nil_l : bin_or nil M M
   | bin_or_nil_r : bin_or M nil M
   | bin_or_l     : bin_or M N K → bin_or (cons one M) (cons zero N) (cons one K)
   | bin_or_r     : bin_or M N K → bin_or (cons zero M) (cons one N) (cons one K)
   | bin_or_zero  : bin_or M N K → bin_or (cons zero M) (cons zero N) (cons zero K)
;
```

We then proceed to prove commutativity, associativity and uniqueness of `bin_or`. Finally, we mechanized the proofs of the properties about our equational theory of context joins as total functions in Beluga. In particular, we mechanized proofs of lemma 1 and 2. Here we take advantage of Beluga's first-class contexts and in the base cases rely on the commutativity and associativity properties of the binary encoding of context variable indices. We note that context equality is entirely syntactic and can thus be defined simply in terms of reflection.

Although we had to model our mixed contexts of unrestricted and linear assumptions explicitly, Beluga's support for encoding formal systems using

higher-order abstract syntax still significantly simplified our definitions of typing rules and hereditary substitution operation. In particular, it allowed us to elegantly model variable bindings in abstractions and Π-types.

Inductive properties about typing and substitution are implemented as recursive functions in Beluga. Many of the proofs in this paper become fairly tedious and complex on paper and mechanizing Lincx therefore helps us build trust in our foundation. Given the substantial amount of time and lines of code we devote to model contexts and context joins, our mechanization also demonstrate the value Lincx can bring to mechanizing linear systems or more generally systems that work with resources. [4]

5 Related Work

The idea of using logical framework methodology to build a specification language for linear logic dates back three decades, beginning with Cervesato's and Pfenning's linear logical framework LLF [9] providing $\multimap$, & and $\top$ operators from intuitionistic linear logic, the maximal set of connectives for which unique canonical forms exist. The idea was later expanded to the concurrent logical framework CLF [31], which uses a monad to encapsulate less well-behaved operators. The quest to design meta-logics that allow us to reason about linear logical frameworks has been marred with difficulties in the past.

In proof theory, McDowell and Miller [18, 19] and later Gacek et.al. [13] propose a two-level approach to reason about formal systems where we rely on a first-order sequent calculus together with inductive definitions and induction on natural numbers as a meta-reasoning language. We encode our fomal system in a specification logic that is then embedded in the first-order sequent calculus, the reasoning language. The design of the two-level approach is in principle modular and in fact McDowell's PhD thesis [18] describes a linear specification logic. However the context of assumptions is encoded as a list explicitly in this approach. As a consequence, we need to reason modulo the equational properties of context joins and we may need to prove properties about the uniqueness of assumptions. Such bureaucratic reasoning then still pollutes our main proof.

In type theory, McCreight and Schürmann [17] give a tailored meta-logic $\mathcal{L}_\omega^+$ for linear LF, which is an extension of the meta-logic for LF [28]. While $\mathcal{L}_\omega^+$ also characterize partial linear derivations using contextual objects that depend on a linear context, the approach does not define an equational theory on contexts and context variables. It also does not support reasoning about contextual objects modulo such an equational theory. In addition $\mathcal{L}_\omega^+$ does not cleanly separate the meta-theoretic (co)inductive reasoning about linear derivations from specifying and modelling the linear derivations themselves. We believe the modular design of Beluga, i.e. the clean separation of representing and modelling specifications and derivations on one hand and reasoning about such derivations

[4] Lincx Mechanization: `https://github.com/Beluga-lang/Beluga/tree/master/examples/lincx_mechanization`

on the other, offers many advantages. In particular, it is more robust and also supports extensions to (co)inductive definitions [6, 29].

The hybrid logical framework HLF by Reed [25] is in principle capable to support reasoning about linear specifications. In HLF, we reason about objects that are valid at a specific world, instead of objects that are valid within a context. However, contexts and worlds seem closely connected. Most recently Bock and Schürmann [4] propose a contextual logical framework XLF. Similarly to LINCX, it is also based on contextual modal type theory with first-class contexts. However, context variables have a strong nominal flavor in their system. In particular, Bock and Schürmann allow multiple context variables in the context and each context variable is associated with a list of variable names (and other context variable domains) from which it must be disjoint – otherwise the system is prone to repetition of linear variables upon instantiation.

On a more fundamental level the difference between HLF and XLF on the one hand and our approach on the other is how we think about encoding meta-theoretic proofs. HLF and XLF follow the philosophy of Twelf system and encoding proofs as relations. This makes it sometimes challenging to establish that a given relation constitutes an inductive proof and hence both systems have been rarely used to establish such meta-theoretic proofs. More importantly, the proof-theoretic strength of this approach is limited. For example, it is challenging to encode formal systems and proofs that rely on (co)inductive definitions such as proofs by logical relations and bisimulation proofs within the logical framework itself. We believe the modular design of separating cleanly between LINCX as a specification framework and embedding LINCX into the proof and programming language BELUGA provides a simpler foundation for representing the meta-theory of linear systems. Intuitively, meta-proofs about linear systems only rely on linearity to model the linear derivations – however the reasoning about these linear derivation trees is not linear, but remains intuitionistic.

6 Conclusion and Future Work

We have presented LINCX, a linear contextual modal logical framework with first-class contexts as a foundation to model linear systems and derivations. In particular, LINCX satisfies the necessary requirements to serve as a specification and index language for BELUGA and hence provides a suitable foundation for implementing proofs about (linear) derivation trees as recursive functions. We have also mechanized the key equational properties of context joins in BELUGA. This further increases our confidence in our development.

There is a number of research questions that naturally arise and we plan to pursue in the future. First, we plan to extend LINCX with additional linear connectives such as $\top$ and $A\&B$. These additional connectives are for example present in [9]. We omitted them here to concentrate on modelling context joins and their equational theory, but we believe it is straightforward to add them.

Dealing with first-class contexts in the presence of additive operators is more challenging, as they may break canonicity. We plan to follow the approach in CLF

[31] enclosing them into a monad to control their behaviour. Having also additive operators would allow us to for example model the meta-theory of session type systems [5] and reason about concurrent computation. Further we plan to add first-class substitution variables [7] to Lincx. This woud allow us to abstractly describe relations between context. This seems particularly important as we allow richer schemas definitions that model structured sequences.

Last but not least, we would like to implement Lincx as a specification language for Beluga to enable reasoning about linear specifications in practice.

References

1. Belanger, O.S., Monnier, S., Pientka, B.: Programming type-safe transformations using higher-order abstract syntax. In: Gonthier, G., Norrish, M. (eds.) 3rd International Conference on Certified Programs and Proofs (CPP'13). pp. 243–258. Lecture Notes in Computer Science (LNCS 8307), Springer (2013)
2. Bengtson, J., Jensen, J.B., Birkedal, L.: Charge! - A framework for higher-order separation logic in Coq. In: Beringer, L., Felty, A.P. (eds.) Third International Conference on Interactive Theorem Proving (ITP'12). pp. 315–331. Lecture Notes in Computer Science (LNCS 7406), Springer (2012)
3. Berdine, J., O'Hearn, P.W., Reddy, U.S., Thielecke, H.: Linear continuation-passing. Higher-Order and Symbolic Computation 15(2-3), 181–208 (2002)
4. Bock, P.B., Schürmann, C.: A contextual logical framework. In: 20th International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR'15). pp. 402–417. Lecture Notes in Computer Science (LNCS 9450), Springer (2015)
5. Caires, L., Pfenning, F.: Session types as intuitionistic linear propositions. In: Gastin, P., Laroussinie, F. (eds.) 21th International Conference on Concurrency Theory (CONCUR'10). pp. 222–236. Lecture Notes in Computer Science (LNCS 6269), Springer (2010)
6. Cave, A., Pientka, B.: Programming with binders and indexed data-types. In: 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'12). pp. 413–424. ACM (2012)
7. Cave, A., Pientka, B.: First-class substitutions in contextual type theory. In: 8th ACM SIGPLAN International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP'13). pp. 15–24. ACM (2013)
8. Cave, A., Pientka, B.: A case study on logical relations using contextual types. In: Cervesato, I., K.Chaudhuri (eds.) 10th International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP'15). pp. 18–33. Electronic Proceedings in Theoretical Computer Science (EPTCS) (2015)
9. Cervesato, I., Pfenning, F.: A linear logical framework. In: Clarke, E. (ed.) 11th Annual Symposium on Logic in Computer Science. pp. 264–275. IEEE Press, New Brunswick, New Jersey (1996)
10. Cervesato, I., Pfenning, F.: A linear spine calculus. Journal of Logic and Computation 13(5), 639–688 (2003)
11. Danvy, O., Filinski, A.: Representing control: A study of the CPS transformation. Mathematical Structures in Computer Science 2(4), 361–391 (1992)
12. Fluet, M., Morrisett, G., Ahmed, A.J.: Linear regions are all you need. In: Sestoft, P. (ed.) 15th European Symposium on Programming (ESOP'06). pp. 7–21. Lecture Notes in Computer Science (LNCS 3924), Springer (2006)

13. Gacek, A., Miller, D., Nadathur, G.: A two-level logic approach to reasoning about computations. Journal of Automated Reasoning 49(2), 241–273 (2012)
14. Harper, R., Honsell, F., Plotkin, G.: A framework for defining logics. Journal of the ACM 40(1), 143–184 (January 1993)
15. Martens, C., Crary, K.: LF in LF: Mechanizing the metatheories of LF in Twelf. In: 7th International Workshop on Logical Frameworks and Meta-languages:Theory and Practice (LFMTP'12). pp. 23–32. ACM (2012)
16. McCreight, A.: Practical tactics for separation logic. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) 22nd International Conference on Theorem Proving in Higher Order Logics (TPHOLs'09). pp. 343–358. Lecture Notes in Computer Science (LNCS 5674), Springer (2009)
17. McCreight, A., Schürmann, C.: A meta-linear logical framework. In: 4th International Workshop on Logical Frameworks and Meta-Languages (LFM'04) (2004)
18. McDowell, R.: Reasoning in a Logic with Definitions and Induction. Ph.D. thesis, University of Pennsylvania (1997)
19. McDowell, R.C., Miller, D.A.: Reasoning with higher-order abstract syntax in a logical framework. ACM Transactions on Computational Logic 3(1), 80–136 (2002)
20. Nanevski, A., Pfenning, F., Pientka, B.: Contextual modal type theory. ACM Transactions on Computational Logic 9(3), 1–49 (2008)
21. Pientka, B.: A type-theoretic foundation for programming with higher-order abstract syntax and first-class substitutions. In: 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'08). pp. 371–382. ACM (2008)
22. Pientka, B., Abel, A.: Structural recursion over contextual objects. In: Altenkirch, T. (ed.) 13th International Conference on Typed Lambda Calculi and Applications (TLCA'15). pp. 273–287. Leibniz International Proceedings in Informatics (LIPIcs) of Schloss Dagstuhl (2015)
23. Pientka, B., Cave, A.: Inductive Beluga:Programming Proofs (System Description). In: Felty, A.P., Middeldorp, A. (eds.) 25th International Conference on Automated Deduction (CADE-25). pp. 272–281. Lecture Notes in Computer Science (LNCS 9195), Springer (2015)
24. Pientka, B., Dunfield, J.: Beluga: a framework for programming and reasoning with deductive systems (System Description). In: Giesl, J., Haehnle, R. (eds.) 5th International Joint Conference on Automated Reasoning (IJCAR'10). pp. 15–21. Lecture Notes in Artificial Intelligence (LNAI 6173), Springer (2010)
25. Reed, J.: A hybrid logical framework. Ph.D. thesis, Carnegie Mellon (2009)
26. Schack-Nielsen, A.: Implementing Substructural Logical Frameworks. Ph.D. thesis, IT University of Copenhagen (2011)
27. Schack-Nielsen, A., Schürmann, C.: Pattern unification for the lambda calculus with linear and affine types. In: Crary, K., Miculan, M. (eds.) International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP'10). Electronic Proceedings in Theoretical Computer Science (EPTCS), vol. 34, pp. 101–116 (Jul 2010)
28. Schürmann, C.: Automating the Meta Theory of Deductive Systems. Ph.D. thesis, Department of Computer Science, Carnegie Mellon University (2000), CMU-CS-00-146
29. Thibodeau, D., Cave, A., Pientka, B.: Indexed codata. In: Garrigue, J., Keller, G., Sumii, E. (eds.) 21st ACM SIGPLAN International Conference on Functional Programming (ICFP'16). pp. 351–363. ACM (2016)

30. Walker, D., Watkins, K.: On regions and linear types. In: Pierce, B.C. (ed.) 6th ACM SIGPLAN International Conference on Functional Programming (ICFP'01). pp. 181–192. ACM (2001)
31. Watkins, K., Cervesato, I., Pfenning, F., Walker, D.: A concurrent logical framework I: Judgments and properties. Tech. Rep. CMU-CS-02-101, Department of Computer Science, Carnegie Mellon University (2002)