Lincx
A Linear Logical Framework with First-Class Contexts

OTIS, Shawn

Master of Science

Computer Science

McGill University

Montreal,Quebec

2017-04-15

A thesis submitted to McGill University in partial fulfillment of the requirements of the degree of Master of Science

ACKNOWLEDGEMENTS

First and foremost, I thank my supervisor Brigitte Pientka for her patience, support and understanding throughout my M.Sc. I also thank her for her supervision, guidance and financial support. Without her help, I would not be half the scholar I am today.

I also thank my coauthors, Agata Murawska, Aina-Linn Georges and Brigitte Pientka, with whom I have worked on the project which led to our submission of this work to ESOP 2016 and which is the basis for this current thesis. I would also like to thank Aina-Linn for being there to bounce off ideas back and forth.

Next, I would like to thank the other members of my lab : Andrew Cave, Francisco Ferreira, David Thibodeau, Rohan Jacob-Rao and Stefan Knudsen, who have been there for me throughout and answered my numerous questions when the occasion arose. A particular mention must be made for Francisco, whom I have particularly bombarded with questions, and who also supported me morally through the writing of this thesis.

A special mention is to be made to Steven Thephsourinthone, the unofficial member of our lab, who has been there for me throughout and supported me morally.

I would also like to thank my parents and other relatives who have supported me. In particular, I would like to thank my mother, who was always available for me and who helped me through rough patches.

I would also like to thank three people for helping me proof-read parts of my thesis: Adam Angel, Francisco Ferreira and David Sherratt.

My masters degree has been funded by the Fonds Quebecois de Recherche sur la Nature et les Technologies (FQRNT) and by McGill University.

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.

ABRÉGÉ

La logique linéaire represente une structure élégante pour modeler des systèmes impératifs, concurrents et avec des systèmes a états, en représentant un contexte d'hypothèses comme une collection de ressources.Cependant, la mécanisation de la métathéorie de ces systèmes demeure un défi, puisque nous devons gérer et raisonner à propos de contextes d'hypothèses mixtes linéaires et intuitionistiques.

Nous présentons LINCX, une structure logique linéaire et contextuelle avec des contextes mixtes de première classe. LINCX nous permet d'établir des modèles (linéaires) d'arbres de syntaxe abstraits en tant que structures syntactiques qui peuvent dependre d'hypothèses intuitionistiques et linéaires. LINCX peut également servir de fondation pour raisonner à propos de telles structures. LINCXétend la structure logique linéaire LLF avec des contextes (linéaires) de premier ordre et une théorie d'equations d'assemblage de contextes qui peut autrement être très fastidieux et complexe à développer. Cet oeuvre peut également être perçu comme une généralisation du LF contextuel qui supporte les fonctions, les hypothéses et les variables intuitionistiques et linéaires.

Nous décrivons une fondation de la théorie des types décidable pour LINCX qui ne décrit que les formes canoniques et montrons que notre theorie d'équations d'assemblage de contextes est associative et commutative. Finalement, nous donnons un aperçu de comment LINCX peut servir de fondation pratique pour la mécanisation de la métathéorie de systèmes à états.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1
Introduction

1.1 A Need for Formalism

Proofs and formalisms have long been an important aspect of science and mathematics. One of the most common instances of formalisms in mathematics consists in the notion of axiomatization. In fact, we can trace this ideology back to Ancient Greece, where Euclid developped an axiomatic representation of a model of geometry now referred to as Euclidean geometry. In fact, axiomatization is at the base of most fields of mathematics. Whether it be abstract algebra or analysis, the theory is built from axioms and we are interested in studying their implications.

During the early twentieth century, a strong push was made towards formalizing mathematics. In particular, we can look at Hilbert's program, which pushed for the field of mathematics to be made more formal and precise. And while the Hilbert program was shown to be unattainable by Gödel, it nevertheless reinvigorated the idea of having a more precise and formal language to describe mathematics.

An important breakthrough in the realm of formalization has been Gentzen's contribution towards logic and the formalization of both natural deduction and the sequent calculus. These calculi form the basis of modern deductive logic and give us important guiding principles. Moreover, from these stemed an array of calculi upon which modern research is based.

And while the early 20th century focused towards the formalization of mathematics, computer science also expanded significant efforts towards developing models of computation. These models sought to create machines that could recognize languages: sets of words (which are strings of characters from some alphabet). These models range, in order of strength, from finite-deterministic automata, which are equivalent to regular expressions; Push-down automata, which are equivalent to context-free grammars; and finally Turing Machines.

The latter model was developed by Alan Turing in 1936 and is now synonymous with computation. Of course, this model shows some of the shortcomings of computation, in particular the fact that some problems are not decidable. But while Turing Machines are the model we've been using to this day, many other models have been shown to be equivalent to it. In particular, we are concerned with the Lambda Calculus, developed by Alonzo Church in the 1930's.

While Turing Machines are a good model to formalize and visualize computations, the lambda calculus is helpful to describe mathematical concepts more concisely, since it is described purely through the notions of variables, functions and function application. Moreover, the lambda calculus allows both the fields of computer science and mathematics to be reunited under the same formalism. This is seen in part through the simply typed lambda calculus, in which we associate types to terms in the language. While terms can be related to computations, types relate to logical formulae: observed through the Curry-Howard Correspondence. This correspondence ensures a mutually beneficial relation between computer science and mathematics. On the one hand, it allows for safer computation by proving logical formulae serving as guarantees for a program. On the other hand, it also

allows us to create mechanization tools in which we can validate a logical formula by providing a proof term.

Due to this strong correspondence and the benefits linked to it, efforts were made towards new calculi with an associated logic (typing) that could serve as stronger tools. An important milestone is the introduction of intuitionistic type theory by Martin-Löf in 1972. While the Curry-Howard Correspondence was defined over impredicative logic, this type theory extends it to the predicative case, in particular by introducing dependent types, which allow for terms to be embedded inside of types.

All this effort eventually led to the development of logical frameworks, the goal of which is to allow for a simpler mechanization of formal systems. This goal is accomplished by providing a single meta-language with abstractions and primitives for common and recurring concepts, such as variables and assumptions in proofs.

There are multiple benefits to using logical frameworks. In particular, by abstracting over low-level operations and handling them automatically (e.g. substitution), mechanization is easier to maintain, and efforts can be expanded towards the essential aspects of proofs, rather than focusing on bureaucratics.

While there exists different logical frameworks, LF [19] is one of the most salient examples. A conservative extension of LF, the contextual logical framework [28, 30], is an interesting framework to consider. The goal of this system is to support a broad range of common features needed when mechanizing formal systems. In particular, a context of assumptions together with properties about uniqueness of assumptions can be represented abstractly using first-class contexts and context variables [30]; single and simultaneous substitutions together with their equational theory are supported via first-class substitutions

[8, 9]; finally, derivation trees that depend on a context of assumptions can be precisely described via contextual objects [28]. This is of particular importance: 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 inductive proofs by writing recursive programs about them [32, 7].

This ideology leads to a modular and robust design where we cleanly separate the representation of formal systems and derivations from the inductive reasoning about them.

Due to the interesting attributes of the contextual logical framework, we are interested in borrowing some of its features when designing a linear logical framework. This is why we developped Lincx, a logical framework combining both contextual type theory and linear logic.

But what exactly is linear logic, and why are we interested in it?

1.2 Linear Logic

While the framework of classical logic might seem convenient, in the advent of computer science, there has been a strong push towards working with constructive logics, due to its intrinsic relationship with computations. For this reason, many systems have opted to exclude the principle of excluded middles; in particular, intuitionistic logic is such a system. However, intuitionistic logic isn't always a satisfactory answer.

Unfortunately, some aspects of proofs in intuitionistic and classical logic are hidden away. For instance, while there is only one disjunctive and one conjunctive, both can be used in two different forms: An additive and a multiplicative. For this reason, we are interested in a logic which allows us to examine more closely classical and intuitionistic proofs. The distinction between both forms consists in the use of the context, and both coincide due to

structural rules. However, this distinction would allow us to more closely examine a system, and we are thus interested in a logic which discriminates them.

This purpose is satisfied by linear logic, a substructural logic designed by Girard [18], in which assumptions must be used exactly once. From this principle, both the conjunctive and disjunctive constructors are split into two, where we can now observe the multiplicative and additive counterparts as their own constructs. However, simply using these new operators does not allow linear logic to englobe either classical or intuitionistic logic. For this, a new constructor can be added to the language: exponentiation. Exponentiation allows us to use a context of assumptions in an unrestricted manner, simulating the substructural rules. With this new construct, classical and intuitionistic linear logic embed, respectively, classical and intuitionistic logic. Moreover, while classical logic isn't constructive, classical linear logic is.

From this new model (linear logic) also came a new notion of proofs: proof nets [18]. The basic premise being that we create a special graph from our formula, and if this graph respects some properties, then we have a proof. This serves as a more visual and less syntactic notion of proof, and carries less redundant information around, while still remaining as correct and rigorous.

Obviously, while the logical aspect is interesting, we are also interested in computations. This means that we would be interested in a pair of terms and types where the latter corresponds to linear logic. For this purpose, a linear lambda calculus has been developped. Moreover, Pfenning and Cervesato have also developped the linear logical framework LLF [10], which serves as a framework to reason about linear logic and as a starting point for Lincx.

1.3 The Omnipresence of Linear Logic

Now that a basic picture of what linear logic is has been drawn, why exactly are we interested in it? This is due to the fact that, despite linear logic stemming from the simple removal of substructural rules, its consequences are vast and complex. For instance, many new insights and observations have been made by studying linear logic.

There are a few examples demonstrating the usefulness of studying logic and computations from the perspective of linear logic. For instance, different linear representations of intuitionistic implication correspond to different evaluation strategies[21]. Recent work by Pistone also reveals an intrinsic relationship between polymorphism and linear logic [33].

On the other hand, linear logic doesn't only allow for new insights, but can also be used as a potent modelling tool. This is due to the fact that assumptions must be used exactly once, allowing them to model resources. We can quickly take a look at a few of these examples.

First of all, in order to reason about concurrency, there is empirical evidence to suggest that linearity is key. For instance, an important model designed to reason about concurrent systems that has been gaining traction in the last few years is that of Session Types [6]. In this system, concurrent programs are allowed to share variables through a linear channel. Moreover, these variables can be defined as linear, since they often represent resources.

Next, we also have the concurrent logical framework CLF [40], which is based on the linear logical framework LLF [10], extended to deal with concurrent systems using a monadic encapsulation. We are then faced with a system to reason about concurrency in which linearity is an intrinsic property.

Briefly, a few more examples in which linearity plays a key role are non-size increasing computation[20], Electronic Voting protocols[15], Narrative modelling [22].

More ambitiously, a large-scope example of linearity resides in the Rust language [1]. An important feature of this language is that of "ownership", which allows Rust to preserve memory safety. The notion of linearity play an important role in this feature and thus, linear logic would be a prime candidate to formalize and mechanize Rust.

1.4 Contribution and Organization

While substructural frameworks such as LLF provide additional abstractions to elegantly model the behaviour of imperative operations such as updating and deallocating memory [39, 16] and concurrent computation (see for example session types [6]), it has been very challenging to mechanize proofs about LLF specifications, meta-theory being one of the main limitations. In particular, managing mixed contexts of unrestricted and linear assumptions remains a challenge.

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, and unrestricted assumptions present in Ψ should remain present in both Ψ_1 and Ψ_2. This mix of unrestricted and restricted assumptions in turn leads to an intricate equational theory of contexts that often stands in the way of mechanizing linear or separation logics in proof assistants that has spurred the development of specialized tactics [24, 3].

Our main contribution is the design of Lincx, 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 LF 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 [7] and Thibodeau et.al. [38]. 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 *[31].* 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.

In this thesis, we present Lincx, a contextual linear logical framework. The goal of this framework is to formalize, mechanize and reason about linear systems. Lincx has been developped in collaboration with Aina-Linn Georges, Agata Murawska and Brigitte Pientka, and submitted to the European Symposium On Programming (ESOP) of 2017.

The work is organized as follows. In Section 2, we present some preliminaries needed to better understand the subsequent theory. In particular, starting from the simply-typed lambda calculus, adding some constructs from the contextual logical framework. We then

present the simply-typed linear lambda calculus and give some insights leading to our handling of context variables.

In Section 3, we present our language through the use of two examples: code simplification and cps-translation. Then, in Section 4, we define LINCX, first presenting the grammar, then discussing the contexts and context joins operation. We describe the new abstract form of context variables, presented in a binary form. Hereditary single substitution, simultaneous substitutions and typing rules are also defined, together with meta-theoretic properties. In particular the substitution lemma and decidability of type checking are established. In Section 5, we discuss the mechanization of LINCX, done in BELUGA, strenghtening our confidence in LINCX.

Finally, in Section 6, we discuss related work and we conclude in Section 7 by discussing possible extensions of the system.

CHAPTER 2
Preliminaries

Before moving on to the examples and the theory, let us first present some prerequisite theory. In particular, we want to present elements from Contextual Modal Type theory, and the Simply Typed Linear Lambda Calculus. First, we quickly review the simply typed lambda calculus and build some constructs from contextual modal type theory on it, meta-variables and context variables, along with the notion of simultaneous substitution. Next, we present the linear lambda calculus in a simply typed form, presenting two new objects, meta-variables and context variables, along with the notion of a stuck substitution. Finally, we describe one of the big issues raised by joining both theory, and present the insights that allowed us to solve this issue.

2.1 Lambda Calculus

Let us start with a quick review of the simply typed lambda calculus. The syntax of terms and types is as follows:

$$\begin{array}{llll} \text{Terms} & M & ::= & x \mid \lambda x{:}A.M \mid M_1 M_2 \\ \text{Types} & A & ::= & A_1 \to A_2 \mid \top \\ \text{Context} & \Gamma & ::= & \cdot \mid \Gamma, x{:}A \end{array}$$

Variables are represented by lower case letters such as x. They are bound by lambda abstraction $\lambda x{:}A.M$. Application is represented by $M_1 M_2$, where we say M_2 is applied to M_1. This means that M_1 is assumed to be a function binding some variable x, while M_2 is

an instantiation for x. This instantiation will be enacted through a substitution operation. This brings us to the operational semantics of our language, where substitution $M\{M'/x\}$ is read as replacing every occurence of x in M by M'.

$$\frac{}{(\lambda x{:}A.M_1)M_2 \to M_1\{M_2/x\}}$$

$$\frac{M_1 \to M_1'}{M_1M_2 \to\ M_1'M_2} \qquad \frac{M_2 \to M_2'}{M_1M_2 \to\ M_1M_2'}$$

Since we are working with variables in a simply typed setting, we want to describe the type of these variables. In order to do this, we carry a context around, which is simply a list of different variables with their associated type. We note that, while not explicit, it is assumed that no variable appears more than once in a context.

We can now describe our types. In order for the types to terminate, we need some base type, which we call $\top$. This represent our base case in the inductive definition of A. The type $A_1 \to A_2$ represents the type of a lambda expression $\lambda x{:}A_1.M$, where A_1 is the input type, or type of variable x, while A_2 is the output type, or expected type of M after instantiating x. This becomes clearer with the typing rules.

$$\frac{}{\Gamma, x{:}A_1 \vdash x{:}A_1} \quad \frac{\Gamma, x{:}A_1 \vdash M : A_2}{\Gamma \vdash \lambda x{:}A_1.M : A_1 \to A_2}$$

$$\frac{\Gamma \vdash M : A_1 \to A_2 \quad \Gamma \vdash M_2 : A_2}{\Gamma \vdash\ M_1M_2 : A_2}$$

2.2 Contextual Modal Logic

Let us now move on to the contextual modal setting. In this section, we shall extend our calculus with two new constructs: meta-variables and context variables. However, before moving on to these constructs, we first need to present a new notion of substitution:

simultaneous substitution. The reason for this will become clear as we progress through this section.

The idea behind simultaneous substitution is that all the variables we could substitute for are present at the same time. Thus, we don't apply each substitution one after the other, as they appear through beta reductions, but instead cluster them together. The syntax for a simultaneous substitution is as follows:

$$\text{Substitution} \quad \sigma \quad ::= \quad \cdot \mid \sigma, M/x$$

The next logical step is to describe how to apply this simultaneous substitution. Let us thus do this. In the current setting, it is akin to applying single substitution multiple times. We shall, nevertheless, describe it in the standard fashion.

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

$$(\sigma, M/x)(x) = M : A$$

$$(\sigma, M/x')(x) = \sigma(x) \qquad \text{where } x' \neq x$$

$$\cdot(x) = \bot$$

$\boxed{[\sigma]^\Phi M}$ Substitution by σ in a term (leaving elements of Φ unchanged)

$$[\sigma]^\Phi(\lambda x{:}A.M) = \lambda x{:}A.M' \text{ where } [\sigma]^{\Phi, x:A} M = M', \text{ choosing } x \notin \mathsf{FV}(\sigma)$$

$$[\sigma]^\Phi(\ M_1 M_2) \ = \ M_1' M_2' \quad \text{where } [\sigma]^\Phi M_1 = M_1' \text{ and } [\sigma]^\Phi M_2 = M_2'$$

$$[\sigma]^\Phi(x) = M \qquad \text{where } x \notin \Phi \text{ and } \Phi(x) = M : A$$

An interesting aspect of using simultaneous substitution is that it allows us to move between contexts. This notion is made clearer with typing over the simultaneous substitution,

denoted as $\Gamma_1 \vdash \sigma : \Gamma_2$, which states that substitution σ moves us from context Γ_2 to context Γ_1.

$$\frac{}{\Gamma \vdash \cdot : \cdot} \qquad \frac{\Gamma \vdash \sigma : \Gamma' \quad \Gamma \vdash M : A}{\Gamma \vdash \sigma, M/x : \Gamma', x : A}$$

Let us now expand our calculus with meta-variables. A meta variable is denoted as u, however, it will be associated with a "stuck" substitution, which will be made clear later, and thus represented as $u[\sigma]$:

$$\text{Terms} \quad M \quad ::= \quad x \mid \lambda x{:}A.M \mid M_1 M_2 \mid u[\sigma]$$

In order for this meta-variable to be well-described, we need to add a new context, called meta-context, which will keep track of meta-variables, similarly to the relation between contexts and variables. However, due to some peculiarities of meta-variables (the reason for which they are associated with a "stuck" substitution), the typing must also be associated with a context. This new meta-variable provides us with a new abstraction of terms.

$$\text{Meta-Context} \quad \Delta \quad ::= \quad \cdot \mid \Delta, u : \{\Gamma \vdash A\}$$

We quickly note that the typing of the meta-variable is based not only on a type, but is also associated with a specific context. Moreover, when typing a meta-variable, we also rely on the typing of the associated stuck simultaneous substitution.

$$\frac{u[\sigma] : \{\Gamma' \vdash A\} \in \Delta \quad \Gamma \vdash \sigma : \Gamma'}{\Delta; \Gamma \vdash u[\sigma] : A}$$

Let us now revisit our notion of substitution. The question is now: What does it mean to replace variable x in a meta-variable u? The answer is that the substitution cannot be

applied yet, since we do not know what u stands for: the substitution gets stuck, and we add it and apply it to the simultaneous substitution σ. Thus, our simultaneous substitution σ accumulates all the variables that need to be substituted in u once we instantiate it. Let us thus write the definition of a single substitution:

$\boxed{\{M/x\}M'}$ Single substitution over a term

$$\{M/x\}x = M$$

$$\{M/x\}\lambda y{:}A.M' = \lambda y{:}A.\{M/x\}M' \qquad \text{where } y \notin FV(M) \text{ and } x \neq y$$

$$\{M/x\}\ M_1 M_2 = \{M/x\}M_1\{M/x\}M_2$$

$$\{M/x\}u[\sigma] = u[\{M/x\}\sigma]$$

$\boxed{\{M/x\}M'}$ Single substitution over simultaneous substitution

$$\{M/x\}\cdot = [\cdot, M/x]$$

$$\{M/x\}(\sigma, M'/y) = (\{M/x\}\sigma), (\{M/x\}M')/y \qquad \text{where } y \neq x$$

Similarly, we can expand the notion of simultaneous substitution to meta-variables, and thus apply a simultaneous substitution to another simultaneous substitution.

We now move on to context variables, which shall be denoted as ψ. However, since we want them to properly abstract contexts and be sure that our instantiation is valid, we must add a new notion of typing for contexts: schemas. Moreover, a new substitution must be added to our language, the identity substitution id_ψ.

$$
\begin{array}{llll}
\text{Substitution} & \sigma & ::= & \cdot \mid id_\psi \mid \sigma, M/x \\
\text{Context} & \Gamma & ::= & \cdot \mid \psi \mid \Gamma, x{:}A \\
\text{Schema} & G & ::= & \lambda(\overrightarrow{x_i{:}A_i}).A \mid G + \lambda(\overrightarrow{x_i{:}A_i}).A \\
\text{Meta-Context} & \Delta & ::= & \cdot \mid \Delta, u : \{\Gamma \vdash A\} \mid \Delta, \psi : G
\end{array}
$$

The description of a context variable in the meta-context is associated with a schema. A schema describes the possible types of variables in a context with their possible dependencies (described as a vector). Thus, a schema element $\lambda(\overrightarrow{x_i{:}A_i}).A$ represents a variable of type A that can depend on variables x_i of respective types A_i. Finally, we have the new substitution construct id_ψ, which allows us to go from a domain with only ψ to a co-domain containing ψ and other variables. When ψ gets instantiated, this becomes an identity substitution.

We note that one desired property is that all subcontexts have the same schema as the full context. This means, for instance, that we allow the empty context to have any valid schema. Similarly, when extending a context, we only ensure that the new assumption corresponds to one of the possibilities in the schema.

We can now present the typing rules for the contexts, along with the typing of the new substitution:

$$
\frac{}{\Delta; \psi, \Gamma \vdash id_\psi : \psi}
$$

$$
\frac{}{\Delta \vdash \cdot : G} \qquad \frac{\psi : G \in \Delta}{\Delta \vdash \psi : G}
$$

$$
\frac{\Delta \vdash \Gamma : G \quad \lambda(\overrightarrow{x_i{:}A_i}).B \in G \quad \Delta; \Gamma \vdash \sigma : \overrightarrow{(x_i{:}A_i)} \quad [\sigma]B = A}{\Delta \vdash (\Gamma, x{:}A) : G}
$$

Finally, we add meta-substitution to our system, where we instantiate our meta-objects:

$$\text{Meta-substitution} \quad \Theta \quad ::= \quad \cdot \mid \Theta, \Psi.M/u \mid \Theta, \Psi/\psi$$

The application of a meta-substitution is fairly straight-forward and follows the same principles as simultaneous substitution. The restrictive aspect of it follows from the typing itself, where the context of the instantiation to align with the context of u's type. This follows work on the Contextual Modal Type Theory [28] and work by Cave [7]

$$\frac{}{\Delta \vdash \cdot : \cdot} \qquad \frac{\Delta \vdash \Psi : G \quad \Delta \vdash \Theta : \Delta'}{\Delta \vdash \Theta, \Psi/\psi : \Delta', \psi : G}$$

$$\frac{\Delta; \Psi \vdash M : A \quad \Delta \vdash \Theta : \Delta'}{\Delta \vdash \Theta, \Psi.M/u : \Delta', u : \{\Psi.A\}}$$

2.3 Linear Lambda Calculus

Let us now build the linear lambda calculus, where the syntax is slightly altered. In order to differentiate the intuitionistic setting from the linear one, linear implication is now denoted as $A_1 \multimap A_2$, linear lambda abstraction as $\widehat{\lambda}$, linear application as $M_1 \hat{} M_2$ and linear typing of a variable as $x \hat{:} A$

$$\begin{array}{llll} \text{Terms} & M & ::= & x \mid \widehat{\lambda} x \hat{:} A.M \mid M_1 \hat{} M_2 \\ \text{Types} & A & ::= & A_1 \multimap A_2 \mid \top \\ \text{Context} & \Gamma & ::= & \cdot \mid \Gamma, x \hat{:} A \end{array}$$

Variables are still represented by x, where they are bound by the linear lambda abstraction $\widehat{\lambda} x : A.M$. The basic intuition of linearity is that, once a variable is bound linearly, it must be used exactly once. This implies that it must be used at least once (no weakening),

and cannot be used multiple times (no contraction). This will be enforced through the typing rules, using a join operation $\bowtie$, described later.

Next, we have the linear application $M_1 \hat{} M_2$, where M_2 is linearly applied to M_1. This means that M_1 is assumed to be a linear function binding some variable x(which must appear unbounded in M_1 exactly once), while M_2 is an instantiation of x.

Next, there are types. The type $A_1 \multimap A_2$ represents the type of a linear lambda expression $\widehat{\lambda} x \hat{:} A_1 . M$, where A_1 is the input type, or type of variable x, while A_2 is the output type, or expected type of M after instantiating x. Once again, x is expected to appear exactly once.

$$\frac{}{x \hat{:} A_1 \vdash x \hat{:} A_1} \quad \frac{\Gamma, x \hat{:} A_1 \vdash M : A_2}{\Gamma \vdash \widehat{\lambda} x \hat{:} A_1 . M : A_1 \multimap A_2}$$

$$\frac{\Gamma_1 \vdash M : A_1 \multimap A_2 \quad \Gamma_2 \vdash M_2 : A_2 \quad \Gamma = \Gamma_1 \bowtie \Gamma_2}{\Gamma \vdash M_1 \hat{} M_2 : A_2}$$

$$\frac{\Gamma_1, x \hat{:}_2 A_2, x \hat{:}_1 A_1, \Gamma_2 \vdash M : A}{\Gamma_1, x \hat{:}_1 A_1, x \hat{:}_2 A_2, \Gamma_2 \vdash M : A}$$

We can notice that this is similar in nature to the simply typed lambda calculus, with the big distinction residing in the handling of contexts. There are two main differences in our typing rules. First, as a direct consequence of linearity, application $M_1 \hat{} M_2$ must respect that any variable x appears in either M_1 or M_2 (cases where it appears in neither or in both would not be linear). This is because linear variables must be used exactly once. For this reason, the context is split using the join operation $\Gamma = \Gamma_1 \bowtie \Gamma_2$. Second, the axiom case also differs: Because we are splitting the context to represent where we use variables, the axiom must use a context containing exactly the desired variable and none other. The

last rule presented is the exchange rule (a substructural rule). This is made explicit since contraction and weakening are not allowed in this calculus.

Let us now look at the join operation:

$$\frac{}{\cdot = \cdot \bowtie \cdot} \qquad \frac{\Gamma = \Gamma_1 \bowtie \Gamma_2}{\Gamma, x\hat{:}A = \Gamma_1, x\hat{:}A \bowtie \Gamma_2} \qquad \frac{\Gamma = \Gamma_1 \bowtie \Gamma_2}{\Gamma, x\hat{:}A = \Gamma_1 \bowtie \Gamma_2, x\hat{:}A}$$

In short, $\Gamma = \Gamma_1 \bowtie \Gamma_2$ forces each variable in Γ to go in either Γ_1 or Γ_2.

This calculus also comes with operational semantics, since we are not dealing with the normal form. The operational semantics for this language are the same as for the standard lambda calculus, the only difference being that substitution would replace each variable exactly once. Note that the substructural rules of weakening and contracting are not allowed, while the exchange rule is still used.

$$\frac{}{(\widehat{\lambda}x\hat{:}A.M_1)\hat{}M_2 \to M_1\{M_2/x\}}$$

$$\frac{M_1 \to M_1'}{M_1\hat{}M_2 \to M_1'\hat{}M_2} \qquad \frac{M_2 \to M_2'}{M_1\hat{}M_2 \to M_1\hat{}M_2'}$$

Before moving on to the next section, let us modify slightly our contexts. An alternative to the current formulation would be to keep every variable, but mark them as unavailable when they have been "used". We shall mark unavailable variables in our context as $x\check{:}T$. Thus, our context formulation becomes as follows:

$$\text{Context} \quad \Gamma \quad ::= \quad \cdot \mid \Gamma, x\hat{:}A \mid \Gamma, x\check{:}A$$

This formulation follows the work of Schack-Nielsen [35]. Now, we obviously need to modify our axiom and also notion of join. Let us first define a new operation over contexts, $\mathsf{unr}(\Phi)$, which is valid iff all variables in Φ are unavailable.

$$\frac{}{\mathsf{unr}(\cdot)} \qquad \frac{\mathsf{unr}(\Gamma)}{\mathsf{unr}(\Gamma, x\check{:}T)}$$

Now, we are ready to see the new variants of the axiom rule and the join:

$$\frac{\mathsf{unr}(\Gamma)}{\Gamma, x\hat{:}A \vdash x : A}$$

$$\frac{}{\cdot = \cdot \bowtie \cdot} \qquad \frac{\Gamma = \Gamma_1 \bowtie \Gamma_2}{\Gamma, x\check{:}A = \Gamma_1, x\check{:}A \bowtie \Gamma_2, x\check{:}A}$$

$$\frac{\Gamma = \Gamma_1 \bowtie \Gamma_2}{\Gamma, x\hat{:}A = \Gamma_1, x\hat{:}A \bowtie \Gamma_2, x\check{:}A} \qquad \frac{\Gamma = \Gamma_1 \bowtie \Gamma_2}{\Gamma, x\hat{:}A = \Gamma_1, x\check{:}A \bowtie \Gamma_2, x\hat{:}A}$$

2.4 Tree-Structure of Contexts

Let us now look at context variables in the setting of linear logic. One thing we must notice is that, through the application rule, a context variable is expected to be split. However, one important note is that, none of the explicit variables in a context can appear in a context variable. Thus, when we have the split $\psi, \Gamma = \psi_1, \Gamma_1 \bowtie \psi_2, \Gamma_2$, we know that $\psi = \psi_1 \bowtie \psi_2$ and $\Gamma = \Gamma_1 \bowtie \Gamma_2$. This thus allows us to handle context variables separately.

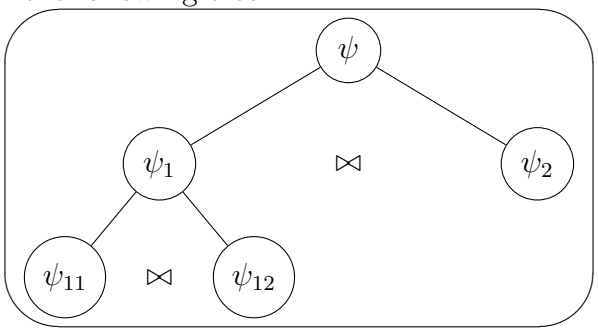
This, in turn, signifies that when we extend a context with a variable through a lambda binding, this does not affect the context variable at all. Based on this observation, we can look at the tree-structure of the typing tree and notice that, whenever the context is extended, the context variable is unaffected, while when we split a context, the context

variables are split. This means that the context variables can form a tree where each parent is the join of its children. Finally, the identity substitution represents the leaves of the tree.

For our purposes, let us look at a brief example:

$$\dfrac{\dfrac{\dfrac{\dfrac{\Delta, \psi_{11}, \Gamma'_{11} \vdash id_{\psi_{11}} : \psi_{11}}{\vdots}}{\Delta; \psi_{11}, \Gamma_{11} \vdash u_1[id_{\psi_{11}}] : A_1 \multimap B_1} \quad \dfrac{\dfrac{\Delta, \psi_{12}, \Gamma'_{12} \vdash id_{\psi_{12}} : \psi_{12}}{\vdots}}{\Delta; \psi_{12}, \Gamma_{12} \vdash u_2[id_{\psi_{12}}] : A_1}}{\vdots}}{\Delta; \psi_1, \Gamma_1 \vdash u_1[id_{\psi_{11}}]\hat{}u_2[id_{\psi_{12}}] : A \multimap B} \quad \dfrac{\dfrac{\Delta, \psi_2, \Gamma'_2 \vdash id_{\psi_2} : \psi_2}{\vdots}}{\Delta; \psi_2, \Gamma'_2 \vdash u_3[id_{\psi_2}] : B_2}}{\Delta; \psi, \Gamma \vdash (u_1[id_{\psi_{11}}]\hat{}u_2[id_{\psi_{12}}])\hat{}u_3[id_{\psi_2}] : B}$$

This would result in the following tree



Now that we know that the context variables respect a tree structure, the question becomes how to exploit it. While we might expect to simply need to specify the structure itself, one further property makes this impractical. Since we want to emulate the join operation, we also want to maintain its properties. Two important properties are associativity and commutativity.

In other words, if we have $\psi = \psi_1 \bowtie \psi_2$, we also want $\psi = \psi_2 \bowtie \psi_1$. This, in itself is manageable; However, some issues arise when dealing with associativity. In particular, we would need the following to hold: $\psi = \psi_1 \bowtie \psi_2$ and $\psi_1 = \psi_{11} \bowtie \psi_{12}$ implies that

$\psi = \psi_{11} \bowtie \psi'$ and $\psi' = \psi_{12} \bowtie \psi_2$. The main problem here is that we need a fresh context variable with a fresh name.

Our saving grace, however, is to notice that if we only look at the leaves themselves, associativity and commutativity can easily be maintained. Thus, the basic idea will be to describe a context variable by its set of leaves. This new notion is weaved directly in the name of the context variable, making this a nominal system, similarly to DeBruijn indices with its ordinary bound variables, where the name represents a location in the context. Moreover, we can note a similarity with the notion of used and unused variables, especially when using binary numbers to describe the system. We could use a binary number with n bits, where n is the number of leaves, and positive bits represent the leaves it contains, or uses, while the neutral bits represent the unused leaves.

CHAPTER 3
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 (presented in Chapter 4) and then write recursive programs that analyze the syntactic structure of these objects by pattern matching. The goal of this is to both give a flavour of LINCX before presenting it formally, and to highlight the role that contexts and context joins play.

3.1 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 the 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 close. 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.

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. Since the recursive type uses the context explicitly, it resides on the computational level, rather than at the LINCX level.

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

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

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

Figure 3–1: Translation of Linear ML-Expressions to a Linear Core Language

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. 3–1). 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. 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[1] . 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,x̂ml ⊢ ml]`, since we interpret

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

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.

While the variable case seems straightforward, it is important to note that this is because we are working directly with the context. In other systems, the variable case would require extra steps to make the translation possible.

Next, the most interesting cases are the third and fourth, as we must split the context. When we analyze for example $[\gamma_{(1\rhd\lhd 2)} \vdash \texttt{app ~ M ~N}]$, then `M` has some type $[\gamma_1 \vdash \texttt{ml}]$ and `N` has some type $[\gamma_2 \vdash \texttt{ml}]$ where $\gamma_{(1\rhd\lhd 2)} = \gamma_1 \bowtie \gamma_2$. We specify these type annotations and context constraints explicitly. We overload the $\bowtie$ here. 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 we explicitly state $\delta_{1\rhd\lhd 2} = \delta_1 \bowtie \delta_2$. We proceed similarly to translate recursively every let-expression into a function application.

Type checking will verify that a given object is well-typed modulo context joins. This is non-trivial. Consider for example $[\delta_{(1\rhd\lhd 2)} \vdash \texttt{lapp ~ (llam ~(}\widehat{\lambda}\texttt{x. N')) ~ M']}$ where $\delta_{(1\rhd\lhd 2)} = \delta_1 \bowtie \delta_2$. We want our underlying type theory to reason about context constraints modulo associativity and commutativity.

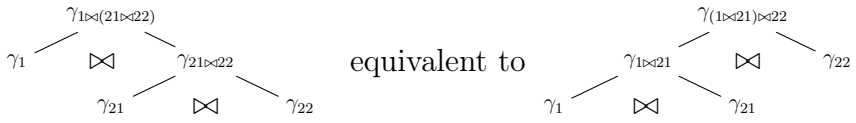


Figure 3–2: Context Joins

As the astute reader will have noticed, we only allow one context variable in every context, i.e. writing $[\delta_1, \delta_2 \vdash$ `lapp ^ (llam ^ (`$\widehat{\lambda}$`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 deliberately have the same name, but differ in their subscripts. The context variables γ_1 and γ_2 are the *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, consider the following deeply nested pattern: $[\gamma_{((11\bowtie 12)\bowtie 2)} \vdash$ `lapp ^ (lapp ^ (llam ^ (`$\widehat{\lambda}$`x. M)) ^ N') ^ K]` where `M : [`γ_{11}`, x:: ml` $\vdash$ `ml], N : [`$\gamma_{12} \vdash$ `ml], K : [`$\gamma_2 \vdash$ `ml]`. We again encode the splitting of γ in its subscript. Our underlying theory of context joins will treat $\gamma_{(11\bowtie(12\bowtie 2))}$ as equivalent to $\gamma_{((11\bowtie 12)\bowtie 2)}$ or $\gamma_{((12\bowtie 11)\bowtie 2)}$ as our equational theory on context joins 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 $\gamma_{21\bowtie 22}$).

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.

3.2 Example: CPS-translation

As a second example, we implement the translation of programs into continuation passing style following Danvy and Filinski [14]. Concretely, we follow closely the existing implementation of type-preserving CPS translation in BELUGA by Belanger et.al [2], but enforce that the continuations are used linearly, an idea from Berdine et.al [4]. Although context splits do not arise in this example, as we only have one linear variable in our context, namely the variable standing for the continuation, we include this example, 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 to relate the `source` context to the `value` context. Notice that we explicitly state that the type `s` of a source and target expression does not depend on γ or δ. It is closed. To distinguish between objects that depend on their surrounding context and objects that do not, we associate an

index and pattern variable with a substitution. As mentioned earlier, by default every index and pattern variable depends on its surrounding context and is associated with an identity substitution. If we want to state that a given variable is closed, we associate it with the empty substitution `[]`.

```
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[]] ;
```

We can now define the translation itself (see Fig. 3–3). 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. However, S itself does not depend on γ, since it is associated with the empty substitution `[]`. 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.

```
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]]];
```

Figure 3–3: 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, `k̂value _ → exp`, as `k` is well-typed in the context that only contains the declaration `k̂value _ → 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,`k̂value nat → exp`. Note that an application `k kz` is well-typed in δ,`k̂value nat → exp`, as `kz` is well-typed in δ, i.e. its unrestricted part.

Lambda Case. 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` λ`x.`$\widehat{\lambda}$`c.P`.

Application Case. 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.

CHAPTER 4
Theory

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 [10] and contextual LF [7].

4.1 Disclaimer

It is important to note that the insight into the tree-structure of the context-variables following the structure of the derivation trees was first suggested by the present author. Later, the present author also came up with the notion of a nominal description of context variables to describe the tree-structure, based on the notion that an intermediary node can be uniquely described as the union of its leaves. This was initially described using binary numbers, and was later refined by Agata Murawska into the current monoidal version. Most of the remaining work was done as a collaboration between Agata Murawska, Aina-Linn Georges and the present author.

Fig. 4–1 presents both contextual linear LF and its meta-language (see Sect. 4.7).

4.2 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

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	::=	$\cdot \mid \Theta, C/X$

Figure 4–1: Contextual Linear LF with First-Class Contexts

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 [11], 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 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, following [8]. 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.

4.3 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 [36], 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 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. 4.7, 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. 3–2). 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

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

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

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

Figure 4–2: Well-Formed Contexts

and 001 instead of 1, 21, and 22. Rules of constructing a well-formed context (Fig. 4–2) describe four possible initial cases of context construction. First is simply a context containing no context variable, $\cdot$. Next, a context containing a context variable taken directly from the meta-context, $\psi_i \in \Delta$. A composition rule 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. Here, ψ_ϵ is an abstraction to describe the intuitionistic variant ψ_i. Since each ψ_i must have the same unrestricted variables, then ψ_ϵ is unique. Moreover, since in general we do not want to use a variable that was undeclared, we enforce that ψ_ϵ is the unrestricted form of some declared context variable. Finally, the last case covers context extension. Here we only need to ensure that the type is well-formed.

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. Following [10] we introduce the function $\overline{\Psi}$ which is defined as follows:

$$
\begin{array}{lcl}
\boxed{\overline{\Psi}} & & \text{Intuitionistic part of } \Psi \\
\overline{\cdot} & = & \cdot \\
\overline{\psi_m} & = & \psi_\epsilon \\
\overline{\Psi, x{:}A} & = & \overline{\Psi}, x{:}A \\
\overline{\Psi, x\hat{:}A} & = & \overline{\Psi}, x\check{:}A \\
\overline{\Psi, x\check{:}A} & = & \overline{\Psi}, x\check{:}A
\end{array}
$$

Note that this function does not remove any linear 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.

The rules for joining contexts (see Fig. 4–3) follow [35], 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 $\psi_m = \psi_m \bowtie \psi_\epsilon$.

$\boxed{\Psi = \Psi_1 \bowtie \Psi_2}$ Context Ψ is a join of Ψ_1 and Ψ_2

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

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

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

Figure 4–3: Joining Contexts

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.

While the proofs of this lemma are omitted here, they have been encoded in the mechanization (see Chapter 5)

4.4 Typing for Terms and Substitutions

We now describe the bi-directional typing rules of LINCX terms (see Fig. 4–4). 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). While this operation is in itself non-deterministic, there is only one split that makes the components (for example H and S in $H \cdot S$) typecheck.

Typing rules presented in Fig. 4–4 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.

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

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

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

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

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

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

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

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

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

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

Figure 4–4: Typing Rules for Terms

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.

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

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

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

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

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

Figure 4–5: Typing Rules for Substitutions

Finally, we consider the typing rules for substitutions, presented in Fig. 4–5. We exercise care in making sure the range of the substitution 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 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 $\Phi, x\check{:}A$ 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 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. We quickly note, however, that this enforces A to be inhabited. 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. 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* $\Delta; \overline{\Psi} \vdash \sigma \Leftarrow \overline{\Phi}$.

4.5 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.

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

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 [8], 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:

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

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

For the sake of completeness, the full rules for hereditary single substitution can be found in the appendix A.1 with rules presented on Fig. A–1.

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 term (Fig. 4–6). Hereditary simultaneous substitution requires a lookup function. Notice that $(\sigma, M)_{\Psi, x{:}\hat{}A}(x) = \bot$, since we assume x to be unavailable in the domain of σ.

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

$$(\sigma, M)_{\Psi, x:A}(x) = M : A^-$$

$$(\sigma, M)_{\Psi, x\hat{:}A}(x) = M : A^-$$

$$(\sigma, M)_{\Psi, y:A}(x) = \sigma_\Psi(x) \qquad \text{where } y \neq x$$

$$(\sigma, M)_{\Psi, y\hat{:}A}(x) = \sigma_\Psi(x) \qquad \text{where } y \neq x$$

Unlike many previous formulations of contextual LF, we do not allow substitutions to be directly extended with variables. Instead, following [8], 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 the 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

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

$(\sigma, M)_{\Psi,x:A}(x) = M : A^-$

$(\sigma, M)_{\Psi,x\hat{:}A}(x) = M : A^-$

$(\sigma, M)_{\Psi,y:A}(x) = \sigma_\Psi(x)$ where $y \neq x$

$(\sigma, M)_{\Psi,y\check{:}A}(x) = \sigma_\Psi(x)$ where $y \neq x$

$\boxed{[\sigma]^{\widetilde{\Phi}}_{\Psi} M}$ 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 $[\sigma]^{\widetilde{\Phi}_2}_{\Psi_2} S = S'$

$\boxed{[\sigma]^{\widetilde{\Phi}}_{\Psi} S}$ Substitution of the variables of Ψ in a spine S

$[\sigma]^{\widetilde{\Phi}}_{\Psi}(\epsilon) = \epsilon$ where $\mathsf{unr}(\Psi)$ and $\mathsf{unr}(\widetilde{\Phi})$

$[\sigma]^{\widetilde{\Phi}}_{\Psi}(N\,;S) = N'\,;S'$ where $[\sigma]^{\widetilde{\Phi}}_{\Psi} N = N'$ and $[\sigma]^{\widetilde{\Phi}}_{\Psi} S = S'$

$[\sigma]^{\widetilde{\Phi}}_{\Psi}(N\,\hat{;}\,S) = N'\,\hat{;}\,S'$ where $\Psi = \Psi_1 \bowtie \Psi_2$ and $\widetilde{\Phi} = \widetilde{\Phi}_1 \bowtie \widetilde{\Phi}_2$ and $[\sigma]^{\widetilde{\Phi}_1}_{\Psi_1} N = N'$ and $[\sigma]^{\widetilde{\Phi}_2}_{\Psi_2} S = S'$

$\boxed{[\sigma]^{\widetilde{\Phi}}_{\Psi}\tau}$ Substitution of the variables of Ψ in a substitution τ

$[\sigma]^{\widetilde{\Phi}}_{\Psi}(\cdot) = \cdot$

$[\sigma]^{\widetilde{\Phi}}_{\Psi}(\mathsf{id}_\psi) = \mathsf{id}_\psi$

$[\sigma]^{\widetilde{\Phi}}_{\Psi}(\tau, M) = (\tau', M')$ 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 $[\sigma]^{\widetilde{\Phi}_2}_{\Psi_2} M = M'$

Figure 4–6: Simultaneous Substitution

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.

The correctness of substitution typing is described in (Theorem 3)

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

However, in order to prove it, we first need to generalize it (Theorem 4), and also prove some helper lemmas (Lemma 4 and 5). We quickly note that, as dependent types are always unrestricted, types cannot depend on linear variables. Moreover, linear assumptions must be used exactly once, no matter where it is. For these reason, given, for example, judgement $J = M \Leftarrow A$, we unfold substitution as follows: $[\sigma]^{\widetilde{\Gamma}}_{\Phi} J = [\sigma]^{\widetilde{\Gamma}}_{\Phi} M \Leftarrow [\sigma]^{\widetilde{\Gamma}}_{\Phi} A$. Note, however, that the judgement is more generic than simply type checking.

Lemma 3. *If* $\Phi, \Gamma = \Psi_1 \bowtie \Psi_2$*, then* $\exists \Phi_1, \Phi_2, \Gamma_1, \Gamma_2$ *s.t.* $\Psi_1 = \Phi_1, \Gamma_1$*,* $\Psi_2 = \Phi_2, \Gamma_2$*,* $\Phi = \Phi_1 \bowtie \Phi_2$ *and* $\Gamma = \Gamma_1 \bowtie \Gamma_2$.

Lemma 4. *Suppose* $\Psi = \Psi_1 \bowtie \Psi_2$ *and* $\Delta; \Phi \vdash \sigma \Leftarrow \Psi$*, then* $\Delta; \Phi_1 \vdash \sigma \Leftarrow \Psi_1$ *and* $\Delta; \Phi_2 \vdash \sigma \Leftarrow \Psi_2$ *for* $\Phi = \Phi_1 \bowtie \Phi_2$.

Lemma 5. $[\sigma]^{\widetilde{\Gamma_1}, \widetilde{\Gamma_2}}_{\Psi} J = [\sigma]^{\widetilde{\Gamma_1}, \breve{x}, \widetilde{\Gamma_2}}_{\Psi} J$

Proof. By induction on the simultaneous substitution □

Theorem 4 (Simultaneous substitution property)**.** *If* $\Delta; \Psi, \Gamma \vdash J$ *and* $\Delta; \Psi' \vdash \sigma \Leftarrow \Psi$ *then* $\Delta; \Psi', [\sigma]^{\cdot}_{\Psi} \Gamma \vdash [\sigma]^{\widetilde{\Gamma}}_{\Psi}(J)$.

Proof. Proof by induction on the typing derivation $\mathcal{D} :: \Delta; \Psi, \Gamma \vdash J$. We show some representative cases.

Case. $\mathcal{D} = \dfrac{\Delta; \Psi, \Gamma, x\hat{:}A \vdash M \Leftarrow B}{\Delta; \Psi, \Gamma \vdash \widehat{\lambda} x.M \Leftarrow A \multimap B}$

$\Delta; \Psi, [\sigma]^{\cdot}_{\overline{\Psi}}(\Gamma, x\hat{:}A) \vdash [\sigma]^{\widetilde{\Gamma},\hat{x}}_{\Psi} M \Leftarrow [\sigma]^{\widetilde{\overline{\Gamma}},\check{x}}_{\overline{\Psi}} B$ by IH

$\Delta; \Psi, [\sigma]^{\cdot}_{\overline{\Psi}}\Gamma, x\hat{:}[\sigma]^{\Gamma}_{\overline{\Psi}}A) \vdash [\sigma]^{\widetilde{\Gamma},\hat{x}}_{\Psi} M \Leftarrow [\sigma]^{\widetilde{\overline{\Gamma}},\check{x}}_{\overline{\Psi}} B$ by definition of substitution

$\Delta; \Psi, [\sigma]^{\cdot}_{\overline{\Psi}}\Gamma[\sigma]^{\cdot}_{\overline{\Psi}}A) \vdash \widehat{\lambda} x.[\sigma]^{\widetilde{\Gamma}}_{\Psi} M \Leftarrow [\sigma]^{\cdot}_{\overline{\Psi}}A \multimap [\sigma]^{\widetilde{\overline{\Gamma}},\check{x}}_{\overline{\Psi}} B$ by typing rule

$\Delta; \Psi, [\sigma]^{\cdot}_{\overline{\Psi}}\Gamma[\sigma]^{\cdot}_{\overline{\Psi}}A) \vdash \widehat{\lambda} x.[\sigma]^{\widetilde{\Gamma}}_{\Psi} M \Leftarrow [\sigma]^{\cdot}_{\overline{\Psi}}A \multimap [\sigma]^{\widetilde{\overline{\Gamma}}}_{\overline{\Psi}} B$ By Lemma 5

$\Delta; \Psi, [\sigma]^{\cdot}_{\overline{\Psi}}\Gamma \vdash [\sigma]^{\widetilde{\Gamma}}_{\Psi}\, \widehat{\lambda}\, x.M \Leftarrow [\sigma]^{\widetilde{\Gamma}}_{\overline{\Psi}}(A \multimap B)$ by definition of substitution

Case. $\mathcal{D} = \dfrac{\Delta; \Psi_1 \vdash M \Leftarrow A \qquad \Delta; \Psi_2 \vdash S > B \Rightarrow P \qquad \Phi, \Gamma = \Psi_1 \bowtie \Psi_2}{\Delta; \Phi, \Gamma \vdash M\,\hat{;}\,S > A \multimap B \Rightarrow P}$

$\Psi_1 = \Phi_1, \Gamma_1$

$\Psi_2 = \Phi_2, \Gamma_2$

$\Phi = \Phi_1 \bowtie \Phi_2$

$\Gamma = \Gamma_1 \bowtie \Gamma_2$ by Lemma 3

$\Delta; \Psi_1' \vdash \sigma \Leftarrow \Psi_1$

$\Delta; \Psi_2' \vdash \sigma \Leftarrow \Psi_2$

$\Psi' = \Psi_1' \bowtie \Psi_2'$ by Lemma 4

$\Delta; \Phi_1, [\sigma]^{\cdot}_{\overline{\Phi}_1}\Gamma_1 \vdash [\sigma]^{\widetilde{\Gamma}_1}_{\Phi_1} M \Leftarrow [\sigma]^{\widetilde{\overline{\Gamma}_1}}_{\overline{\Phi}_1} A$

$\Delta; \Phi_2, [\sigma]^{\cdot}_{\overline{\Phi}_2}\Gamma_2 \vdash [\sigma]^{\widetilde{\Gamma}_2}_{\Phi_2} S > [\sigma]^{\widetilde{\overline{\Gamma}_2}}_{\overline{\Phi}_2} B \Rightarrow [\sigma]^{\widetilde{\overline{\Gamma}_2}}_{\overline{\Phi}_2} P$ by IH

$\Phi, [\sigma]_{\overline{\Phi}}\Gamma = \Phi_1, [\sigma]_{\overline{\Phi}_1}\Gamma_1 \bowtie \Phi_2, [\sigma]_{\overline{\Phi}_2}\Gamma_2$

$\Delta; \Phi, [\sigma]_{\overline{\Phi}}\Gamma \vdash [\sigma]^{\widetilde{\Gamma_1}}_{\Phi_1} M\,\hat{;}\,[\sigma]^{\widetilde{\Gamma_2}}_{\Phi_2} S > [\sigma]^{\widetilde{\overline{\Gamma}_1}}_{\overline{\Phi}_1} A \multimap [\sigma]^{\widetilde{\overline{\Gamma}_2}}_{\overline{\Phi}_2} B \Rightarrow [\sigma]^{\widetilde{\overline{\Gamma}_2}}_{\overline{\Phi}_2} P$ By typing rule

$\Delta; \Phi, [\sigma]_{\overline{\Phi}}\Gamma \vdash [\sigma]^{\widetilde{\Gamma_1}}_{\Phi_1} M\,\hat{;}\,[\sigma]^{\widetilde{\Gamma_2}}_{\Phi_2} S > [\sigma]^{\widetilde{\overline{\Gamma}}}_{\overline{\Phi}} A \multimap [\sigma]^{\widetilde{\overline{\Gamma}}}_{\overline{\Phi}} B \Rightarrow [\sigma]^{\widetilde{\overline{\Gamma}}}_{\overline{\Phi}} P$ By Lemma 1

$\Delta; \Phi, [\sigma]_{\overline{\Phi}}\Gamma \vdash [\sigma]^{\widetilde{\Gamma_1}}_{\Phi_1} M \,\hat{;}\, S > [\sigma]^{\widetilde{\overline{\Gamma}}}_{\overline{\Phi}} A \multimap [\sigma]^{\widetilde{\overline{\Gamma}}}_{\overline{\Phi}} B \Rightarrow [\sigma]^{\widetilde{\overline{\Gamma}}}_{\overline{\Phi}} P$ By substitution

□

4.6 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 list all possible contexts Ψ' well-formed under Δ and such that $\overline{\Psi} = \overline{\Psi'}$.

Theorem 5 (Decidability of type checking)**.** *Type checking is decidable.*

4.7 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 [7].

Fig. 4–1 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).

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

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

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

Figure 4–7: Well-Formed Meta-Contexts

$\boxed{\widetilde{\Psi}}$ $\quad$ Name and status of variables from Ψ

$$\begin{array}{lcl}
\widetilde{\cdot} & = & \cdot \\
\widetilde{\psi_m} & = & \psi_m \\
\widetilde{\Psi, x{:}A} & = & \widetilde{\Psi}, x \\
\widetilde{\Psi, x\hat{:}A} & = & \widetilde{\Psi}, \widehat{x} \\
\widetilde{\Psi, x\check{:}A} & = & \widetilde{\Psi}, \check{x}
\end{array}$$

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.

$\boxed{\Psi \perp_\psi \Theta}$ Context Ψ is linearly disjoint from the range of Θ for ψ

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

$\boxed{\Delta \vdash \Theta \Leftarrow \Delta'}$ Θ has domain Δ' and range Δ

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

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

Figure 4–8: Typing Rules for Meta-Substitutions

We now give rules for a well-formed meta-context Δ (see Fig. 4–7). It is defined on the structure of Δ and is mostly straightforward. 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: $\forall j.\psi_j \in \mathsf{dom}(\Delta) \rightarrow \psi_j : G \in \Delta$. 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. Thus, we move the presentation of these rules to the appendix (see Fig. A–2 and Fig. A–3).

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. 4–8) 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 (see Fig. A–4 in the appendix). What is slightly more involved is the variable lookup function.

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

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

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

CHAPTER 5
Mechanization

While describing a system and proving properties about it manually is a good starting point, the confidence we can place in the system remains limited. In order to increase our confidence in a system, a key step is to formalize and mechanize it inside of a programming and reasoning framework. In this case, we have mechanized part of our system, Lincx, in Beluga, a programming and reasoning framework based on a two-level approach: a computational level is built on top of the contextual modal logical framework[31].

In particular, due to the novelty of Lincx, we have worked on the formalization of the contexts and context joins, and proved some properties about them. Since this is at the heart of Lincx, our mechanization allows us to have good confidence in the viability of Lincx. Similarly, we have formalized various parts of the language. In particular, we have fomalized contexts and their equational theory 5.4, typing rules (Section 5.5), substitution (Sections 5.6, 5.7). Moreover, some important lemmas were proven (Sections 5.4, 5.8). We quickly note that, due to a lack of time, meta-variables, parameter variables, meta-contexts and meta-substitutions were not implemented. Thus, the subset we have formalized consists of linear LF with hereditary and simultaneous substitution and context variables.

In this section, we will be presenting the work this implementation. Note that in many figures, we present both the paper definition (as seen in Chapter 4) and the beluga encoding. Both formalisms are put in the same place for ease to the reader.

5.1 Disclaimer

The work presented in this section has been done in collaboration with Aina-Linn Georges. The current author built the initial set-up for representing syntax, typing rules and substitutions. The main aspects of proving and formalizing properties were done in collaboration with A.L. Georges.

5.2 Related Mechanization

In this work, we have chosen to handle contexts explicitly, although other approaches to mechanizing linear logic in LF exist. In particular, there is Crary's mechanization of substructural logics in Twelf [13], which hinges on putting the principles of linear logic on their head: instead of considering contexts and splitting them, he adopts the use of an extra derivation which ensures each variable is used exactly once. While this is an interesting approach which conserves the many benefits of HOAS, such as having the substitution lemma for free, this was impractical for our purposes, in particular since we want to consider context variables. Instead, our approach more closely ressembles the work of Martens where she implements LF in LF [23]. While contexts are not used explicitly in the latter, there are significant parallels to draw between both. In particular, the explicit handling of hereditary substitution is used in a similar manner.

5.3 Syntax

While most of the syntax is self-explanatory, being written following standard LF conventions, it is worth taking a close look at some of the definitions. First of all, since we are working in a linear setting within intuitionistic LF, variables are handled explicitly. For this purpose, as described by the `var` type (Fig.5–1), we explicitly define a variable type which

```
LF var : type =
;
```

Figure 5–1: Variable Encoding

we leave uninhabited (We briefly note that we handle explicitly the constants, which come with constant types and an inductive signature, in a similar manner).

Since we have defined terms (head, spine, atomic terms and canonical terms) mutually, we must also use mutually defined LF-types (respectively `head`, `spine`, `atom` and `canon` in Fig.5–2), which is done in Beluga using the keyword "and". Moreover, we note that Beluga uses higher-order abstract syntax trees (HOAS), which allows us to abstractly handle our variable bindings. The structure of HOAS allows for many benefits related to variables handling, and allows for substitution and substitution properties to be automatically handled for the user. Thus, while we explicitly handle variables, we can still make use of the HOAS provided (In our case, weak HOAS). For this reason, we define, lambda terms using the standard LF-notation, where the body depends on a variable through HOAS. This way, while we explicitly carry our explicit context, we can still capture and use assumptions in the context provided at the computational level.

Let us now turn our attention to explicit contexts. As mentioned previously, we have a nominal form for context variables, describing their place in the derivation tree. This is described by a commutative monoid with a joining operation. In the case of our implementation, we use the special case of binary numbers (`bin` in Fig.5–3). Thus, to describe this nominal form, we define context variables as having an argument, namely a binary number (`cvar` in Fig.5–4).

```
LF head : type =
   | hd_cst : cst → head
   | hd_var : var → head

and spine : type =
   | sp_empty : spine
   | sp_unr : canon → spine → spine
   | sp_lin : canon → spine → spine

and atom : type =
   | atom_base : head → spine → atom

and canon : type =
   | c_atom : atom → canon
   | lam : (var → canon) → canon
   | llam : (var → canon) → canon
;
```

Term Syntax

Heads	H	::=	$x \mid c$
Spines	S	::=	$\epsilon \mid M\,;S \mid M\,\hat{;}\,S$
Atomic Terms	R	::=	$H \cdot S$
Canonical Terms	M, N	::=	$R \mid \lambda x.M \mid \widehat{\lambda} x.M$

Figure 5–2: Syntax Encoding

```
LF bit : type =
   | zero : bit
   | one : bit
;

LF bin : type =
   | nil : bin
   | cons : bit → bin → bin
;
```

Figure 5–3: Binary Numbers Encoding

```
LF flag : type =
   | lin : flag
   | unr : flag
   | unav : flag
;
LF vdecl : type =
   | decl : var → tp → flag → vdecl
;
LF ctx_var : type =
   | cvar : bin → ctx_var
;
LF ctx : type =
   | c_empty : ctx
   | varctx : ctx_var → ctx
   | snoc : ctx → vdecl → ctx
;
```

Context syntax

Variable Declarations	D	::=	$x{:}A \mid x\hat{:}A \mid x\check{:}A$
Contexts	Ψ, Φ	::=	$\cdot \mid \psi_m \mid \Psi, D$

Figure 5–4: Context Encoding

Let us now move on to general contexts (defined as type `ctx` in Fig.5–4). As expected, contexts are defined inductively. The two base cases are either an empty context or a context variables, and we inductively add variable declarations, defined as a tuple of a variable, a flag (linear, unrestricted or unavailable) and a type.

Finally, while single substitution can be handled using the HOAS, linearity of variables forces us to use explicit substitutions. The simultaneous substitution is defined inductively (defined as `sim_subst` in Fig.5–5): the base cases are the empty and identity substitution, and we inductively add canonical terms.

```
LF sim_subst : type =
   | s_empty : sim_subst
   | s_id : sim_subst
   | s_snoc : sim_subst → canon → sim_subst
;
```

Substitution Syntax

Substitutions $\sigma, \tau \quad ::= \quad \cdot \mid \mathsf{id}_\psi \mid \sigma, M$

Figure 5–5: Simultaneous Substitution Syntactical Encoding

5.4 Contexts and Context Joins

First, let us describe the binary representation of context variables. As defined in the previous section, we defined the commutative monoid as the set of binary numbers(Fig.5–3). To be more precise, this is defined as a list of bits. We note that, for simplicity, we are working with the assumption that all context variables are using binary numbers of the same length (except for the empty one), which can be enforced since the number of leaves is constant and is only affected by meta-substitution. Moreover, the number of leaves is constant on either side of a meta-substitution. Under this notation, leaf context variables are denoted with binary numbers where a single bit is 1, and such that no two leaves share the same positive bit. Meanwhile, our "empty set" is represented as the empty list "nil". Finally, an intermediate node will have the bits of their leaves be 1, while all other bits are 0.

The joining operation(`bin_join` in Fig.5–6), in turn, is simply defined as a binary OR, where we ensure that no bit is simultaneously 1 on both sides. This way, the leaves of the "join-node" corresponds to the leaves of both joined nodes, and we ensure that no node

```
LF bin_join : bin → bin → bin → type =
   | bin_join_nil_l : bin_join nil M M
   | bin_join_nil_r : bin_join M nil M
   | bin_join_l : bin_join M N K → bin_join (cons one M) (cons zero N) (cons one K)
   | bin_join_r : bin_join M N K → bin_join (cons zero M) (cons one N) (cons one K)
   | bin_join_zero : bin_join M N K → bin_join (cons zero M) (cons zero N) (cons zero K)
;
```

Figure 5–6: Binary Join Encoding

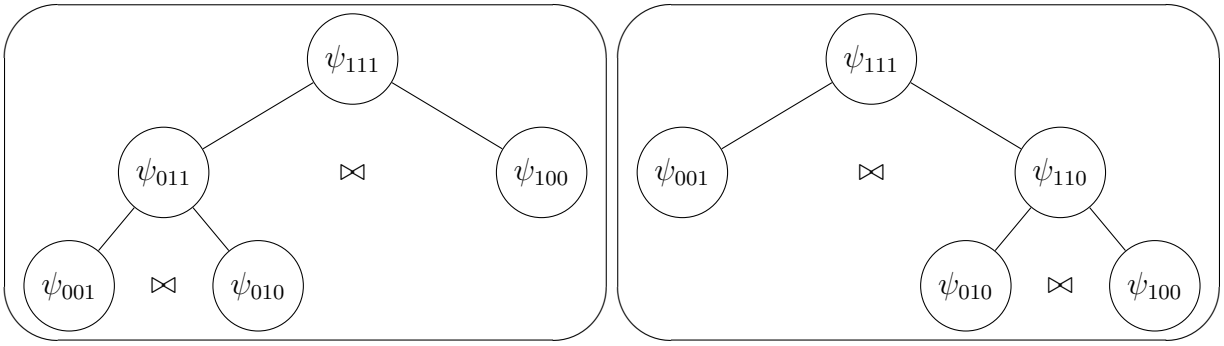


Figure 5–7: Binary Representation of Context Variables

appears more than once, ensuring we respect linearity, or more specifically, the commutative monoid and the properties of its join operation. This operation is simplified by the assumption that the binary numbers have the same length.

In the LF-definitions themselves, nothing too interesting happens: the binary join is defined as expected considering this special binary OR, while the context join operation (`joining` in Fig.5–8) is derived directly from the rules.

Before going further into the implementation of contexts, let us look at an example of how context variables relate to each other using this binary definition. Suppose we have three context variables, ψ_{001}, ψ_{010} and ψ_{100}. In this case, we would have that $\psi_{011} = \psi_{010} \bowtie \psi_{001} = \psi_{001} \bowtie \psi_{010}$. Similarly, we can obtain our associativity directly. This will be shown in the two trees presented in Fig.5–7, where a parent node is the join of its two children.

```
LF join : type =
   | cjoin : ctx → ctx → ctx → join
;

LF joining : join → type =
   | joining_base : bin_join M N K
→ joining (cjoin (varctx (cvar K)) (varctx (cvar M)) (varctx (cvar N)))
   | joining_empty : joining (cjoin c_empty c_empty c_empty)
   | joining_lin_l : joining (cjoin Ψ Ψ1 Ψ2)
→ joining (cjoin (snoc Ψ (decl x A lin)) (snoc Ψ1 (decl x A lin)) (snoc Ψ2 (decl x A unav)))
   | joining_lin_r : joining (cjoin Ψ Ψ1 Ψ2)
→ joining (cjoin (snoc Ψ (decl x A lin)) (snoc Ψ1 (decl x A unav)) (snoc Ψ2 (decl x A lin)))
   | joining_unr       : joining (cjoin Ψ Ψ1 Ψ2)
→ joining (cjoin (snoc Ψ (decl x A unr)) (snoc Ψ1 (decl x A unr)) (snoc Ψ2 (decl x A unr)))
   | joining_unav      : joining (cjoin Ψ Ψ1 Ψ2)
→ joining (cjoin (snoc Ψ (decl x A unav)) (snoc Ψ1 (decl x A unav)) (snoc Ψ2 (decl x A unav)))
;
```

$\boxed{\Psi = \Psi_1 \bowtie \Psi_2}$ Context Ψ is a join of Ψ_1 and Ψ_2

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

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

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

Figure 5–8: Context Join Encoding

```
LF ctx_mer : ctx → ctx → ctx → type =
   | m_empty : ctx_mer Ψ c_empty Ψ
   | m_cons : ctx_mer Φ Γ Ψ → ctx_mer Φ (snoc Γ V) (snoc Ψ V)
;
```

Figure 5–9: Context Merge Encoding

A more noteworthy aspect of our LF-definitions is the fact that some functions must be made explicit in the mechanization, since we are working with explicit contexts. First, in order to talk about greater contexts Φ, Γ, we need to create a special "merge" operator (`ctx_mer` in Fig.5–9) which takes as arguments Φ and Γ and the merged context Φ, Γ, which we denote as Ψ. It is important to note that, while this represents a function, it is formulated as a relation, where the return argument corresponds to the last element, rather than an actual function. This means that, if we want to state that this "function" has a unique output, it needs to be proven explicitly.

Similarly, we need to explicitly define what the unrestricted version of a context is and whether a context is unrestricted (respectively `unrest_ctx` and `is_unr` in Fig.5–10), which once again is represented as a relation rather than a function. We once again remark that the "empty set" is represented by the nil list, which means that the unrestricted version of a context variable is the context variable with the nil binary.

While `unrest_ctx` is used to compute the unrestricted form of a context, `is_ctx` represents a unary relation which tests whether a given context is unrestricted.

An important aspect to consider in order to define our properties, especially in the case of contexts and context joins, is the notion of equality. More precisely, when are two contexts equal? This is done using the usual syntactic equality approach (`eq` in Fig.5–11). Thus, our

```
LF unrest_ctx : ctx → ctx → type =
   | unrest_base_var   : unrest_ctx (varctx Psi) (varctx (cvar nil))
   | unrest_base_e   : unrest_ctx (c_empty) (c_empty)
   | unrest_l        : unrest_ctx Ψ1 Ψ̄
→ unrest_ctx (snoc Ψ1 (decl x A lin)) (snoc Ψ̄ (decl x A unav))
   | unrest_unr      : unrest_ctx Ψ1 Ψ̄
→ unrest_ctx (snoc Ψ (decl x A unr)) (snoc Ψ̄ (decl x A unr))
   | unrest_unav     : unrest_ctx Ψ Ψ̄
→ unrest_ctx (snoc Ψ (decl x A unav)) (snoc Ψ̄ (decl x A unav))
;

LF is_unr : ctx → type =
   | is_unr_b : is_unr (varctx (cvar nil))
   | is_unr_e : is_unr c_empty
   | is_unr_cons : is_unr Ψ → is_unr (snoc Ψ (decl x A unr))
   | is_unr_unav_cons : is_unr Ψ → is_unr (snoc Ψ (decl x A unav))
;
```

$$
\begin{array}{lcl}
\boxed{\overline{\Psi}} & & \text{Intuitionistic part of } \Psi \\
\overline{\cdot} & = & \cdot \\
\overline{\psi_m} & = & \psi_\epsilon \\
\overline{\Psi, x{:}A} & = & \overline{\Psi}, x{:}A \\
\overline{\Psi, x\hat{:}A} & = & \overline{\Psi}, x\check{:}A \\
\overline{\Psi, x\check{:}A} & = & \overline{\Psi}, x\check{:}A
\end{array}
$$

Figure 5–10: Unrestricted Contexts

```
LF eq : bin → bin → type =
   | bin_refl : eq K K
;
```

Figure 5–11: Context Equality

context equality will have a type with two arguments, and be inhabited by a single term where we force both arguments to be identical: reflexivity.

Due to our low-level approach to handling contexts, describing them explicitly, multiple properties must also be handled explicitly. In particular, we must prove that: given two contexts being equal, they can be used "interchangeably"; the unrestricted form of a context exists; etc., which in paper proofs we might use directly. Similarly, we need to prove that relations on contexts behave as expected. For instance, the unrestricted form of a context is unrestricted; the unrestricted form of an unrestricted context is unchanged; the merge of two unrestricted contexts is also unrestricted; etc., which should hold in any linear system, and thus shouldn't need to be handled explicitly when a proper linear tool, such as Lincx could prove to be.

Let us quickly look at the existence of the unrestricted version of a context (`unrest_exist` in Fig.5–12). It is worth noting that, since we are proving existence, we need to define a new LF-type (`int_unrest_ex` in Fig.5–12) to formalize this property. This new type, `int_unrest_ex`, takes an argument, being the context for which we want existence, and has a unique term. This unique term, `int_unr`, has in its argument the unrestricted version, which shows that an instance actually exists.

Let us look at Fig.5–12 in more depth and the proof `unrest_exist`. First, we take a look at the type, which refers to the statement we are trying to prove: `(Γ:sch) {Ψ : [Γ ⊢ ctx]} [Γ ⊢ int_unrest_ex Ψ]`. First of all, we (implicitly) declare a (LF) context through the declaration

```
LF int_unrest_ex : ctx → type =
   | int_unr : unrest_ctx Psi Psi_unr → int_unrest_ex Psi
;
rec ue_helper1 : {ψ : [⊢ ctx_var]} [⊢ unrest_ctx (varctx ψ) (varctx (cvar nil))] =
    mlam ψ ⇒
    [⊢ unrest_base_var]
;
rec ue_helper2 : {ψ : [⊢ ctx_var]} [⊢ unrest_ctx (varctx (cvar nil)) (varctx (cvar nil))] =
    mlam ψ ⇒
    [⊢ unrest_base_var]
;

schema sch = var;

rec unrest_exist : (g:sch) {Ψ : [Γ ⊢ ctx]} [Γ ⊢ int_unrest_ex Ψ] =
    mlam Ψ ⇒
    case [_ ⊢ Ψ] of
    | [Γ ⊢ c_empty] ⇒ [Γ ⊢ int_unr unrest_base_e]
    | [Γ ⊢ varctx ψ[]] ⇒
      let [⊢ D] = ue_helper1 [⊢ ψ] in
      [Γ ⊢ int_unr D[]]
    | [Γ ⊢ varctx (cvar nil)] ⇒
      [Γ ⊢ int_unr unrest_base_var]
    | [Γ ⊢ snoc C VD] ⇒
      let [Γ ⊢ decl X A F] = [Γ ⊢ VD] in
      let [Γ ⊢ int_unr D] = unrest_exist [Γ ⊢ C] in
      (case [Γ ⊢ F] of
       | [Γ ⊢ lin] ⇒ [Γ ⊢ int_unr (unrest_l D)]
       | [Γ ⊢ unr] ⇒ [Γ ⊢ int_unr (unrest_unr D)]
       | [Γ ⊢ unav] ⇒ [Γ ⊢ int_unr (unrest_unav D)]
      )
;
```

Figure 5–12: Existence of the Unrestricted Version of a Context

`Γ:sch`, which states that context variable Γ refers to a context of schema `sch`. This schema itself is defined as **schema** `sch = var;`, which states that the context can only contain terms of type variable. Next, we have a universal quantifier `{Ψ : [Γ ⊢ ctx]}` which specifies the type of meta-variable Ψ, stating it is an explicit context under (LF) context Γ. Finally, we have the conclusion `[Γ ⊢ int_unrest_ex Ψ]`, which, as described earlier, states that Ψ has an unrestricted form.

Here, context variables are paired with a context schema (`sch`) representing a family of contexts, while meta-variables represent an abstraction over syntactic derivation trees. They refer to an abstract term under some (possibly abstract) context. This way, we can work with proofs at a more abstract level, and similarly, we can pattern match and use meta-variables to describe the new terms. We note that our meta-variable which we described under a universal quantifier must now be abstracted over with a meta-lambda **mlam** Ψ

Next, we are interested in the matching of pattern. In this case, we are matching over `[_ ⊢ Ψ]`. To explain this, we note that, while Ψ has been explicitly abstracted over, the context under which it is defined (Γ) has not. For this reason, we use the underscore to state that there should be a context present, but we do not know what it is. Afterwards, in our actual patterns, we get our abstraction for this context.

We can now look at a few of the cases. First, we have the trivial case, where our context is empty, given by `| [Γ ⊢ c_empty] ⇒ [Γ ⊢ int_unr unrest_base_e]`. As stated earlier, we now give an abstraction for the (LF) context. Moreover, since we are in a constructive setting, we must give an instantiation for an existence of an unrestricted form of the context (`int_unr`), which is given by the empty context being unrestricted (`unrest_base_e`).

```
rec ctxjoinasso1 : (Γ : sch){J1 : [Γ ⊢ joining (cjoin Ψ Ψ1 Ψ2)]}
{J2 : [Γ ⊢ joining (cjoin Ψ1 Ψ11 Ψ12)]}
[Γ ⊢ assoc_double_joining J1 J2]
```

Figure 5–13: Associativity of Context Joins

The context variable case is an more interesting pattern: `[Γ ⊢ varctx ψ[]]`. First of all, we use a meta-variable to describe the (explicit) context-variable (ψ). However, we need to associate a substitution to it to represent which variables in Γ it depends on. Since a context variable only depends on binary numbers and not on term, we thus proceed with the empty substitution `[]`. We then have a call to a helper funtion `ue_helper1` where we capture the result in a meta-variable `D` which is closed. Finally, we return this meta-variable as a proof of an appropriate unrestricted context proof. However, since it is closed, we must associate it with the empty substitution: `[Γ⊢ int_unr D[]]`.

As a final part of the proof `unr_exist`, we quickly look at the inductive case `[Γ ⊢ snoc c VD]`. There are two more things to note. First of all, here, since we do not explicitly associate a substitution to meta-variables `c` and `VD`, this means that they come with the identity substitution, and can thus depend on all the variables in Γ. Then, we also notice a recursive call, encapsulated by **let** `[Γ ⊢ int_unr D] = unrest_exist [Γ ⊢ C]` **in**.

Finally, we prove lemma 1, which represents properties of the context joins. While these are mostly obvious for the case of a concrete context, one must handle the context variables separately. This means that, in order to prove properties about context joins, we also need to prove a corresponding property about binary joins.

Let us for instance take lemma (1.2), which describes associativity of the join operation (`ctxjoinasso1` in Fig.5–13).

```
LF assoc_double_joining : joining J1 → joining J2 → type =
   | adcjoin : joining (cjoin Ψ0 Ψ12 Ψ2) → joining (cjoin Ψ Ψ11 Ψ0)
→ assoc_double_joining (J1 : joining (cjoin Ψ Ψ1 Ψ2)) (J2 : joining (cjoin Ψ1 Ψ11 Ψ12))
;
```

Figure 5–14: Special Type for Associativity of Context Join

```
let [Γ ⊢ adcjoin J1 J2] = ctxjoinasso1 [Γ ⊢ D1] [Γ ⊢ D2] in
```

Figure 5–15: Instance of Term `adcjoin`

Note that, while the arguments are taken directly from the paper definition of the lemma (two joins where that we can apply association to), the output is defined as `assoc_double_joining J1 J2` (Fig.5–14). While this might be odd at first glance, it is because our conclusion is a tuple describing two different context joins. Thus, in order to describe this, we employ a fresh LF-Type defined based on the whole theorem. Let us further examine this type:

Here, we can notice that `assoc_double_joining` takes in two objects, both joins. It only has one term, so that it respects exactly the property we desire for our theorem. The term `adcjoin` takes two arguments, both of our lemma's conclusion, and goes to type `assoc_double_joining` with both of the hypotheses. This way, when pattern matching on an object of type `assoc_double_joining`, we can retrieve both conclusions directly. This can be observed in our other associativity lemma, which uses this lemma (Fig.5–15).

Given that `D1` and `D2` are joins of the form $\Psi = \Psi_1 \bowtie \Psi_2$ and $\Psi_1 = \Psi_{11} \bowtie \Psi_{12}$, then we can extract our conclusion `J1` and `J2`, joins of the form $\Psi = \Psi_{11} \bowtie \Psi_0$ and $\Psi_0 = \Psi_{12} \bowtie \Psi_2$.

Another thing to note is that, in some cases, in order to refine the type of our arguments, we need to use a helper function of the required type. This is due to the nature of some terms which could have various types.

5.5 Typing Rules

While the typing judgements are mostly standard, there are still a few noteworthy comments to be made. First, we can notice the use of the HOAS in the typing of a lambda expression, thus making use of our LF-context, rather than only using the explicit context. Moreover, since both `M` and `B` depend on `x`, we annotate both with an application by `x` to signify their dependence, which is, abstractly, represented by the simulateneous substitution binding both meta-variables. Similarly, in the case of a linear lambda, the type `B` is independent of variable `x`, hence the lack of such an application.

Due to the fact that the definitions of the different typing judgements are mutually defined, we need mutually recursive definitions for their LF-type, which is done using the **and** keyword between the different definitions.

Next, for some of the typings, we are required to use some specification about the context. This is only done at the level of the explicit context, since Beluga, being an intuitionistic tool, does not allow us to split an LF-context. For instance, the empty spine type synthesis necessites an unrestricted context, which must be verified. Moreover, some rules require the notion of a context split to be used.

Let us now look at some of the typing rules and their implementations(Fig.5–16). The typing rules can be found in Fig.4–4.

Take a quick look at the spine-typing (`spine_synth_tp`). Given a split $\Psi = \Psi_1 \bowtie \Psi_2$, from which we can synthesize the head with Ψ_1 and the spine with Ψ_2, then we can check the whole typing with Ψ. Despite the fact a unique context split for the typing should exist, it will not be directly provided: it will need to be manually built within our proofs instead of simply using a linear implication, causing extra overhead with the handling of contexts.

Note that hereditary substitution has to be used explicitly within the rules, as dictated by the unrestricted spine type synthesis. This is done using `({x : var} subst_tp M Alpha (λx.B x) B_sub)` in term `sp_syn_lam`, which substitutes variable `x` for term `M` in type `B`. This shall be further explored in the subsequent section about hereditary substitution.

Let us look at the encoding of type-checking of canonical terms, `check_tp`. Referring back to Fig.4–4, we first have the lambda case. On paper, the hypothesis states that, if we extend our context with unrestricted variable x, then we have appropriate typing on M. Similarly, our code's hypothesis says: extending our LF context with variable `x` (`{x:var}`) and our explicit context with unrestricted variable `x` (`snoc Ψ (decl x A unr)`), then our meta-variable `M` which can depend on `x` (`M x`) has type `B`, which can also depend on `x` : `({x:var} check_tp (snoc Ψ (decl x A unr )) (M x) (B x))`.

Meanwhile, the conclusions are pretty self-explanatory. As for the linear-lambda case, it is identical, except for the fact the explicit context is extended with a linear variable. As for the head and spine case, it simply relies on the other typing rules, those for typing of head and typing of spine. However, we can see that each hypothesis has its correspondent: $\Delta; \Psi_1 \vdash H \Rightarrow A$ corresponds to `synth_tp Ψ₁ H A`; $\Delta; \Psi_2 \vdash S > A \Rightarrow P$ corresponds to `spine_synth_tp Ψ₂ S A P` and $\Psi = \Psi_1 \bowtie \Psi_2$ corresponds to `joining (cjoin Ψ₁ Ψ₂ Ψ)` (We note that, instead of explicitly defining equality, since we are expecting syntactic equality, we simply reuse the same variable). Similarly, the conclusions also correspond to one-another: $\Delta; \Psi \vdash H \cdot S \Leftarrow Q$ and `check_tp Ψ (c_atom (atom_base H S)) (atp_canon P)`.

Finally, before moving on to hereditary substitution, let us discuss the typing of simultaneous substitutions (Fig.4–5 on paper, `subst_typing` in Fig.5–17 for implementation). The only new particular cases to take into consideration can be observed in the unavailable case

```
LF check_tp : ctx → canon → tp → type =
   | chk_lam : ({x:var} check_tp (snoc Ψ (decl x A unr)) (M x) (B x))
→ check_tp Ψ (lam (λx.M x)) (pi A (λx.B x))
   | chk_llam : ({x:var} check_tp (snoc Ψ (decl x A lin)) (M x) B)
→ check_tp Ψ (llam (λx.M x)) (limp A B)
   | chk_sp : joining (cjoin Ψ1 Ψ2 Ψ) → synth_tp Ψ1 H A
→ spine_synth_tp Ψ2 S A P → check_tp Ψ (c_atom (atom_base H S)) (atp_canon P)

and synth_tp : ctx → head → tp → type =
   | syn_cst : cst_in C A S → is_unr Ψ → synth_tp Ψ (hd_cst C) A
   | syn_var_unr : ({x : var} is_unr Ψ) → synth_tp (snoc Ψ (decl x A unr)) (hd_var x) A
   | syn_var_lin : ({x : var} is_unr Ψ) → synth_tp (snoc Ψ (decl x A lin)) (hd_var x) A

and spine_synth_tp : ctx → spine → tp → atp → type =
   | sp_syn_empty : is_unr Ψ → spine_synth_tp Ψ sp_empty (atp_canon P) P
   | sp_syn_lam : unrest_ctx Ψ Ψ̄ → check_tp Ψ̄ M A → ({x : var} subst_tp M Alpha (λx.B x) B_sub)
→ spine_synth_tp Ψ S B_sub P → spine_synth_tp Ψ (sp_unr M S) (pi A (λx.B x)) P
   | sp_syn_llam : joining (cjoin Ψ Ψ1 Ψ2) → check_tp Ψ1 M A → spine_synth_tp Ψ2 S B P
→ spine_synth_tp Ψ (sp_lin M S) (limp A B) P
;
```

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

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

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

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

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

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

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

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

Figure 5–16: Typing Rules Encoding

(`stp_unav`). First, we need to use an operation (mentioned in the previous section) to retrieve the unrestricted part of a context: `unrest_ctx Phi Phi_unr`. Once again, we note that while this is deterministic, it is defined as a relation instead of a function, and uniqueness thus needs to be defined separately. Moreover, due to the definition of an unavailable substitution extension, we need to test equality of two different contexts, which is done with `ctx_eq Psi_unr Psi'_unr` (also described in the previous section).

5.6 Hereditary Substitution

First, we take a look at the reduce operation (`reduce` in Fig.5–18), which gets rid of any possible beta-reduction and ensures terms retain a normal form. These follow the rules we have defined (Fig.A–1) pretty straight-forwardly. We define each case recursively with the empty case serving as our base case. Note, however, that reduce must be defined in a mutually recursive way with the rest of the hereditary substitution definitions, due to their interdependency.

Let us now look at the more interesting and involved case of the hereditary substitution over canonical terms (`subst_canon` in Fig.5–19). The first thing to note is that we are making use of the higher-order functions to define the substitution. Indeed, since we can assume that the canonical term depends on the variable to be substituted, we define this term as a `var → canon` expression, which means that the choice of the variable, instead of being explicitly defined, is encapsulated by an LF-lambda expression. We make special note of the variable spine case with the variable to be substituted, where we must reduce the term with `M`, but also convert the spine application into a function from `x`: `λx.c_atom (atom_base (hd_var x) (S x))`.

```
LF subst_typing : sim_subst → ctx → ctx → type =
  | stp_empty    : is_unr Ψ → subst_typing s_empty c_empty Ψ
  | stp_base     : is_unr Ψ0 → ctx_mer (varctx ψ) Ψ0 Ψ1
→ subst_typing (s_id) (varctx ψ) Ψ1
  | stp_lin      : unrest_ctx Φ Φ̄ → joining (cjoin Ψ Ψ1 Ψ2)
→ subst_typing sigma Φ Ψ1 → sim_subst_tp sigma Φ̄ cer_empty A B
→ check_tp Ψ2 M B → subst_typing (s_snoc sigma M) (snoc Φ (decl x A lin)) Ψ
  | stp_unr      : unrest_ctx Φ Φ̄ → subst_typing sigma Φ Ψ
→ unrest_ctx Ψ Ψ̄ → sim_subst_tp sigma Φ̄ cer_empty A B
→ check_tp Ψ̄ M B → subst_typing (s_snoc sigma M) (snoc Φ (decl x A unr)) Ψ
  | stp_unav     : unrest_ctx Φ Phi → subst_typing sigma Φ Ψ
→ unrest_ctx Ψ Ψ̄ → unrest_ctx Ψ' Ψ̄'
→ ctx_eq Ψ̄ Ψ̄' → sim_subst_tp sigma Φ̄ cer_empty A B
→ check_tp Ψ' M B → subst_typing (s_snoc sigma M) (snoc Φ (decl x A unav)) Ψ
;
```

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

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

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

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

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

Figure 5–17: Encoding of Substitution Typing

```
LF reduce : canon → tp_appro → spine → canon → type =
  | r_unr : subst_canon N alpha M M' → reduce M' beta S R
→ reduce (lam M) (tpa_unr alpha beta) (sp_unr N S) R
  | r_lin : subst_canon N alpha M M' → reduce M' beta S R
→ reduce (llam M) (tpa_lin alpha beta) (sp_lin N S) R
  | r_empty : reduce R (tpa_a a) empty R
```

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

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

Figure 5–18: Reduce Encoding

```
and subst_canon :  canon  → tp_appro → (var → canon) → canon → type =
    | sc_lam : ({y : var} subst_canon M alpha (λx.N y x) (N' y))
→ subst_canon M alpha (λx.lam λy.N y x) (lam λy.N' y)
    | sc_llam : ({y : var} subst_canon M alpha (λx.N y x) (N' y))
→ subst_canon M alpha (λx.llam λy.N y x) (llam λy.N' y)
    | sc_cspine : subst_spine M alpha (λx. S x) (S')
→ subst_canon M alpha (λx.c_atom (atom_base (hd_cst c) (S x))) (c_atom (atom_base (hd_cst c) (S')))
    | sc_varspine_neq : ({y : var} subst_spine (M y) alpha (λx.S y x) (S' y))
→ subst_canon (M y) alpha (λx.c_atom (atom_base (hd_var y) (S y x))) (c_atom (atom_base (hd_var y) (S' y)))
    | sc_varspine_eq : subst_spine M alpha (λx.S x) (S') → reduce M alpha (S') (N)
→ subst_canon M alpha (λx.c_atom (atom_base (hd_var x) (S x))) (N)
```

Figure 5–19: Hereditary Substitution over Canonical Terms

```
LF var_look : sim_subst → (var → ctx) → canon → tp_appro → type =
  | vl_unr : depend_er A A' → is_unr Ψ
→ var_look (s_snoc sigma M) (λx.snoc Ψ (decl x A unr)) M A'
  | vl_lin : depend_er A A' → is_unr Ψ
→ var_look (s_snoc sigma M) (λx.snoc Ψ (decl x A lin)) M A'
  | vl_ind_unr : ({y : var} var_look sigma (λx.Ψ x) M A')
→ var_look (s_snoc sigma N) (λx.snoc (Ψ x) (decl y A unr)) M A'
  | vl_ind_unav : ({y : var} var_look sigma (λx.Ψ x) M A')
→ var_look (s_snoc sigma N) (λx.snoc (Ψ x) (decl y A unav)) M A'
```

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

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

Figure 5–20: Variable Lookup

5.7 Simultaneous Substitution

Similar to hereditary substitution, due to the interdependency of the different parts, we once again require a mutually-recursive definition of our LF-types in order to describe simultaneous substitution. However, unlike the former, we now need to carry around two contexts Ψ, the domain of our substitution, and an erased context $\widetilde{\Phi}$ representing bound variables encountered in our descent, as shown in Fig.4–6.

We first take into consideration the variable lookup(`var_look` in Fig.5–20). Once again, we use a function to declare which variable we are looking for. In this case, however, it is the context that becomes a function, since we are sifting through it to find the desired variable. Moreover, since the same type sometimes appears both in an erased and full form, we must declare this erasure (`depend_er A A'`).

Unlike hereditary substitution, there is no need to directly define which variable is being replaced. Instead, this is carried out through the substitution domain Ψ which we

pass around. Quickly note that, for the case where a variable appears in the substitution domain, the rules also state that it must be absent from the bound variables. This is done through the variable bindings themselves, ensuring that we are in a context devoid of the variable in question.

Let us now look at the simultaneous substitution over canonical term (`sim_subst_canon` in Fig.5–21). Looking at the type, `sim_subst → ctx → ctx_erased → canon → canon →` **type**, we carry the simultaneous substitution, both contexts and the canonical term, and have as last argument the canonical term after substituting. Thus, once again, we are left with a relation rather than a function proper. For this reason, uniqueness of substitution needs to be proven separately as a lemma. We note that, in the case of lambda abstractions, we need to extend both the explicit and LF-context. For this reason, we use a universal quantifier for `y`, `{y:var}`, to extend the LF-context, and manually extend the explicit context as an argument for the hypothesis.

Special mention must be brought to the variable cases. There are two possible cases to take into consideration: The first where the variable is only bound by the substitution, and the other one where it is bound outside the substitution (In the tracking context). In the former, we need to obtain the erased type of our variable (`var_look sigma1 (λx.Psi1 x) M alpha`) and reduce the term with the substitution to obtain a normal form. We note that, as mentioned above, we implicitly have that x is not in `Phi`. Also, the domain of the substitution must be split.

In the latter case, we need to verify that the variable is indeed in the tracking context and split this context, before converting the individual parts into their erased form. The reason for this approach, differing from the paper version, is to simplify the cases. If we have

```
and sim_subst_canon : sim_subst → ctx → ctx_erased → canon → canon → type =
    | ssc_unr : ({y : var} sim_subst_canon sigma Ψ (cer_snoc Φ (decl_er y unr)) (N y) (N' y)
→ sim_subst_canon sigma Ψ Φ (lam λy.N y) (lam λy.N' y)
    | ssc_lin : ({y : var} sim_subst_canon sigma Ψ (cer_snoc Φ (decl_er y lin)) (N y) (N' y))
→ sim_subst_canon sigma Ψ Φ (llam λy.N y) (llam λy.N' y)
    | ssc_cspine : sim_subst_spine sigma Ψ Φ S S'
→ sim_subst_canon sigma Ψ Φ (c_atom (atom_base (hd_cst c) S)) (c_atom (atom_base (hd_cst c) S'))
    | ssc_varspine_nin : reduce M alpha S' N
→ ({x : var} var_look σ1 (λx.Ψ1 x) M alpha) → ctx_er Φ Φ̃
→ joining (cjoin Ψ (Ψ1 x) Ψ2) → sim_subst_spine σ2 Ψ2 Φ̃ S S'
→ sim_subst_canon sigma Ψ Φ̃ (c_atom (atom_base (hd_var x) S)) N
    | ssc_varspine_in : ({y : var} in_ctx (λy.Φ1 y)) → joining (cjoin Φ (Φ1 y) Φ2)
→ ctx_er Φ Φ̃ → ctx_er Φ2 Φ̃2
→ sim_subst_spine σ Ψ Φ̃2 S S'
→ sim_subst_canon σ Ψ Φ̃ (c_atom (atom_base (hd_var y) S)) (c_atom (atom_base (hd_var y) S'))
```

$\boxed{[\sigma]^{\widetilde{\Phi}}_{\Psi} M}$ Substitution of the variables of Ψ in a canonical term (leaving elements of $\widetilde{\Phi}$ unchanged)

$$
\begin{array}{lll}
[\sigma]^{\widetilde{\Phi}}_{\Psi}(\lambda y.N) = \lambda y.N' & \text{where } [\sigma]^{\widetilde{\Phi},y}_{\Psi} N = N', \text{ choosing } y \notin \Psi, y \notin \mathsf{FV}(\sigma) \\
[\sigma]^{\widetilde{\Phi}}_{\Psi}(\widehat{\lambda} y.N) = \widehat{\lambda} y.N' & \text{where } [\sigma]^{\widetilde{\Phi},\widetilde{y}}_{\Psi} N = N', \text{ choosing } y \notin \Psi, y \notin \mathsf{FV}(\sigma) \\
[\sigma]^{\widetilde{\Phi}}_{\Psi}(c \cdot S) \ = c \cdot S' & \text{where } [\sigma]^{\widetilde{\Phi}}_{\Psi} S = S' \\
[\sigma]^{\widetilde{\Phi}}_{\Psi}(x \cdot S) \ = \mathsf{reduce}(M : \alpha, S') & \text{where } \Psi = \Psi_1 \bowtie \Psi_2 \text{ and } x \notin \widetilde{\Phi} \\
 & \quad \text{and } \sigma_{\Psi_1}(x) = M : \alpha \text{ and } [\sigma]^{\widetilde{\Phi}}_{\Psi_2} S = S' \\
[\sigma]^{\widetilde{\Phi}}_{\Psi}(y \cdot S) \ = y \cdot S' & \text{where } y \in \widetilde{\Phi} \text{ and } [\sigma]^{\widetilde{\Phi}}_{\Psi} S = S' \\
[\sigma]^{\widetilde{\Phi}}_{\Psi}(y \cdot S) \ = y \cdot S' & \text{where } \widetilde{y} \in \widetilde{\Phi}, \text{ and } [\sigma]^{\widetilde{\Phi}\backslash\widetilde{y}}_{\Psi} S = S'
\end{array}
$$

Figure 5–21: Simultaneous Substitution over Canonical Terms

an unrestricted variable, then the join should be with the unrestricted part of the context, while if we have a linear variable, the join should be with a context where the only linear variable is this variable. We quickly note that the `in_ctx` relation ensures that the variable is not found as unavailable, since this would be a non-sensical substitution.

5.8 Lemmas and Theorems

In our mechanization, we concentrate on key lemmas. In particular, we have proven lemmas about contexts and context join (Lemma 1) and the substitution split lemma (Lemma 4),

```
 LF sub_split : joining (cjoin Φ Φ1 Φ2) → subst_typing σ Φ Ψ → type =
   | sub_s : subst_typing σ Φ1 Ψ1 → subst_typing σ Φ2 Ψ2
→ joining (cjoin Ψ Ψ1 Ψ2)
→ sub_split (J1 : joining (cjoin Φ Φ1 Φ2)) (S : subst_typing σ Φ Ψ)
;
rec subst_split : (Γ:sch){J : [Γ ⊢ joining (cjoin Φ Φ1 Φ2)]}
{S : [ g ⊢ subst_typing σ Φ Ψ]} [g ⊢ sub_split J S]
```

Figure 5–22: Substitution Split Lemma

```
 rec subst_lemma_canon : (Γ : sch) [Γ ⊢ ctx_mer Φ1 Φ2 Ψ]
→  [Γ ⊢ check_tp Ψ M A] → [Γ ⊢ subst_typing σ Φ1 Φ1'] 
→ [Γ ⊢ unrest_ctx Φ1 Φ̄1]
→ [Γ ⊢ sim_subst_ctx σ Φ̄1 Φ2 Φ2']
→ [Γ ⊢ ctx_mer Φ1' Φ2' Ψ']
→ [Γ ⊢ ctx_er Φ2 Φ̃2]
→ [Γ ⊢ sim_subst_canon σ Φ1 Φ̃2 M N]
→ [Γ ⊢ unrest_ctx Φ2 Φ̄2]
→ [Γ ⊢ ctx_er Φ̄2 Φ̃̃2]
→ [Γ ⊢ sim_subst_tp σ Φ̄1 Φ̃̃2 A B]
→ [Γ ⊢ check_tp Ψ' N B]
```

Figure 5–23: Simultaneous Substitution Property

`subst_split` in Fig.5–22). We believe that mechanizing the remaining parts is mostly straightforward, although it requires us to keep track of many different assumptions. In the function `subst_lemma_canon` (Fig.5–23), that encodes the substitution lemma (Lemma 4), we have a total of 11 arguments. The encoding is more complex than the paper definition, because we must keep track of the linearity conditions separately. If this wasn't the case, we could reduce this number to five arguments.

As of writing this, the mechanization spans approximately 1700 lines, and a version can be found in the Beluga examples repository (`https://github.com/Beluga-lang/Beluga/tree/master/examples/lincx_mechanization`).

CHAPTER 6
Related Work

In this chapter, we present other works that are related to the formalization and mechanization of linear systems.

6.1 LLF

The idea of using a logical framework methodology to build a specification language for linear logic dates back a few decades, beginning with Cervesato and Pfenning's description of the linear logical framework LLF [10], which serves as a basis for our work. In their framework, they provide linear implication($\multimap$), additive pairs(&) and the unit type($\top$), along with intuitionistic dependent function types (Π). LLF thus serves as a conservative extension to LF permitting us to represent and reason about linear objects on top of intuitionistic ones. The idea of having a mixed system with both linear and intuitionistic variables is by no means a recent feat.

The system LLF comes with its terms and types, along with its typing rules. An interesting aspect of this language is that, while Lincx in its current state only allows for an intuitionistic representation, LLF is more general and it was shown it could work with classical logic, as an important example in the paper is the cut elimination in classical linear logic.

Moreover, the paper served to lay some ground on the notion of using linear logic to formalize imperative computations. In particular, based on Chirimar's work [12], they show how a Mini-ML with references can be embedded inside of LLF.

6.2 CELF

Developped by Anders Schack-Nielsen [35], CELF is an implementation of the concurrent logical framework CLF [40], which is an extension of LLF where monads are used to encapsulate less well-behaved operators.

This system, also having roots in LLF, is thus in essence similar to Lincx. It also comes with a linear unification. Due to its similarity to Lincx, it does give us confidence that this should also be feasible in our system. We quickly note that there are some key differences between both systems, in particular the use of affine variables in CELF, rather than purely being restricted to the case of unrestricted and linear variables. This is in part due to linear unification, and might imply such a need is also a necessary step towards the completion of Lincx.

While as of yet not present in Lincx, CELF does include some additional features from linear logic, such as the exponential, the additive conjunction and the unit. This in turn builds confidence in the possibility of adding these constructs to our language.

Finally, while the essence of both systems are similar, CELF does lack some of the features of Lincx, in particular the contextual-objects which allows us to reason more abstractly about proof-trees.

6.3 Higher-Order Representation of Substructural Logics

While linear logic can be seen through the lenses of the context, where we split the assumptions whenever using multiplicative constructors, handling it explicitly can be clumsy and unnecessarily rigorous, as can be seen through the mechanization of Lincx in Beluga (see Chapter 5).

In the absence of a satisfactory proof assistant that handles linearity, Crary [13] used an alternate approach. Pfenning [29] proposed enforcing linearity using a meta-jdgement to trace the use of assumptions throughout a typing derivation. Based on this principle, Crary observed that proof terms alone are enough to track the use of restricted assumptions. Linear logic is thus expressed using two judgements: the usual typing judgement, and a linearity judgement.

This methodology was used to implement linear logic in Twelf. Thus, representing both judgements, respectively, we have:

```
of : term → tp → type.
linear: (term → term) → type
```

The second judgements, $linear(\lambda x.M_x)$, can be read as "the variable x is used exactly once in M_x", thus respecting linearity. The first judgement, however, is similar to what we would expect, except for one difference. Whenever we bind a variable, we need to ensure that it is used linearly. For this reason, the lambda typing depends on the linearity judgement.

```
of/llam : of (llam ([x] M x )) (lolli A B)
<- ({x:term} of  x A → of (M x) B)
<- linear ([x] M x)
```

On the other hand, special attention needs to be paid to the multiplicative and additive constructors with the `linear` judgement. More explicit information is provided in the Crary's paper [13].

But while this ideology can work for some general systems, it does not allow one to reason abstractly about contexts themselves, which is one of the key features of Lincx.

6.4 Other Approaches

While the quest to design meta-logics that allow us to reason about linear logical frameworks is by no means recent, it has, in the past, been marred with difficulties.

In proof theory, McDowell and Miller [26, 27] and later Gacek et. al. [17] 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 formal 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 Ph.D. thesis [26] 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 [25] give a tailored meta-logic $\mathcal{L}_\omega^+$ for linear LF, which is an extension of the meta-logic for LF [37]. 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 on the other, offers many advantages and more robust and also supports extensions to (co)inductive definitions [7, 38].

The hybrid logical framework HLF by Reed [34] 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 [5] 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.

CHAPTER 7
Conclusion

In this work, we have presented Lincx, a linear contextual modal logical framework with first-class contexts and context variables presented using a nominal form: 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. It should thus provide a suitable foundation to implement proofs about linear derivation trees as recursive functions. Part of the work also consists in the mechanization of key equational properties of context joins in Beluga, along with progress towards proofs relating to the formalization of substitution. Our confidence in Lincx is thus further cemented through this mechanization.

7.1 Future Work

While Lincx currently has a sufficient representation to formalize some systems, it is currently not in its final form. Ideally, we would want to expand the system to a full linear logic. This means that additive constructs along with their unit would need to be added. This might prove to be a challenge, and could possibly affect the nature of canonicity, and further investigation is thus needed. Similarly, despite having unrestricted assumptions in our system, it would be interested to add the exponential to Lincx, which might require some additional work in order to better understand the relation between our intuitionistic assumptions and those obtained from transforming linear assumptions through the exponential.

While we are interested in expanding the language with more constructs from LLF, we are also interested in adding more constructs from the contextual logical framework, in particular, a great addition to the system would be that of substitution variables and first-class substitution. This would allow us to reason more abstractly about substitution. Due to the challenges this poses, the interplay between different contexts in our system should be further investigated, looking at the relationship between the domain and codomain of different substitution.

While part of the mechanization has been completed, it would be a great milestone if we could completely formalize the language and prove all of its desired properties within Beluga. Considering most proofs have been completed on paper, this should not pose too much of a challenge, and should mostly be an exercise in rigor. It remains, however, that a complete mechanization of Lincx would allow for a great level of confidence in the system.

Next, we need to consider the long term goal of Lincx, which is to be extended to a full linear programming and reasoning framework. Thus, some work needs to be expanded upon in order to achieve this state. Amongst other things, unification needs to be investigated. While some work on linear unification has been done by Anders [35], we still need to ensure adequacy of the algorithm towards Lincx, and extend it to handle Lincx' meta-objects. Similarly, pattern coverage and totality checking must also be adapted. We believe, however, that these shouldn't pose a challenge.

Finally, we would be interested in implementing such a framework. This work would follow the implementation of Beluga and possibly use the latter as a frame from which to build Lincx as a complete programming and reasoning framework. We then would like to encode some important and interesting case studies within this implementation.

REFERENCES

[1] https://www.rust-lang.org/en-us/.

[2] Olivier Savary Belanger, Stefan Monnier, and Brigitte Pientka. Programming type-safe transformations using higher-order abstract syntax. In Georges Gonthier and Michael Norrish, editors, *3rd International Conference on Certified Programs and Proofs (CPP'13)*, Lecture Notes in Computer Science (LNCS 8307), pages 243–258. Springer, 2013.

[3] Jesper Bengtson, Jonas Braband Jensen, and Lars Birkedal. Charge! - A framework for higher-order separation logic in Coq. In Lennart Beringer and Amy P. Felty, editors, *Third International Conference on Interactive Theorem Proving (ITP'12)*, Lecture Notes in Computer Science (LNCS 7406), pages 315–331. Springer, 2012.

[4] Josh Berdine, Peter W. O'Hearn, Uday S. Reddy, and Hayo Thielecke. Linear continuation-passing. *Higher-Order and Symbolic Computation*, 15(2-3):181–208, 2002.

[5] Peter Brottveit Bock and Carsten Schürmann. A contextual logical framework. In *20th International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR'15)*, Lecture Notes in Computer Science (LNCS 9450), pages 402–417. Springer, 2015.

[6] Luís Caires and Frank Pfenning. Session types as intuitionistic linear propositions. In Paul Gastin and François Laroussinie, editors, *21th International Conference on Concurrency Theory (CONCUR'10)*, Lecture Notes in Computer Science (LNCS 6269), pages 222–236. Springer, 2010.

[7] Andrew Cave and Brigitte Pientka. Programming with binders and indexed data-types. In *39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'12)*, pages 413–424. ACM, 2012.

[8] Andrew Cave and Brigitte Pientka. First-class substitutions in contextual type theory. In *8th ACM SIGPLAN International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP'13)*, pages 15–24. ACM, 2013.

[9] Andrew Cave and Brigitte Pientka. A case study on logical relations using contextual types. In I. Cervesato and K.Chaudhuri, editors, *10th International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP'15)*, pages 18–33. Electronic Proceedings in Theoretical Computer Science (EPTCS), 2015.

[10] Iliano Cervesato and Frank Pfenning. A linear logical framework. In E. Clarke, editor, *11th Annual Symposium on Logic in Computer Science*, pages 264–275, New Brunswick, New Jersey, 1996. IEEE Press.

[11] Iliano Cervesato and Frank Pfenning. A linear spine calculus. *Journal of Logic and Computation*, 13(5):639–688, 2003.

[12] J. L. Chirimar. *Proof Theoretic Approach to Specication Languages*. PhD thesis, University of Pennsylvania, 1995.

[13] Karl Crary. Higher-order representation of substructural logics. *SIGPLAN Not.*, 2010.

[14] Olivier Danvy and Andrzej Filinski. Representing control: A study of the CPS transformation. *Mathematical Structures in Computer Science*, 2(4):361–391, 1992.

[15] Henry DeYoung and Carsten Schürmann. *Linear Logical Voting Protocols*, pages 53–70. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[16] Matthew Fluet, Greg Morrisett, and Amal J. Ahmed. Linear regions are all you need. In Peter Sestoft, editor, *15th European Symposium on Programming (ESOP'06)*, Lecture Notes in Computer Science (LNCS 3924), pages 7–21. Springer, 2006.

[17] Andrew Gacek, Dale Miller, and Gopalan Nadathur. A two-level logic approach to reasoning about computations. *Journal of Automated Reasoning*, 49(2):241–273, 2012.

[18] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50(1):1–101, 1097.

[19] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, January 1993.

[20] Martin Hofmann. The strength of non-size increasing computation. *SIGPLAN Not.*, 37(1):260–269, January 2002.

[21] J. Maraist, M. Odersky, D.N. Turner, and P. Wadler. Call-by-name, call-by-value, call-by-need and the linear lambda calculus. *Theoretical Computer Science*, 1999.

[22] Chris Martens, Anne-Gwenn Bosser, João F. Ferreira, and Marc Cavazza. *Linear Logic Programming for Narrative Generation*. Springer Berlin Heidelberg, 2013.

[23] Chris Martens and Karl Crary. 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)*, pages 23–32. ACM, 2012.

[24] Andrew McCreight. Practical tactics for separation logic. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *22nd International Conference on Theorem Proving in Higher Order Logics (TPHOLs'09)*, Lecture Notes in Computer Science (LNCS 5674), pages 343–358. Springer, 2009.

[25] Andrew McCreight and Carsten Schürmann. A meta-linear logical framework. In *4th International Workshop on Logical Frameworks and Meta-Languages (LFM'04)*, 2004.

[26] Raymond McDowell. *Reasoning in a Logic with Definitions and Induction*. PhD thesis, University of Pennsylvania, 1997.

[27] Raymond C. McDowell and Dale A. Miller. Reasoning with higher-order abstract syntax in a logical framework. *ACM Transactions on Computational Logic*, 3(1):80–136, 2002.

[28] Aleksandar Nanevski, Frank Pfenning, and Brigitte Pientka. Contextual modal type theory. *ACM Transactions on Computational Logic*, 9(3):1–49, 2008.

[29] Frank Pfenning. Structural cut elimination in linear logic. Technical report, Carnegie Mellon University, 1994.

[30] Brigitte Pientka. 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)*, pages 371–382. ACM, 2008.

[31] Brigitte Pientka and Andrew Cave. Inductive Beluga:Programming Proofs (System Description). In Amy P. Felty and Aart Middeldorp, editors, *25th International Conference on Automated Deduction (CADE-25)*, Lecture Notes in Computer Science (LNCS 9195), pages 272–281. Springer, 2015.

[32] Brigitte Pientka and Joshua Dunfield. Beluga: a framework for programming and reasoning with deductive systems (System Description). In J. Giesl and R. Haehnle, editors, *5th International Joint Conference on Automated Reasoning (IJCAR'10)*, Lecture Notes in Artificial Intelligence (LNAI 6173), pages 15–21. Springer, 2010.

[33] Paolo Pistone. Polymorphism and the notion of type: the input of linear logic. 2016.

[34] Jason Reed. *A hybrid logical framework*. PhD thesis, Carnegie Mellon, 2009.

[35] Anders Schack-Nielsen. *Implementing Substructural Logical Frameworks.* PhD thesis, IT University of Copenhagen, 2011.

[36] Anders Schack-Nielsen and Carsten Schürmann. Pattern unification for the lambda calculus with linear and affine types. In Karl Crary and Marino Miculan, editors, *International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP'10)*, volume 34 of *Electronic Proceedings in Theoretical Computer Science (EPTCS)*, pages 101–116, July 2010.

[37] Carsten Schürmann. *Automating the Meta Theory of Deductive Systems.* PhD thesis, Department of Computer Science, Carnegie Mellon University, 2000. CMU-CS-00-146.

[38] David Thibodeau, Andrew Cave, and Brigitte Pientka. Indexed codata. In Jacques Garrigue, Gabriele Keller, and Eijiro Sumii, editors, *21st ACM SIGPLAN International Conference on Functional Programming (ICFP'16)*, pages 351–363. ACM, 2016.

[39] David Walker and Kevin Watkins. On regions and linear types. In Benjamin C. Pierce, editor, *6th ACM SIGPLAN International Conference on Functional Programming (ICFP'01)*, pages 181–192. ACM, 2001.

[40] Kevin Watkins, Iliano Cervesato, Frank Pfenning, and David Walker. A concurrent logical framework I: Judgments and properties. Technical Report CMU-CS-02-101, Department of Computer Science, Carnegie Mellon University, 2002.

APPENDIX A
Appendix

We present here partial proofs and generalized reformulations of lemmas and theorems mentioned in Chapter 4 of this paper. We also give definitions of single hereditary substitution and simultaneous meta-substitution.

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

Proof. Proof by induction on typing derivation $\mathcal{D} :: \Delta; \Psi \vdash \sigma \Leftarrow \Phi$. We show a couple of cases below, the remaining ones are straightforward.

Case. $\mathcal{D} = \dfrac{\mathsf{unr}(\Gamma)}{\Delta; \psi_m, \Gamma \vdash \mathsf{id}_{\psi_m} \Leftarrow \psi_m}$

$\mathsf{unr}(\Gamma)$ by assumption

$\Delta; \psi_\epsilon, \Gamma \vdash \mathsf{id}_{\psi_m} \Leftarrow \psi_\epsilon$ by substitution typing

Case. $\mathcal{D} = \dfrac{\Delta; \Psi_1 \vdash \sigma \Leftarrow \Phi \qquad \Delta; \Psi_2 \vdash M \Leftarrow [\sigma]_{\overline{\Phi}} A \qquad \Psi = \Psi_1 \bowtie \Psi_2}{\Delta; \Psi \vdash \sigma, M \Leftarrow \Phi, x\hat{:}A}$

$\Delta; \overline{\Psi_1} \vdash \sigma \Leftarrow \overline{\Phi}$ by IH

$\overline{\Psi} = \overline{\Psi_1} = \overline{\Psi_2}$ by Lemma 1(4)

$\Delta; \overline{\Psi} \vdash \sigma, M \Leftarrow \overline{\Phi}, x\check{:}A$ by substitution typing □

A.1 Hereditary Single Substitution

Hereditary single substitution in LINCX closely follows [8]. We present complete rules on Fig. A–1.

The following theorem establishes typing for single substitutions. Notice that it is more general compared to the variant presented in Chapter 4.

Theorem 2 (Hereditary single substitution property). *1. If* $\Delta; \overline{\Psi} \vdash M \Leftarrow A$ *and* $\Delta; \Psi, x{:}A, \Phi \vdash J$ *then* $\Delta; \Psi, [M/x]_A\Phi \vdash [M/x]^*_A(J)$ *where* $* \in \{c, s, l\}$.

2. If $\Delta; \Psi_1 \vdash M \Leftarrow A$, $\Delta; \Psi_2, x\hat{:}A, \Phi \vdash J$ *and* $\Psi = \Psi_1 \bowtie \Psi_2$ *then* $\Delta; \Psi, \Phi \vdash [M/x]^*_A(J)$ *where* $* \in \{c, s, l\}$.

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* $\Delta; \Psi \vdash M' \Leftarrow B$

A.2 Typing for Meta-Terms

Rules for constructing a context of a given schema presented on Fig. A–2 describe four possible initial cases of context construction, which correspond to four cases of constructing a valid context. Note that since their instantiations should contain the same variables (albeit with some variables becoming unavailable), all the context variables should have the same schema. Next, we have three cases for context extension, i.e.: extending a context with either an unrestricted, a linear or an unavailable variable. In either case, we must ensure that the schema as an element of the proper type, and the proper status, for the context variable, which is to say, an unrestricted variable should expects an unrestricted element, while a linear or unavailable variable expects a linear schema element.

Typing of other meta-terms, as presented on Fig. A–3 is straightforward: a meta-object $\widetilde{\Psi}.R$ has type $(\Psi \vdash P)$, if R has type P in the context Ψ. The typing of variable objects, i.e. an object of type $(\Psi \vdash \#A)$, must be reconsidered carefully. A parameter type is inhabited only by variable objects, i.e. either concrete variables from Ψ or parameter variable associated

with a variable substitution. The typing for parameter variables follows the typing for meta-objects. There are two cases to consider when we have a concrete variable x from the context: either Ψ contains only unrestricted variable declarations and $x : A$ is one of them; or x is in fact a linear variable of type A which forces Ψ to be a context with only one linear declaration $x\hat{:}A$.

A.3 Meta-Substitution

$\boxed{\mathsf{id}(\Psi)}$ Identity substitution

$$\begin{aligned}
\mathsf{id}(\cdot) &= \cdot \\
\mathsf{id}(\mathsf{id}_\psi) &= \mathsf{id}_\psi \\
\mathsf{id}(\Psi, x{:}A) &= \mathsf{id}(\Psi), \eta\text{-}\mathsf{exp}_{(\mathsf{A}^-)}(x, \epsilon) \\
\mathsf{id}(\Psi, x\hat{:}A) &= \mathsf{id}(\Psi), \eta\text{-}\mathsf{exp}_{(\mathsf{A}^-)}(x, \epsilon) \\
\mathsf{id}(\Psi, x\check{:}A) &= \mathsf{id}(\Psi), \eta\text{-}\mathsf{exp}_{(\mathsf{A}^-)}(x, \epsilon)
\end{aligned}$$

$\boxed{\eta\text{-}\mathsf{exp}_{\mathsf{A}}(H, S)}$ η-expansion

$$\begin{aligned}
\eta\text{-}\mathsf{exp}_{\mathsf{a}}(H, S) &= H \cdot S \\
\eta\text{-}\mathsf{exp}_{\alpha\to\beta}(H, S) &= \lambda x.\eta\text{-}\mathsf{exp}_{\beta}(H, S@\eta\text{-}\mathsf{exp}_{\alpha}(x)) \\
\eta\text{-}\mathsf{exp}_{\alpha\multimap\beta}(H, S) &= \widehat{\lambda} x.\eta\text{-}\mathsf{exp}_{\beta}(H, S\widehat{@}\eta\text{-}\mathsf{exp}_{\alpha}(x))
\end{aligned}$$

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

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

$\boxed{[M/x]^c_\alpha N = N'}$ N' is a result of substituting M for x in a canonical term N

$$\begin{aligned}
[M/x]^c_\alpha(\lambda y.N) &= \lambda y.N' && \text{where } [M/x]^c_\alpha N = N', \text{ choosing } y \neq x, y \notin \mathsf{FV}(M) \\
[M/x]^c_\alpha(\widehat{\lambda} y.N) &= \widehat{\lambda} y.N' && \text{where } [M/x]^c_\alpha N = N', \text{ choosing } y \neq x, y \notin \mathsf{FV}(M) \\
[M/x]^c_\alpha(u[\sigma]) &= u[\sigma'] && \text{where } [M/x]^s_\alpha \sigma = \sigma' \\
[M/x]^c_\alpha(c \cdot S) &= c \cdot S' && \text{where } [M/x]^l_\alpha S = S' \\
[M/x]^c_\alpha(x \cdot S) &= N && \text{where } [M/x]^l_\alpha S = S' \text{ and } \mathsf{reduce}(M : \alpha, S') = N \\
[M/x]^c_\alpha(y \cdot S) &= y \cdot S' && \text{where } [M/x]^l_\alpha S = S' \text{ and } x \neq y \\
[M/x]^c_\alpha(p[\sigma] \cdot S) &= p[\sigma'] \cdot S' && \text{where } [M/x]^s_\alpha \sigma = \sigma' \text{ and } [M/x]^l_\alpha S = S'
\end{aligned}$$

$\boxed{[M/x]^l_\alpha S = S'}$ S' is a result of substituting M for x in a spine S

$$\begin{aligned}
[M/x]^l_\alpha(\epsilon) &= \epsilon \\
[M/x]^l_\alpha(N\,;S) &= N'\,;S' && \text{where } [M/x]^c_\alpha N = N' \text{ and } [M/x]^l_\alpha S = S' \\
[M/x]^l_\alpha(N\,\hat{;}\,S) &= N'\,\hat{;}\,S' && \text{where } [M/x]^c_\alpha N = N' \text{ and } [M/x]^l_\alpha S = S'
\end{aligned}$$

$\boxed{[M/x]^s_\alpha \sigma = \sigma'}$ σ' is a result of substituting M for x in a substitution σ

$$\begin{aligned}
[M/x]^s_\alpha(\cdot) &= \cdot \\
[M/x]^s_\alpha(\mathsf{id}_\psi) &= \mathsf{id}_\psi \\
[M/x]^s_\alpha(\sigma, N) &= \sigma', N' && \text{where } [M/x]^s_\alpha \sigma = \sigma' \text{ and } [M/x]^c_\alpha N = N'
\end{aligned}$$

Figure A–1: Hereditary Single Substitution

$\boxed{\Delta \vdash \Psi \Leftarrow G}$ Context Ψ checks against schema G

$$\frac{}{\Delta \vdash \cdot \Leftarrow G} \qquad \frac{\psi_i : G \in \Delta}{\Delta \vdash \psi_\epsilon \Leftarrow G} \qquad \frac{\psi_i : G \in \Delta}{\Delta \vdash \psi_i \Leftarrow G}$$

$$\frac{\Delta \vdash \psi_k \Leftarrow G \quad \Delta \vdash \psi_l \Leftarrow G \quad m = k \bowtie l}{\Delta \vdash \psi_m \Leftarrow G}$$

$$\frac{\Delta \vdash \Psi \Leftarrow G \quad \lambda(\overrightarrow{x_i{:}A_i}).B \in G \quad \Delta; \Psi \vdash \sigma \Leftarrow (\overrightarrow{x_i{:}A_i}) \quad \Delta; \overline{\Psi} \vdash A = [\sigma]\overline{\overrightarrow{(x_i{:}A_i)}}B}{\Delta \vdash \Psi, x{:}A \Leftarrow G}$$

$$\frac{\Delta \vdash \Psi \Leftarrow G \quad \lambda(\overrightarrow{x_i{:}A_i}).\widehat{B} \in G \quad \Delta; \Psi \vdash \sigma \Leftarrow (\overrightarrow{x_i{:}A_i}) \quad \Delta; \overline{\Psi} \vdash A = [\sigma]\overline{\overrightarrow{(x_i{:}A_i)}}B}{\Delta \vdash \Psi, x\hat{:}A \Leftarrow G}$$

$$\frac{\Delta \vdash \Psi \Leftarrow G \quad \lambda(\overrightarrow{x_i{:}A_i}).\widehat{B} \in G \quad \Delta; \Psi \vdash \sigma \Leftarrow (\overrightarrow{x_i{:}A_i}) \quad \Delta; \overline{\Psi} \vdash A = [\sigma]\overline{\overrightarrow{(x_i{:}A_i)}}B}{\Delta \vdash \Psi, x\check{:}A \Leftarrow G}$$

Figure A–2: Typing Rules for Contexts of a Given Schema

$\boxed{\Delta \vdash C \Leftarrow U}$ Meta-level term C checks agains type U

$$\frac{\Delta; \Psi \vdash R \Leftarrow P}{\Delta \vdash \widetilde{\Psi}.R \Leftarrow (\Psi \vdash P)}$$

$$\frac{x{:}A \in \Psi \quad \mathsf{unr}(\Psi)}{\Delta \vdash \widetilde{\Psi}.x \Leftarrow (\Psi \vdash \#A)} \qquad \frac{\mathsf{unr}(\Psi_1) \quad \mathsf{unr}(\Psi_2)}{\Delta \vdash (\widetilde{\Psi_1}, \widehat{x}, \widetilde{\Psi_2}).x \Leftarrow (\Psi_1, x\hat{:}A, \Psi_2 \vdash \#A)}$$

$$\frac{p : (\Phi \vdash \#A) \in \Delta \quad \Delta; \Psi \vdash \pi \Leftarrow \Phi \quad \Delta; \overline{\Psi} \vdash B = [\pi]_{\overline{\Phi}}(A)}{\Delta \vdash \widetilde{\Psi}.p[\pi] \Leftarrow (\Psi \vdash \#B)}$$

Figure A–3: Typing Rules for Meta-Terms

$\boxed{[\![\Theta]\!]_\Delta M}$ Simultaneous meta-substitution for terms

$$
\begin{array}{lll}
[\![\Theta]\!]_\Delta(\lambda x.M) & = \lambda x.M' & \text{where } [\![\Theta]\!]_\Delta M = M' \\
[\![\Theta]\!]_\Delta(\widehat{\lambda} x.M) & = \widehat{\lambda} x.M' & \text{where } [\![\Theta]\!]_\Delta M = M' \\
[\![\Theta]\!]_\Delta(u[\sigma]) & = R' & \text{where } \Theta_\Delta(u) = \widetilde{\Psi}.R : (\Psi \vdash P) \text{ and } [\![\Theta]\!]_\Delta\, \sigma = \sigma' \\
 & & \text{and } [\sigma']_\Psi R = R' \\
[\![\Theta]\!]_\Delta(c \cdot S) & = c \cdot S' & \text{where } [\![\Theta]\!]_\Delta\, S = S' \\
[\![\Theta]\!]_\Delta(x \cdot S) & = x \cdot S' & \text{where } [\![\Theta]\!]_\Delta\, S = S' \\
[\![\Theta]\!]_\Delta(p[\sigma] \cdot S) & = M' & \text{where } \Theta_\Delta(p) = \widetilde{\Psi}.x : (\Psi \vdash \#A) \text{ and } [\![\Theta]\!]_\Delta\, \sigma = \sigma' \\
 & & \text{and } [\![\Theta]\!]_\Delta\, S = S' \text{ and } \sigma'_\Psi(x) = M : \alpha \\
 & & \text{and } \mathsf{reduce}(M : \alpha, S') = M' \\
[\![\Theta]\!]_\Delta(p[\sigma] \cdot S) & = q[\tau'] \cdot S' & \text{where } \Theta_\Delta(p) = \widetilde{\Psi}.q[\pi] : (\Psi \vdash \#A) \text{ and } [\![\Theta]\!]_\Delta\, \sigma = \sigma' \\
 & & \text{and } [\![\Theta]\!]_\Delta\, S = S' \text{ and } [\sigma']\pi = \tau
\end{array}
$$

$\boxed{[\![\Theta]\!]_\Delta \sigma}$ Simultaneous meta-substitution for substitutions

$$
\begin{array}{lll}
[\![\Theta]\!]_\Delta(\cdot) & = \cdot & \\
[\![\Theta]\!]_\Delta(\mathsf{id}_\psi) & = \mathsf{id}(\Psi) & \text{where } \Theta_\Delta(\psi_\epsilon) = \Psi \\
[\![\Theta]\!]_\Delta(\sigma, M) & = \sigma', M' & \text{where } [\![\Theta]\!]_\Delta\, \sigma = \sigma' \text{ and } [\![\Theta]\!]_\Delta\, M = M'
\end{array}
$$

$\boxed{[\![\Theta]\!]_\Delta \Psi}$ Simultaneous meta-substitution for contexts

$$
\begin{array}{lll}
[\![\Theta]\!]_\Delta(\cdot) & = \cdot & \\
[\![\Theta]\!]_\Delta(\psi_m) & = \Psi & \text{where } \Theta_\Delta(\psi_m) = \Psi \\
[\![\Theta]\!]_\Delta(\Psi, x{:}A) & = \Psi', x{:}A' & \text{where } [\![\Theta]\!]_\Delta\, \Psi = \Psi' \text{ and } [\![\Theta]\!]_\Delta\, A = A' \\
[\![\Theta]\!]_\Delta(\Psi, x\hat{:}A) & = \Psi', x\hat{:}A' & \text{where } [\![\Theta]\!]_\Delta\, \Psi = \Psi' \text{ and } [\![\Theta]\!]_\Delta\, A = A' \\
[\![\Theta]\!]_\Delta(\Psi, x\check{:}A) & = \Psi', x\check{:}A' & \text{where } [\![\Theta]\!]_\Delta\, \Psi = \Psi' \text{ and } [\![\Theta]\!]_\Delta\, A = A'
\end{array}
$$

Figure A–4: Simultaneous Meta-Substitution