Typing with Leftovers

A mechanization of Intuitionistic Multiplicative-Additive Linear Logic

Guillaume Allais
iCIS, Radboud University Nijmegen, The Netherlands
gallais@cs.ru.nl

Abstract

We start from an untyped, well-scoped λ-calculus and introduce a bidirectional typing relation corresponding to a Multiplicative-Additive Intuitionistic Linear Logic. We depart from typical presentations to adopt one that is well-suited to the intensional setting of Martin-Löf Type Theory. This relation is based on the idea that a linear term consumes some of the resources available in its context whilst leaving behind leftovers which can then be fed to another program.

Concretely, this means that typing derivations have both an input and an output context. This leads to a notion of weakening (the extra resources added to the input context come out unchanged in the output one), a rather direct proof of stability under substitution, an analogue of the frame rule of separation logic showing that the state of unused resources can be safely ignored, and a proof that typechecking is decidable. Finally, we demonstrate that this alternative formalization is sound and complete with respect to a more traditional representation of Intuitionistic Linear Logic.

The work has been fully formalised in Agda, commented source files are provided as additional material available at `https://github.com/gallais/typing-with-leftovers`.

2012 ACM Subject Classification Theory of computation → Type theory,Theory of computation → Linear logic

Keywords and phrases Type System, Bidirectional, Linear Logic, Agda

Digital Object Identifier 10.4230/LIPIcs.TYPES.2017.1

Supplement Material https://github.com/gallais/typing-with-leftovers

Funding The research leading to these results has received funding from the European Research Council under the European Union's Seventh Framework Programme (FP7/2007-2013) / ERC grant agreement nr. 320571

1 Introduction

The strongly-typed functional programming community has benefited from a wealth of optimisations made possible precisely because the library author as well as the compiler are aware of the type of the program they are working on. These optimisations have ranged from Danvy's type-directed partial evaluation [18] residualising specialised programs to erasure mechanisms –be they user-guided like Coq's extraction [27] which systematically removes all the purely logical proofs put in Prop by the developer or automated like Brady and Tejiščák's erased values [12, 13]– and including the library defining the State-Thread [25] monad which relies on higher-rank polymorphism and parametricity to ensure the safety of using an actual mutable object in a lazy, purely functional setting.

However, in the context of the rising development of dependently-typed programming languages [11, 31] which, unlike ghc's Haskell [36], incorporate a hierarchy of universes in order to ensure that the underlying logic is consistent, some of these techniques are not applicable anymore. Indeed, the use of large quantification in the definition of the ST-monad crucially relies on impredicativity. As a consequence, the specification of programs allowed to update a mutable object in a safe way has to change.

23rd International Conference on Types for Proofs and Programs (TYPES 2017).
Editors: Andreas Abel, Fredrik Nordvall Forsberg, and Ambrus Kaposi; Article No. 1; pp. 1:1–1:21
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Idris has been extended with experimental support for uniqueness types inspired by Clean's [1] and Rust's ownership types [20], all of which stem from linear logic [22].

In order to be able to use type theory to formally study the meta-theory of the programming languages whose type system includes notions of linearity, we need to have a good representation of such constraints.

Section 2 introduces the well-scoped untyped λ-calculus we are going to use as our language of raw terms. Section 3 defines the linear typing rules for this language as relations which record the resources consumed by a program. The next sections are dedicated to proving properties of this type system: Section 4 proves that the status of unused variables rightfully does not matter, Section 5 (and respectively Section 6) demonstrates that these typing relations are stable under weakening (respectively substitution), Section 7 demonstrates that these relations are functional, and Section 8 that they are decidable i.e. provides us with a typechecking algorithm. Finally Section 9 goes back to a more traditional presentation of Intuitionistic Multiplicative-Additive Linear Logic and demonstrates it is equivalent to our type system.

Notations

This whole development has been fully formalised in Agda. Rather than using Agda's syntax, the results are reformulated in terms of definitions, lemmas, theorems, and examples. However it is important to keep in mind the distinction between various kinds of objects. `Teletype` is used to denote data constructors, SMALL CAPITALS are characteristic of defined types. A type family's index is written as a subscript e.g. $\textsc{Var}_n$.

We use two kinds of inference rules to describe inductive families: double rules are used to define types whilst simple ones correspond to constructors. In each case the premises correspond to arguments (usually called parameters and indices for types) and the conclusion shows the name of the constructor. A typical example is the inductively defined set of unary natural numbers. The inductive type is called NAT and it has two constructors: 0 takes no argument whilst 1+· takes a NAT n and represents its successor.

$$\frac{}{\textsc{Nat} : \mathit{Set}} \qquad \frac{}{\texttt{0} : \textsc{Nat}} \qquad \frac{n : \textsc{Nat}}{\texttt{1+}n : \textsc{Nat}}$$

2 The Calculus of Raw Terms

The calculus we study in this paper is meant to be a core language, even though it will be rather easy to write programs in it. As a consequence all the design choices have been guided by the goal of facilitating its mechanical treatment in a dependently-typed language. That is why we use de Bruijn indices to represent variable bindings. We demonstrate in the code accompanying the paper how to combine a parser and a scope checker to turn a surface level version of the language using strings as variable names into this representation.

Following Bird and Patterson [9] and Altenkirch and Reus [5], we define the raw terms of our language not as an inductive type but rather as an inductive *family* [21]. This technique, sometimes dubbed "type-level de Bruijn indices", makes it possible to keep track, in the index of the family, of the free variables currently in scope. As is nowadays folklore, instead of using a set-indexed presentation where a closed terms is indexed by the empty set ⊥ and fresh variables are introduced by wrapping the index in a `Maybe` type constructor[1], we index our terms by a natural number instead.

[1] The value `nothing` represents the fresh variable whilst the constructor `just` lifts the other ones in the new scope.

The VAR type family[2] defined below represents the de Bruijn indices [19] corresponding to the n free variables present in a scope n.

$$\frac{n : \text{NAT}}{\text{VAR}_n : \mathit{Set}} \qquad \frac{}{\texttt{z} : \text{VAR}_{1+n}} \qquad \frac{k : \text{VAR}_n}{\texttt{s}\ k : \text{VAR}_{1+n}}$$

We present the calculus in a bidirectional fashion [32]. This definition style scales well to more complex type theories where full type-inference is not tractable anymore whilst keeping the type annotations the programmer needs to add to a minimum. The term constructors of the calculus are split in two different syntactic categories corresponding to constructors of canonical values on one hand and eliminators on the other. These categories characterise the flow of information during typechecking: given a context assigning a type to each free variable, canonical values (which we call CHECK) can be *check*ed against a type whilst we may *infer* the type of computations (which we call INFER). Each type is indexed by a scope:

$$\frac{n : \text{NAT}}{\text{INFER}_n : \mathit{Set}} \qquad \frac{n : \text{NAT}}{\text{CHECK}_n : \mathit{Set}}$$

On top of the constructors one would expect for a usual definition of the untyped λ-calculus ($\texttt{var}\cdot$, $\texttt{app}\cdot\cdot$, and $\texttt{lam}\cdot$) we have constructors and eliminators for sums ($\texttt{inl}\cdot$, $\texttt{inr}\cdot$, $\texttt{case}\cdot\texttt{return}\cdot\texttt{of}\cdot\,\%\,\cdot$), products ($\texttt{prd}\cdot\cdot$, $\texttt{let}\cdot\texttt{:=}\cdot\texttt{in}\cdot$, $\texttt{prj}_1\cdot$, $\texttt{prj}_2\cdot$), unit ($\texttt{unit}$, $\texttt{let}\cdot\texttt{:=}\cdot\texttt{in}\cdot$) and void ($\texttt{exfalso}\cdot\cdot$). Two additional rules ($\texttt{neu}\cdot$ and $\texttt{cut}\cdot\cdot$ respectively) allow the embedding of INFER into CHECK and vice-versa. They make it possible to form redexes by embedding canonical values into computations and then applying eliminators to them. In terms of typechecking, they correspond to a change of direction between inferring and checking.

$$\begin{array}{lcl}
\langle \text{INFER}_n \rangle & ::= & \texttt{var}\ \langle \text{VAR}_n \rangle \\
& \mid & \texttt{app}\ \langle \text{INFER}_n \rangle\ \langle \text{CHECK}_n \rangle \\
& \mid & \texttt{case}\ \langle \text{INFER}_n \rangle\ \texttt{return}\ \langle \text{TYPE} \rangle\ \texttt{of}\ \langle \text{CHECK}_{1+n} \rangle\ \%\ \langle \text{CHECK}_{1+n} \rangle \\
& \mid & \texttt{prj}_1\ \langle \text{INFER}_n \rangle \mid \texttt{prj}_2\ \langle \text{INFER}_n \rangle \\
& \mid & \texttt{exfalso}\ \langle \text{TYPE} \rangle\ \langle \text{INFER}_n \rangle \\
& \mid & \texttt{cut}\ \langle \text{CHECK}_n \rangle\ \langle \text{TYPE} \rangle \\
\\
\langle \text{CHECK}_n \rangle & ::= & \texttt{lam}\ \langle \text{CHECK}_{1+n} \rangle \\
& \mid & \texttt{let}\ \langle \text{PATTERN}_m \rangle \texttt{:=} \langle \text{INFER}_n \rangle \texttt{in}\ \langle \text{CHECK}_{m+n} \rangle \\
& \mid & \texttt{unit} \\
& \mid & \texttt{inl}\ \langle \text{CHECK}_n \rangle \mid \texttt{inr}\ \langle \text{CHECK}_n \rangle \\
& \mid & \texttt{prd}\ \langle \text{CHECK}_n \rangle\ \langle \text{CHECK}_n \rangle \\
& \mid & \texttt{neu}\ \langle \text{INFER}_n \rangle
\end{array}$$

Figure 1 Grammar of the Language of Raw Terms

The constructors `cut`, `case`, and `exfalso` take an extra TYPE argument in order to guarantee the success and uniqueness of type-inference for INFER terms.

A notable specificity of this language is the ability to use nested patterns in a let binder rather than having to resort to cascading lets. This is achieved thanks to a rather simple piece of kit: the PATTERN

[2] It is also known as `Fin` (for "finite set") in the dependently typed programming community.

type family. A value of type PATTERN_n represents an irrefutable pattern binding n variables. Because variables are represented as de Bruijn indices, the base pattern does not need to be associated with a name, it simply is a constructor `v` binding exactly one variable. The brackets pattern $\langle\rangle$ matches unit values and binds nothing. The comma pattern constructor takes two nested patterns respectively binding m and n variables and uses them to deeply match a pair thus binding $(m+n)$ variables.

$$\frac{n : \text{NAT}}{\text{PATTERN}_n : \mathit{Set}} \qquad \frac{}{\mathtt{v} : \text{PATTERN}_1} \qquad \frac{}{\langle\rangle : \text{PATTERN}_0} \qquad \frac{p : \text{PATTERN}_m \qquad q : \text{PATTERN}_n}{p\,\mathbf{,}\,q : \text{PATTERN}_{m+n}}$$

The grammar of raw terms only guarantees that all expressions are well-scoped by construction. It does not impose any other constraint, which means that a user may write valid programs but also invalid ones as the following examples demonstrate:

▶ **Example 1.** `swap` is a closed, well-typed linear term taking a pair as an input and swapping its components. It corresponds to the mathematical function $(x, y) \mapsto (y, x)$.

```
swap = lam (let (v , v) := var z
            in prd (neu (var (s z))) (neu (var z)))
```

▶ **Example 2.** `illTyped` is a closed linear term. However it is manifestly ill-typed: the let-binding it uses tries to break down a function as if it were a pair.

```
illTyped = let (v , v) := cut (lam (neu (var z))) (a ⊸ a)
           in prd (neu (var z)) (neu (var (s z)))
```

▶ **Example 3.** Finally, `diagonal` is a term typable in the simply-typed lambda calculus but it is not linear: it duplicates its input just like $x \mapsto (x, x)$ does.

```
diagonal = lam (prd (neu (var z)) (neu (var z)))
```

3 Linear Typing Rules

These examples demonstrate that we need to define a typing relation describing the rules terms need to abide by in order to qualify as well-typed linear programs. We start by defining the types our programs may have using the grammar in Figure 2. Apart from the usual linear type formers, we have a constructor κ which makes it possible to have countably many different base types.

$$\begin{array}{rcl} \langle\text{TYPE}\rangle & ::= & \kappa\ \langle\mathbb{N}\rangle \mid \mathbb{0} \mid \mathbb{1} \\ & \mid & \langle\text{TYPE}\rangle \multimap \langle\text{TYPE}\rangle \mid \langle\text{TYPE}\rangle \otimes \langle\text{TYPE}\rangle \\ & \mid & \langle\text{TYPE}\rangle \oplus \langle\text{TYPE}\rangle \mid \langle\text{TYPE}\rangle \ \&\ \langle\text{TYPE}\rangle \end{array}$$

Figure 2 Grammar of TYPE

A linear type system is characterised by the fact that all the resources available in the context have to be used exactly once by the term being checked. In traditional presentations of linear logic this is achieved by representing the context as a multiset and, in each rule, cutting it up and distributing its parts among the premises. This is epitomised by the introduction rule for tensor.

However, multisets are an intrinsically *extensional* notion and therefore quite arduous to work with in an *intensional* type theory. Various strategies can be applied to tackle this issue; most of them rely on using linked lists to represent contexts together with ways to reorganise the context.

In Figure 3 we show two of the most common representations of the tensor rule. The first one splits the context into Γ and Δ and dispatches them into the subproofs; it relies on the existence of structural rules which the user will be able to use to reorganise the context appropriately. The second one is a combined rule letting the user re-arrange the context on the fly by using the notion of "bag-equivalence" for lists denoted $\_ \approx \_$.

$$\frac{\Gamma \vdash \sigma \qquad \Delta \vdash \tau}{\Gamma, \Delta \vdash \sigma \otimes \tau}\otimes_i \qquad\qquad \frac{\Gamma \vdash \sigma \qquad \Delta \vdash \tau \qquad \Gamma, \Delta \approx \Theta}{\Theta \vdash \sigma \otimes \tau}\otimes_i$$

Figure 3 Introduction rules for tensor (left: usual presentation, right: with reordering on the fly)

In both of these cases, the user has to explicitly rearrange the context either by using structural rules or proving that two distinct contexts are bag equivalent. Although one can find coping mechanisms to handle such clunky systems (for instance using a solver for bag-equivalence [17] based on the proof-by-reflection [10] approach to automation), we would rather not.

All of these strategies are artefacts of the unfortunate mismatch between the ideal mathematical objects one wishes to model and their internal representation in the proof assistant. Short of having proper quotient types, this will continue to be an issue when dealing with multisets. The solution described in the rest of this paper is syntax-directed; it does not try to replicate a set-theoretic approach in intuitionistic type theory but rather strives to find the type theoretical structures which can make the problem more tractable. Indeed, given the right abstractions most proofs become direct structural inductions.

3.1 Usage Annotations

McBride's recent work [29] on combining linear and dependent types highlights the distinction one can make between referring to a resource and actually consuming it. In the same spirit, rather than dispatching the available resources in the appropriate subderivations, we consider that a term is checked in a *given* context on top of which usage annotations are super-imposed. These usage annotations indicate whether resources have been consumed already or are still available. Type-inference (resp. Type-checking) is then inferring (resp. checking) a term's type but *also* annotating the resources consumed by the term in question and returning the *leftovers* which gave their name to this paper.

▶ **Definition 4.** A **CONTEXT** is a list of TYPEs indexed by its length. It can be formally described by the following inference rules:

$$\frac{n : \text{NAT}}{\text{CONTEXT}_n : \mathit{Set}} \qquad \frac{}{[] : \text{CONTEXT}_0} \qquad \frac{\gamma : \text{CONTEXT}_n \qquad \sigma : \text{TYPE}}{\gamma \bullet \sigma : \text{CONTEXT}_{1+n}}$$

▶ **Definition 5.** A **USAGE** is a predicate on a type σ describing whether the resource associated to it is available or not. We name the constructors describing these two states `f` (for **fresh**) and `s` (for **stale**) respectively. These are naturally lifted to contexts in a pointwise manner and we reuse the USAGE name and the `f` and `s` names for the functions taking a context and returning either a fully fresh or fully stale USAGE for it.

$$\frac{\sigma : \text{TYPE}}{\text{USAGE}_\sigma : \mathit{Set}} \qquad \frac{}{\mathtt{f}_\sigma : \text{USAGE}_\sigma} \qquad \frac{}{\mathtt{s}_\sigma : \text{USAGE}_\sigma}$$

$$\frac{\gamma : \text{CONTEXT}_n}{\text{USAGE}_\gamma : \mathit{Set}} \qquad \frac{}{[] : \text{USAGE}_{[]}} \qquad \frac{\Gamma : \text{USAGE}_\gamma \quad S : \text{USAGE}_\sigma}{\Gamma \bullet S : \text{USAGE}_{\gamma\bullet\sigma}}$$

3.2 Typing as Consumption Annotation

A Typing relation seen as a consumption annotation process describes what it means to analyze a term in a scope of size n: given a context of types for these n variables, and a usage annotation for that context, it ascribes a type to the term whilst crafting another usage annotation containing all the leftover resources. Formally:

▶ **Definition 6.** A **Typing Relation** for T a NAT-indexed inductive family is an indexed relation $\mathcal{T}_n$ such that:

$$\frac{n : \text{NAT} \quad \gamma : \text{CONTEXT}_n \quad \Gamma, \Delta : \text{USAGE}_\gamma \quad t : T_n \quad \sigma : \text{TYPE}}{\mathcal{T}_n(\Gamma, t, \sigma, \Delta) : \mathit{Set}}$$

This definition clarifies the notion but also leads to more generic statements later on: weakening, substitution, framing can all be expressed as properties a Typing Relation might have. We can already list the typing relations introduced later on in this article which fit this pattern. We have split their arguments into three columns depending on whether they should be understood as either inputs (the known things), scrutinees (the things being validated), or outputs (the things that we learn) and hint at what the flow of information in the typechecker will be.

$$\frac{\begin{array}{lcl} \gamma : \text{CONTEXT}_n & & \sigma : \text{TYPE} \\ \Gamma : \text{USAGE}_\gamma & k : \text{VAR}_n & \Delta : \text{USAGE}_\gamma \end{array}}{\Gamma \vdash_v k \in \sigma \boxtimes \Delta : \mathit{Set}}\ \text{VAR}$$

$$\frac{\begin{array}{lcl} \gamma : \text{CONTEXT}_n & & \sigma : \text{TYPE} \\ \Gamma : \text{USAGE}_\gamma & t : \text{INFER}_n & \Delta : \text{USAGE}_\gamma \end{array}}{\Gamma \vdash t \in \sigma \boxtimes \Delta : \mathit{Set}}\ \text{INFER}$$

$$\frac{\begin{array}{lcl} \gamma : \text{CONTEXT}_n & & \\ \Gamma : \text{USAGE}_\gamma & t : \text{CHECK}_n & \Delta : \text{USAGE}_\gamma \\ \sigma : \text{TYPE} & & \end{array}}{\Gamma \vdash \sigma \ni t \boxtimes \Delta : \mathit{Set}}\ \text{CHECK}$$

$$\frac{\sigma : \text{TYPE} \quad p : \text{PATTERN}_n \quad \gamma : \text{CONTEXT}_n}{\sigma \ni p \rightsquigarrow \gamma : \mathit{Set}}\ \text{PATTERN}$$

Figure 4 Typing relations for VAR, INFER, CHECK and PATTERN

▶ Remark. The use of $\boxtimes$ is meant to suggest that the input Γ gets distributed between the type σ of the term and the leftovers Δ obtained as an output. Informally $\Gamma \simeq \sigma \otimes \Delta$, hence the use of a tensor-like symbol.

3.2.1 Typing de Bruijn indices

The simplest instance of a Typing Relation is the one for de Bruijn indices: given an index k and a usage annotation, it successfully associates a type σ to that index if and only if the kth resource in context is of type σ and fresh (i.e. its USAGE_σ is $\mathtt{f}_\sigma$). In the resulting leftovers, this resource will have turned stale ($\mathtt{s}_\sigma$) because it has now been used:

▶ **Definition 7.** The typing relation for VAR is presented in a sequent-style: $\Gamma \vdash_v k \in \sigma \boxtimes \Delta$ means that starting from the usage annotation Γ, the de Bruijn index k is ascribed type σ with leftovers Δ. It is defined inductively by two constructors (cf. Figure 5).

$$\frac{}{\Gamma \bullet \mathtt{f}_\sigma \vdash_v \mathtt{z} \in \sigma \boxtimes \Gamma \bullet \mathtt{s}_\sigma} \qquad \frac{\Gamma \vdash_v k \in \sigma \boxtimes \Delta}{\Gamma \bullet A \vdash_v \mathtt{s}\, k \in \sigma \boxtimes \Delta \bullet A}$$

Figure 5 Typing rules for VAR

▶ Remark. The careful reader will have noticed that there is precisely one typing rule for each VAR constructor. It is not a coincidence. And if these typing rules are not named it's because in Agda, they can be given the same name as their VAR counterpart and the typechecker will perform type-directed disambiguation. The same will be true for INFER, CHECK and PATTERN which means that writing down a typable program could be seen as either writing a raw term or the typing derivation associated to it depending on the author's intent.

▶ **Example 8.** The de Bruijn index 1 has type τ in the context $(\gamma \bullet \tau \bullet \sigma)$ with usage annotation $(\Gamma \bullet \mathtt{f}_\tau \bullet \mathtt{f}_\sigma)$, no matter what Γ actually is:

$$\frac{\dfrac{}{\Gamma \bullet \mathtt{f}_\tau \vdash \mathtt{z} \in \tau \boxtimes \Gamma \bullet \mathtt{s}_\tau}}{\Gamma \bullet \mathtt{f}_\tau \bullet \mathtt{f}_\sigma \vdash \mathtt{s}\ \mathtt{z} \in \tau \boxtimes \Gamma \bullet \mathtt{s}_\tau \bullet \mathtt{f}_\sigma}$$

Or, as it would be written in Agda, taking advantage of the fact that the language constructs and the typing rules about them have been given the same names:

```
one : Γ • f τ • f σ ⊢ s z ∈ τ ⊠ Γ • s τ • f σ
one = s z
```

3.2.2 Typing Terms

We now face compound untyped terms such as $\mathtt{app}\ f\ t$ whose subterms f and t have been defined in the *same* scope of size n. Therefore the typing relation for these terms needs to use the *same* context of size n for both premises. Trying to cut up a CONTEXT_n in two just like in Figure 3 would not only be cumbersome, it wouldn't be type correct. This is where usage annotations shine.

The key idea appearing in all the typing rules for compound expressions is to use the input USAGE to type one of the sub-expressions, collect the leftovers from that typing derivation and use them as the new input USAGE when typing the next sub-expression.

Another common pattern can be seen across all the rules involving binders, be they λ-abstractions, let-bindings or branches of a case. Typechecking the body of a binder involves extending the input USAGE with fresh variables and observing that they have become stale in the output one. This guarantees that these bound variables cannot escape their scope as well as that they have indeed been used. Although not the focus of this paper, it is worth noting that relaxing the staleness restriction would lead to an affine type system which would be interesting in its own right.

▶ **Definition 9.** The Typing Relation for INFER is typeset in a fashion similar to the one for VAR: in both cases the type is inferred. $\Gamma \vdash t \in \sigma \boxtimes \Delta$ means that given Γ a USAGE$_\gamma$, and t an INFER, the type σ is inferred together with leftovers Δ, another USAGE$_\gamma$. The rules are listed in Figure 6.

$$\frac{\Gamma \vdash_v k \in \sigma \boxtimes \Delta}{\Gamma \vdash \texttt{var}\ k \in \sigma \boxtimes \Delta} \qquad \frac{\Gamma \vdash t \in \sigma \multimap \tau \boxtimes \Delta \qquad \Delta \vdash \sigma \ni u \boxtimes \Theta}{\Gamma \vdash \texttt{app}\ t\ u \in \tau \boxtimes \Theta}$$

$$\frac{\Gamma \vdash t \in \sigma \oplus \tau \boxtimes \Delta \qquad \begin{array}{c}\Delta \bullet \texttt{f}_\sigma \vdash \nu \ni l \boxtimes \Theta \bullet \texttt{s}_\sigma \\ \Delta \bullet \texttt{f}_\tau \vdash \nu \ni r \boxtimes \Theta \bullet \texttt{s}_\tau\end{array}}{\Gamma \vdash \texttt{case}\ t\ \texttt{return}\ \nu\ \texttt{of}\ l \invamp r \in \nu \boxtimes \Theta} \qquad \frac{\Gamma \vdash t \in \sigma \& \tau \boxtimes \Delta}{\Gamma \vdash \texttt{prj}_1\ t \in \sigma \boxtimes \Delta}$$

$$\frac{\Gamma \vdash t \in \sigma \& \tau \boxtimes \Delta}{\Gamma \vdash \texttt{prj}_2\ t \in \tau \boxtimes \Delta} \qquad \frac{\Gamma \vdash t \in \mathbb{0} \boxtimes \Delta}{\Gamma \vdash \texttt{exfalso}\ \sigma\ t \in \sigma \boxtimes \Delta} \qquad \frac{\Gamma \vdash \sigma \ni t \boxtimes \Delta}{\Gamma \vdash \texttt{cut}\ t\ \sigma \in \sigma \boxtimes \Delta}$$

Figure 6 Typing rules for INFER

▶ **Definition 10.** For CHECK, the type σ comes first: $\Gamma \vdash \sigma \ni t \boxtimes \Delta$ means that given Γ a USAGE$_\gamma$, a type σ, the CHECK t can be checked to have type σ with leftovers Δ. The rules can be found in Figure 7.

$$\frac{\Gamma \bullet \texttt{f}_\sigma \vdash \tau \ni b \boxtimes \Delta \bullet \texttt{s}_\sigma}{\Gamma \vdash \sigma \multimap \tau \ni \texttt{lam}\ b \boxtimes \Delta} \qquad \frac{\Gamma \vdash \sigma \ni a \boxtimes \Delta \qquad \Delta \vdash \tau \ni b \boxtimes \Theta}{\Gamma \vdash \sigma \otimes \tau \ni \texttt{prd}\ a\ b \boxtimes \Theta}$$

$$\frac{\Gamma \vdash \sigma \ni t \boxtimes \Delta}{\Gamma \vdash \sigma \oplus \tau \ni \texttt{inl}\ t \boxtimes \Delta} \qquad \frac{\Gamma \vdash \tau \ni t \boxtimes \Delta}{\Gamma \vdash \sigma \oplus \tau \ni \texttt{inr}\ t \boxtimes \Delta} \qquad \frac{\Gamma \vdash \sigma \ni a \boxtimes \Delta \qquad \Gamma \vdash \tau \ni b \boxtimes \Delta}{\Gamma \vdash \sigma \& \tau \ni \texttt{prd}\ a\ b \boxtimes \Delta}$$

$$\frac{}{\Gamma \vdash \mathbb{1} \ni \texttt{unit} \boxtimes \Gamma} \qquad \frac{\begin{array}{c}\Gamma \vdash t \in \sigma \boxtimes \Delta \qquad \sigma \ni p \leadsto \delta \\ \Delta \mathbin{+\!\!+} \texttt{f}_\delta \vdash \tau \ni u \boxtimes \Theta \mathbin{+\!\!+} \texttt{s}_\delta\end{array}}{\Gamma \vdash \tau \ni \texttt{let}\ p := t\ \texttt{in}\ u \boxtimes \Theta} \qquad \frac{\Gamma \vdash t \in \sigma \boxtimes \Delta}{\Gamma \vdash \sigma \ni \texttt{neu}\ t \boxtimes \Delta}$$

Figure 7 Typing rules for CHECK

We can see that both variants of a product type –tensor ($\otimes$) and with ($\&$)– use the same surface language constructor but are disambiguated in a type-directed manner in the checking relation. The premises are naturally widely different: With lets its user pick which of the two available types they want and as a consequence both components have to be proven using the same resources. Tensor on the other hand forces the user to use both so the leftovers are threaded from one premise to the other.

▶ **Definition 11.** Finally, PATTERNs are checked against a type and a context of newly bound variables is generated. If the variable pattern always succeeds, the pair constructor pattern on the other hand only succeeds if the type it attempts to split is a tensor type. The context of newly-bound variables is then the collection of the contexts associated to the nested patterns. The rules are given in Figure 8.

$$\frac{}{\sigma \ni \mathtt{v} \rightsquigarrow [\,] \bullet \sigma} \qquad \frac{}{\mathbb{1} \ni \langle\rangle \rightsquigarrow [\,]} \qquad \frac{\sigma \ni p \rightsquigarrow \gamma \qquad \tau \ni q \rightsquigarrow \delta}{\sigma \otimes \tau \ni (p\,,q) \rightsquigarrow \delta +\!\!+ \gamma}$$

Figure 8 Typing rules for PATTERN

Example 12. Given these rules, we see that the identity function can be checked at type $(\sigma \multimap \sigma)$ in an empty context:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{[\,] \bullet \mathtt{f}_\sigma \vdash_v \mathtt{z} \in \sigma \boxtimes [\,] \bullet \mathtt{s}_\sigma}}{[\,] \bullet \mathtt{f}_\sigma \vdash \mathtt{var}\ \mathtt{z} \in \sigma \boxtimes [\,] \bullet \mathtt{s}_\sigma}}{[\,] \bullet \mathtt{f}_\sigma \vdash \sigma \ni \mathtt{neu}\,(\mathtt{var}\ \mathtt{z}) \boxtimes [\,] \bullet \mathtt{s}_\sigma}}{[\,] \vdash \sigma \multimap \sigma \ni \mathtt{lam}\,(\mathtt{neu}\,(\mathtt{var}\ \mathtt{z})) \boxtimes [\,]}$$

Or, as it would be written in Agda where the typing rules were given the same name as their term constructor counterparts:

```
identity : [] ⊢ σ ⊸ σ ∋ lam (neu (var z)) ⊠ []
identity = lam (neu (var z))
```

Example 13. It is also possible to revisit Example 1 to prove that `swap` can be checked against type $(\sigma \otimes \tau) \multimap (\tau \otimes \sigma)$ in an empty context. This gives the lengthy derivation included in the appendix or the following one in Agda which is quite a lot more readable:

```
swapTyped : [] ⊢ (σ ⊗ τ) ⊸ (τ ⊗ σ) ∋ swap ⊠ []
swapTyped = lam (let (v , v) := var z
                 in prd (neu (var (s z))) (neu (var z))
```

4 Framing

The most basic property one can prove about this typing system is the fact that the state of the resources which are not used by a lambda term is irrelevant. We call this property the Framing Property because of the obvious analogy with the frame rule in separation logic. This can be reformulated as the fact that as long as two pairs of an input and an output USAGE exhibit the same consumption pattern then if a derivation uses one of these, it can use the other one instead. Formally (postponing the definition of $\Gamma - \Delta \equiv \Theta - \Xi$):

Definition 14. A Typing Relation $\mathcal{T}$ for a NAT-indexed family T has the **Framing Property** if for all k a NAT, γ a CONTEXT$_k$, Γ, Δ, Θ, Ξ four USAGE$_\gamma$, t an element of T_k and σ a Type, if $\Gamma - \Delta \equiv \Theta - \Xi$ and $\mathcal{T}_k(\Gamma, \mathrm{t}, \sigma, \Delta)$ then $\mathcal{T}_k(\Theta, \mathrm{t}, \sigma, \Xi)$ also holds.

Remark. This is purely a property of the type system as witnessed by the fact that the term t is left unchanged wich won't be the case when defining stability under Weakening or Substitution for instance.

▶ **Definition 15. Consumption Equivalence** for a given CONTEXT γ characterises the pairs of an input and an output USAGE_γ which have the same consumption pattern. The usages annotations for the empty context are trivially related. If the context is not empty, then there are two cases: if the resource is left untouched on one side, then so should it on the other side but the two annotations may be different (here denoted A and B respectively). On the other hand, if the resource has been consumed on one side then it has to be on the other side too.

$$\frac{\Gamma, \Delta, \Theta, \Xi \,:\, \text{USAGE}_\gamma}{\Gamma—\Delta \equiv \Theta—\Xi \,:\, \mathit{Set}} \qquad \frac{}{[]—[] \equiv []—[]}$$

$$\frac{\Gamma—\Delta \equiv \Theta—\Xi}{(\Gamma \bullet A)—(\Delta \bullet A) \equiv (\Theta \bullet B)—(\Xi \bullet B)} \qquad \frac{\Gamma—\Delta \equiv \Theta—\Xi}{(\Gamma \bullet \mathtt{f}_\sigma)—(\Delta \bullet \mathtt{s}_\sigma) \equiv (\Theta \bullet \mathtt{f}_\sigma)—(\Xi \bullet \mathtt{s}_\sigma)}$$

▶ Remark. Two pairs of usages which are consumption equivalent are defined over the *same* context (and thus scope). Stability of a typing rule with respect to consumption equivalence will not be sufficient to introduce new variables. This will be dealth with by defining weakening in Section 5.

▶ **Definition 16.** The **Consumption Partial Order** $\Gamma \subseteq \Delta$ is defined as $\Gamma — \Delta \equiv \Gamma — \Delta$. It orders USAGE from least consumed to maximally consumed.

▶ **Lemma 17.** *The following properties on the Consumption relations hold:*

1. *The consumption equivalence is a partial equivalence [30].*
2. *The consumption partial order is a partial order.*
3. *If there is a USAGE X "in between" two others Γ and Δ according to the consumption partial order (i.e. $\Gamma \subseteq \mathrm{X}$ and $\mathrm{X} \subseteq \Delta$), then any pair of USAGE Θ, Ξ consumption equal to Γ and Δ (i.e. $\Gamma—\Delta \equiv \Theta—\Xi$) can be split in a manner compatible with X. In other words: one can find Z such that $\Gamma—\mathrm{X} \equiv \Theta—\mathrm{Z}$ and $\mathrm{X}—\Delta \equiv \mathrm{Z}—\Xi$.*

▶ **Lemma 18** (Consumption)**.** *The Typing Relations for VAR, INFER and CHECK all imply that if a typing derivation exists with input USAGE annotation Γ and output USAGE annotation Δ then $\Gamma \subseteq \Delta$.*

▶ **Theorem 19.** *The Typing Relation for VAR has the Framing Property. So do the ones for INFER and CHECK.*

Proof. The proofs are by structural induction on the typing derivations. They rely on the previous lemmas to, when faced with a rule with multiple premises and leftover threading, generate the inclusion evidence and use it to split up the witness of consumption equivalence and distribute it appropriately in the induction hypotheses. ◀

▶ **Example 20.** Coming back to the typing derivation for the de Bruijn index 1 in Example 8, we can use the Framing theorem to transport the proof that $\Gamma \bullet \mathtt{f}_\tau \bullet \mathtt{f}_\sigma \vdash \mathtt{s\ z} \in \tau \boxtimes \Gamma \bullet \mathtt{s}_\tau \bullet \mathtt{f}_\sigma$ to a proof that $\Delta \bullet \mathtt{f}_\tau \bullet \mathtt{s}_\sigma \vdash \mathtt{s\ z} \in \tau \boxtimes \Delta \bullet \mathtt{s}_\tau \bullet \mathtt{s}_\sigma$ for any Δ. Indeed, we can see that these two pairs of USAGE are consumption equivalent ($\Gamma — \Gamma \equiv \Delta — \Delta$ holds by induction):

$$\dfrac{\dfrac{\dfrac{\vdots}{\Gamma—\Gamma \equiv \Delta—\Delta}}{\Gamma \bullet \mathtt{f}_\tau—\Gamma \bullet \mathtt{s}_\tau \equiv \Delta \bullet \mathtt{f}_\tau—\Delta \bullet \mathtt{s}_\tau}}{\Gamma \bullet \mathtt{f}_\tau \bullet \mathtt{f}_\sigma—\Gamma \bullet \mathtt{s}_\tau \bullet \mathtt{f}_\sigma \equiv \Delta \bullet \mathtt{f}_\tau \bullet \mathtt{s}_\sigma—\Delta \bullet \mathtt{s}_\tau \bullet \mathtt{s}_\sigma}$$

5 Weakening

It is perhaps surprising to find a notion of weakening for a linear calculus: the whole point of linearity is precisely to ensure that all the resources are used. However, when opting for a system based on consumption annotations it becomes necessary to be able to extend the context a term lives in. This will typically be used in the definition of parallel substitution to push the substitution under a binder. Linearity is guaranteed by ensuring that the inserted variables are left untouched by the term.

Weakening arises from a notion of inclusion. The appropriate type theoretical structure to describe these inclusions is well-known and called an Order Preserving Embeddding [15, 4]. Unlike a simple function witnessing the inclusion of its domain into its codomain, the restriction brought by order preserving embeddings guarantees that contraction is simply not possible which is crucial in a linear setting.

▶ **Definition 21.** An **Order Preserving Embedding** (OPE) is an inductive family. Its constructors (dubbed "moves" in this paper) describe a strategy to realise the promise of an injective embedding which respects the order induced by the de Bruijn indices. We start with an example in Figure **??** before giving, in Figure 9, the formal definition of OPEs for NAT, CONTEXT and USAGE.

In the following example, we prove that the source context $\gamma \bullet \tau$ can be safely embedded into the target one $\gamma \bullet \sigma \bullet \tau \bullet \nu$, written $\gamma \bullet \tau \leq \gamma \bullet \sigma \bullet \tau \bullet \nu$. This example proof uses all three of the moves the inductive definition of OPEs offers: $\texttt{insert}_\alpha$ which introduces a new variable of type α, `copy` which embeds the source context's top variable, and `done` which simply copies the source context. Because we read strategies left-to-right and it is easier to see how they act if contexts are also presented left-to-right, we temporarily switch to *cons*-style (i.e. σ, γ) instead of the *snoc*-style (i.e. $\gamma \bullet \sigma$) used in the rest of this paper.

▶ **Example 22.** An Order Preserving Embedding proving: $\gamma \bullet \tau \leq \gamma \bullet \sigma \bullet \tau \bullet \nu$

OPE	$\texttt{insert}_\nu$		`copy`		$\texttt{insert}_\sigma$		`done`
source			τ	,			γ
target	ν	,	τ	,	σ	,	γ

Now that we have seen an example, we can focus on the formal definition. We give the definition of OPE for NAT, CONTEXT and USAGE all side by side in one table: the first column lists the names of the constructors associated to each move whilst the other ones give their corresponding types for each category. It is worth noting that OPEs for CONTEXT are indexed over the ones for NAT and the OPEs for USAGE are indexed by both. The latter definitions are effectively algebraic *ornaments* [16, 28] over the previous ones, that is to say they have the same structure only storing additional information.

	NAT	CONTEXT	USAGE
`done`	$\dfrac{}{k \leq k}$	$\dfrac{}{\gamma \leq \gamma}$	$\dfrac{}{\Gamma \leq \Gamma}$
`copy`	$\dfrac{k \leq l}{1{+}k \leq 1{+}l}$	$\dfrac{\gamma \leq \delta}{\gamma \bullet \sigma \leq \delta \bullet \sigma}$	$\dfrac{\Gamma \leq \Delta \quad S : \mathrm{USAGE}_\sigma}{\Gamma \bullet S \leq \Delta \bullet S}$
`insert`	$\dfrac{k \leq l}{k \leq 1{+}l}$	$\dfrac{\gamma \leq \delta}{\gamma \leq \delta \bullet \sigma}$	$\dfrac{\Gamma \leq \Delta \quad S : \mathrm{USAGE}_\sigma}{\Gamma \leq \Delta \bullet S}$

Figure 9 Order Preserving Embeddings for NAT, CONTEXT and USAGE

- The first row defines the move `done`. It is the strategy corresponding to the trivial embedding of a set into itself by the identity function and serves as a base case.
- The second row corresponds to the `copy` move which extends an existing embedding by copying the current 0th variable from source to target. The corresponding cases for CONTEXTs and USAGE are purely structural: no additional content is required to be able to perform a `copy` move.
- The third row describes the move `insert` which introduces an extra variable in the target set. This is the move used to extend an existing context, i.e. to weaken it. In this case, it is paramount that the OPE for CONTEXTs should take a type σ as an extra argument (it will be the type of the newly introduced variable) whilst the OPE for USAGE takes a USAGE_σ (it will be the usage associated to that newly introduced variable of type σ).

Now that the structure of these OPEs is clear, we have to introduce a caveat regarding this description: the CONTEXT and USAGE case are a bit special. They do not in fact mention the source and target sets in their indices. This is a feature: when weakening a typing relation, the OPE for USAGE will be applied simultaneously to the input *and* the output USAGE which, although of a similar structure because of their shared CONTEXT index, will be different.

▶ **Definition 23.** The **semantics of an OPE** is defined by induction over the proof object. We use the overloaded function name ope(·) for it. They behave as the simplified view given in Figure 9 where γ / Γ is seen as the input, σ / S the additional information stored into the proof object and δ / Δ the output.

We leave out the definition of weakening for raw terms which is the standard definition for the untyped λ-calculus. It proves that given $k \leq l$ we can turn an INFER_k (respectively CHECK_k) into an INFER_l (respectively CHECK_l). It is given by a simple structural induction on the terms themselves, using `copy` to go under binders.

▶ **Definition 24.** A Typing Relation $\mathcal{T}_.$ for a NAT-indexed family T such that we have a function $\texttt{weak}_T$ transporting proofs that $k \leq l$ to functions $T_k \to T_l$ is said to have the **Weakening Property** if for all k, l in NAT, o a proof that $k \leq l$, O a proof that $\text{OPE}(o)$ and $\mathcal{O}$ a proof that $\text{OPE}(O)$ then for all γ a CONTEXT_k, Γ and Δ two USAGE_γ, t an element of T_k and σ a TYPE, if $\mathcal{T}_k(\Gamma, t, \sigma, \Delta)$ holds true then we also have $\mathcal{T}_l(\text{ope}(\mathcal{O}, \Gamma), \texttt{weak}_T(o, t), \sigma, \text{ope}(\mathcal{O}, \Delta))$.

▶ **Theorem 25.** *The Typing Relation for VAR has the Weakening Property. So do the Typing Relations for INFER and CHECK.*

Proof. The proof for VAR is by induction on the typing derivation. The statements for INFER and CHECK are proved by mutual structural inductions on the respective typing derivations. Using the `copy` constructor of OPEs is crucial to be able to go under binders. ◀

Unlike the framing property, this theorem is not purely about the type system: the term is indeed modified between the premise and the conclusion. Now that we know that weakening is compatible with the typing relations, let us study substitution.

6 Substituting

Stability of the typing relations under substitution guarantees that the untyped evaluation of programs will yield results which have the same type as well as preserve the linearity constraints. The notion of leftovers naturally extends to substitutions: the terms meant to be substituted for the variables in context which are not used by a term will not be used when pushing the substitution onto this term. They will therefore have to be returned as leftovers.

Because of this rather unusual behaviour for substitution, picking the right type-theoretical representation for the environment carrying the values to be substituted in is a bit subtle. Indeed, relying on the usual combination of weakening and crafting a fresh variable when going under a binder becomes problematic. The leftovers returned by the induction hypothesis would then live in an extended context and quite a lot of effort would be needed to downcast them back to the smaller context they started in. The solution is to have an explicit constructor for "going under a binder" which can be simply peeled off on the way out of a binder. The values are still weakened to fit in the extended context they end up in but that happens at the point of use (i.e. when they are being looked up to replace a variable) instead of when pushing the substitution under a binder.

▶ **Definition 26.** The environment ENV used to define substitution for raw terms is indexed by two NATs k and l where k is the source's scope and l is the target's scope. There are three constructors: one for the empty environment ([]), one for going under a binder ($\bullet\mathtt{v}$) and one to extend an environment with an INFER_l.

$$\frac{k, l \,:\, \mathrm{NAT}}{\mathrm{ENV}(k, l) \,:\, Set} \qquad \frac{}{[] \,:\, \mathrm{ENV}(0, l)} \qquad \frac{\rho \,:\, \mathrm{ENV}(k, l)}{\rho \bullet\mathtt{v} \,:\, \mathrm{ENV}(1{+}k, 1{+}l)} \qquad \frac{\rho \,:\, \mathrm{ENV}(k, l) \qquad t \,:\, \mathrm{INFER}_l}{\rho \bullet t \,:\, \mathrm{ENV}(1{+}k, l)}$$

Environment are carrying INFER elements because, being in the same syntactical class as VARs, they can be substituted for them without any issue. We now state the substitution lemma on untyped terms because it is, unlike the one for weakening, non-standard by way of our definition of environments.

▶ **Lemma 27.** *Given a VAR_k v and an ENV(k, l) ρ, we can look up the INFER_l associated to v in ρ.*

Proof. The proof goes by induction on v and case analysis on ρ. If the variable we look up has been introduced by a binder we went under using the constructor $\bullet\mathtt{v}$ then we return it immediately. Otherwise we get our hands on a term which we may need to weaken. This corresponds to the following functional specification (the practical implementation can be distinct to avoid retraversing the term once for every single binder we went under):

$$\begin{array}{ll}
\mathtt{var\ z} & [\rho \bullet\mathtt{v}] = \mathtt{z} \\
\mathtt{var\ z} & [\rho \bullet t] = t \\
\mathtt{var}\ (\mathtt{s}\ v) & [\rho \bullet\mathtt{v}] = \mathtt{lift}(\mathtt{var}\ v\,[\,\rho\,]) \\
\mathtt{var}\ (\mathtt{s}\ v) & [\rho \bullet t] = \mathtt{var}\ v\,[\,\rho\,]
\end{array}$$

where `lift` is an instance of weakening defined in the previous section which takes a term in a scope of size k and returns the same term in scope $1{+}k$. ◀

▶ **Lemma 28.** *Raw terms are stable under substitutions: for all k and l, given t a term INFER_k (resp. CHECK_k) and ρ an environment $\mathrm{ENV}(k, l)$, we can apply the substitution ρ to t and obtain an INFER_l (resp. CHECK_l).*

Proof. By mutual induction on the raw terms. The traversals are purely structural except when going under binders where the constructor $\bullet\mathtt{v}$ is used to extend the ENV appropriately. The prototypical case of a binder is the `lam` one, and its functional specification is: $(\mathtt{lam}\ t)\,[\,\rho\,] = \mathtt{lam}\,(t\,[\,\rho \bullet\mathtt{v}\,])$. ◀

▶ **Definition 29.** The **environments** used when proving that Typing Relations are stable under substitution follow closely the ones for raw terms. $\Theta_1 \vdash_e \Gamma \ni \rho \boxtimes \Theta_2$ is a typing relation with input usages Θ_1 and output Θ_2 for the raw substitution ρ targeting the fresh variables in Γ. The typing for the empty environment has the same input and output usages annotation. Formally:

$$\frac{\begin{array}{lll} \theta : \text{CONTEXT}_l & & \\ \Theta_1 : \text{USAGE}_\theta & & \\ \gamma : \text{CONTEXT}_k & \rho : \text{ENV}(k,l) & \Theta_2 : \text{USAGE}_\theta \\ \Gamma : \text{USAGE}_\gamma & & \end{array}}{\Theta_1 \vdash_e \Gamma \ni \rho \boxtimes \Theta_2 : \mathit{Set}} \qquad \frac{}{\Theta_1 \vdash_e [] \ni [] \boxtimes \Theta_1}$$

For **f**resh variables in Γ, there are two cases depending on whether they have been introduced by going under a binder or not. If it is not the case then the typing environment carries around a typing derivation for the term t meant to be substituted for this variable. Otherwise, it does not carry anything extra but tracks in its input / output usages annotation the fact that the variable has been consumed.

$$\frac{\Theta_1 \vdash t \in \sigma \boxtimes \Theta_2 \qquad \Theta_2 \vdash_e \Gamma \ni \rho \boxtimes \Theta_3}{\Theta_1 \vdash_e \Gamma \bullet \mathtt{f}_\sigma \ni \rho \bullet t \boxtimes \Theta_3} \qquad \frac{\Theta_1 \vdash_e \Gamma \ni \rho \boxtimes \Theta_2}{\Theta_1 \bullet \mathtt{f}_\sigma \vdash_e \Gamma \bullet \mathtt{f}_\sigma \ni \rho \bullet \mathtt{v} \boxtimes \Theta_2 \bullet \mathtt{s}_\sigma}$$

For **s**tale variables, there are two cases too. They are however a bit more similar: none of them carry around an extra typing derivation. The main difference is in the shape of the input and output context: in the case for the "going under a binder" constructor, they are clearly enriched with an extra (now consumed) variable whereas it is not the case for the normal environment extension.

$$\frac{\Theta_1 \vdash_e \Gamma \ni \rho \boxtimes \Theta_2}{\Theta_1 \vdash_e \Gamma \bullet \mathtt{s}_\sigma \ni \rho \bullet t \boxtimes \Theta_2} \qquad \frac{\Theta_1 \vdash_e \Gamma \ni \rho \boxtimes \Theta_2}{\Theta_1 \bullet \mathtt{s}_\sigma \vdash_e \Gamma \bullet \mathtt{s}_\sigma \ni \rho \bullet \mathtt{v} \boxtimes \Theta_2 \bullet \mathtt{s}_\sigma}$$

▶ **Definition 30.** A Typing Relation $\mathcal{T}$ for a NAT-indexed family T equipped with a function $\mathtt{subst}_T$ which for all NATs k, l, given an element T_k and an ENV(k,l) returns an element T_l is said to be **stable under substitution** if for all NATs k and l, γ a CONTEXT$_k$, Γ and Δ two USAGE$_\gamma$, t an element of T_k, σ a Type, ρ an ENV(k,l), θ a CONTEXT$_l$ and Θ_1 and Θ_3 two USAGE$_\theta$ such that $\mathcal{T}_k(\Gamma, t, \sigma, \Delta)$ and $\Theta_1 \vdash_e \Gamma \ni \rho \boxtimes \Theta_3$ holds then there exists a Θ_2 of type USAGE$_\theta$ such that $\mathcal{T}_l(\Theta_1, \mathtt{subst}_T(t,\rho), \sigma, \Theta_2)$ and $\Theta_2 \vdash_e \Delta \ni \rho \boxtimes \Theta_3$.

▶ **Theorem 31.** *The Typing Relations for INFER and CHECK are stable under substitution.*

Proof. The proof by mutual structural induction on the typing derivations relies heavily on the fact that these Typing Relations enjoy the framing property in order to adjust the USAGE annotations. ◀

7 Functionality

In the next section we will prove that type-checking and type-inference are decidable. In the cases where the check fails, we have to prove that any purported typing derivation would leave to a contradiction. Thse arguments all follow a similar pattern: assuming that a typing derivation exist, we use inversion lemmas to obtain results in direct contradiction to the observations we have made. These inversion lemmas often rely on the fact that the typing relations are functional.

Although we did highlight that some of our relations' indices are meant to be seen as inputs whilst others are supposed to be outputs, we have not yet made this relationship formal because this fact was seldom used in the proofs so far. Functionality can be expressed by saying that given a typing relation, if two typing derivations exist for some fixed arguments (seen as inputs) then the other arguments (seen as outputs) are equal to each other.

▶ **Definition 32.** We say that a relation R of type $\Pi(ri : RI).II \to O(ri) \to Set$ is **functional** if for all *r*elevant *i*nputs ri, all pairs of *i*rrelevant *i*nputs ii_1 and ii_2 and for all pairs of *o*utputs o_1 and o_2, if both $R(ri, ii_1, o_1)$ and $R(ri, ii_2, o_2)$ hold then $o_1 \equiv o_2$.

▶ **Lemma 33.** *The Typing Relations for VAR and INFER are functional when seen as relations with relevant inputs the context and the scrutinee (either a VAR or an INFER), irrelevant inputs their USAGE annotation and outputs the inferred TYPEs.*

▶ **Lemma 34.** *The Typing Relations for VAR, INFER, CHECK and ENV are functional when seen as relations with relevant inputs all of their arguments except for one of the USAGE annotation or the other. This means that given a USAGE annotation (whether the input one or the output one) and the rest of the arguments, the other USAGE annotation is uniquely determined.*

8 Typechecking

▶ **Theorem 35** (Decidability of Typechecking)**.** ***Type-inference*** *for INFER and* ***Type-checking*** *for CHECK are decidable. In other words, given a NAT k, γ a CONTEXT$_k$ and Γ a USAGE$_\gamma$,*

1. *for all INFER$_k$ t, we can decide if there is a TYPE σ and Δ a USAGE$_\gamma$ such that $\Gamma \vdash t \in \sigma \boxtimes \Delta$*
2. *for all TYPE σ and CHECK$_k$ t, we can decide if there is Δ a USAGE$_\gamma$ such that $\Gamma \vdash \sigma \ni t \boxtimes \Delta$.*

Proof. The proof proceeds by mutual induction on the raw terms, using inversion lemmas to dismiss the impossible cases, using auxiliary lemmas showing that typechecking of VARs and PATTERNs also is decidable and relies heavily on the functionality of the various relations involved. ◀

One of the benefits of having a formal proof of a theorem in Agda is that the theorem actually has computational content and may be run: the proof is a decision procedure.

▶ **Example 36.** We can for instance check that the search procedure succeeds in finding the `swap-Typed` derivation we had written down as Example 13. Because σ and τ are abstract in the following snippet, the equality test checking that σ is equal to itself and so is τ does not reduce and we need to rewrite by the proof `eq-diag` that the equality test always succeeds in this kind of situation:

```
swapChecked : ∀ σ τ → check [] ((σ ⊗ τ) ⊸ (τ ⊗ σ)) swap
                    ≡ yes ([] , swapTyped)
swapChecked σ τ rewrite eq-diag τ | eq-diag σ = refl
```

9 Equivalence to ILL

We have now demonstrated that the USAGE-based formulation of linear logic as a type system is amenable to mechanisation without putting an unreasonable burden on the user. Indeed, the system's important properties can all be dealt with by structural induction and the user still retains the ability to write simple λ-terms which are not cluttered with structural rules.

However this presentation departs quite a lot from more traditional formulations of intuitionistic linear logic. This naturally raises the question of its correctness. In this section we recall a typical presentation of Intuitionistic Linear Logic using a Sequent Calculus, representing the multiset of assumptions as a list.

9.1 A Sequent Calculus for Intuitionistic Linear Logic

The definition of this calculus is directly taken from the Linear Logic Wiki [26] whose notations we follow to the letter. The interested reader will find more details in for instance Troelstra's lectures [35]. In the following figure, γ, δ, and θ are context variables while σ, τ , and ν range over types. We overload the comma to mean both consing a single type to the front of a list and appending two lists, as is customary.

$$\frac{}{\sigma \vdash \sigma}\,ax \qquad \frac{\gamma \vdash \sigma \qquad \sigma, \delta \vdash \tau}{\gamma, \delta \vdash \tau}\,cut \qquad \frac{\gamma \vdash \sigma \qquad \delta \vdash \tau}{\gamma, \delta \vdash \sigma \otimes \tau}\,\otimes^R \qquad \frac{\tau, \sigma, \gamma \vdash \nu}{\sigma \otimes \tau, \gamma \vdash}\,\otimes^L \qquad \frac{}{\vdash \mathbb{1}}\,1^R$$

$$\frac{\gamma \vdash \sigma}{\mathbb{1}, \gamma \vdash \sigma}\,1^L \qquad \frac{}{\mathbb{0}, \gamma \vdash \sigma}\,0^L \qquad \frac{\sigma, \gamma \vdash \tau}{\gamma \vdash \sigma \multimap \tau}\,\multimap^R \qquad \frac{\gamma \vdash \sigma \qquad \tau, \delta \vdash \nu}{(\sigma \multimap \tau), \gamma, \delta \vdash \nu}\,\multimap^L$$

$$\frac{\gamma \vdash \sigma \qquad \gamma \vdash \tau}{\gamma \vdash \sigma \& \tau}\,\&^R \qquad \frac{\sigma, \gamma \vdash \nu}{\sigma \& \tau, \gamma \vdash \nu}\,\&^L_1 \qquad \frac{\tau, \gamma \vdash \nu}{\sigma \& \tau, \gamma \vdash \nu}\,\&^L_2 \qquad \frac{\gamma \vdash \sigma}{\gamma \vdash \sigma \oplus \tau}\,\oplus^R_1 \qquad \frac{\gamma \vdash \tau}{\gamma \vdash \sigma \oplus \tau}\,\oplus^R_2$$

$$\frac{\sigma, \gamma \vdash \nu \qquad \tau, \gamma \vdash \nu}{\sigma \oplus \tau, \gamma \vdash \nu}\,\oplus^L \qquad \frac{\gamma, \delta \vdash \sigma \qquad \gamma, \delta \cong \theta}{\theta \vdash \sigma}\,mix$$

Figure 10 Sequent Calculus for Intuitionistic Linear Logic

Our only departure from the traditional presentation is the *mix* rule which is an artefact of our encoding multisets as lists. It allows the user to pick any interleaving θ of two lists γ and δ. This notion of interleaving is formalised by the following three-place relation.

▶ **Definition 37.** The **interleaving** relation is defined by three constructors: **[]** declares that interleaving two empty lists yields the empty-list whilst $\cdot,_l \cdot$ (and $\cdot,_r \cdot$ respectively) picks the head of the list on the left (the right respectively) as the head of the interleaving and the tail as the result of interleaving the rest.

$$\frac{\gamma, \delta, \theta \,:\, \text{LIST}\, a}{\gamma, \delta \cong \theta \,:\, \mathit{Set}} \qquad \frac{}{[] \,:\, [], [] \cong []} \qquad \frac{p \,:\, \gamma, \delta \cong \theta}{\sigma,_l\, p \,:\, (\sigma, \gamma), \delta \cong (\sigma, \theta)} \qquad \frac{p \,:\, \gamma, \delta \cong \theta}{\sigma,_r\, p \,:\, \gamma, (\sigma, \delta) \cong (\sigma, \theta)}$$

Now that we have our definition of the usual representation of Intuitionistic Linear Logic (ILL), we are left with proving that the linear typing relation we have defined is both sound and complete with respect to that logic.

9.2 Soundness

We start with the easiest part of the proof: soundness. This means that from a typing derivation, we can derive a proof in ILL of what is essentially the same statement. That is to say that if a term is said to have type σ in a fully fresh context γ and proceeds to consume all of the resources in that context during the typing derivation then it corresponds to a proof of $\gamma \vdash \sigma$ in ILL.

This statement needs to be generalised to be proven. Indeed, even if we start with a context full of available resources, at the first split we encounter (e.g. a tensor introduction or a function application), it won't be the case anymore in one of the sub-terms. To state this more general formulation, we need to introduce a new notion: used assumptions.

▶ **Definition 38.** The list of **used** assumptions in a proof $\Gamma \subseteq \Delta$ in the consumption partial order is the list of types which have turned from fresh in Γ to stale in Δ. The used$(\cdot)$ function is defined by recursion over the proof that $\Gamma \subseteq \Delta$.

▶ **Definition 39.** A Typing Relation $\mathcal{T}$ for terms T is said to be **sound** with respect to ILL if, for k a NAT, γ a CONTEXT$_k$, Γ and Δ two USAGE$_\gamma$, t a term T_k and σ a type, from the typing derivation $\mathcal{T}(\Gamma, \mathrm{t}, \sigma, \Delta)$ and p a proof that $\Gamma \subseteq \Delta$ we can derive used$(p) \vdash \sigma$.

▶ Remark. The consumption lemma 18 guarantees that such a proof $\Gamma \subseteq \Delta$ always exists whenever the typing relation is either the one for VAR, INFER or CHECK.

Before we can prove the soudness theorem, we need two auxiliary lemmas allowing us to handle the mismatch we may have between the way the used assumptions of a derivation are computed and the way the ones for its subderivations are obtained.

▶ **Lemma 40.** *Given k a NAT, γ a CONTEXT$_k$ and Γ, Δ, and θ three USAGE$_\gamma$, we have:*

1. *if p and q are proofs that $\Gamma \subseteq \Delta$ then used(p) = used(q).*
2. *if p is a proof that $\Gamma \subseteq \Delta$, q that $\Delta \subseteq \theta$ and pq that $\Gamma \subseteq \theta$ then used(pq) is an interleaving of used(p) and used(q).*

The relation validating patterns is not a typing relation and as such it needs to be handled separately. This can be done by defining a procedure elaborating patterns away by showing that whenever $\sigma \ni p \rightsquigarrow \gamma$, it is morally acceptable to replace σ on the left by γ. Which gives us the following *cut*-like admissible rule:

▶ **Lemma 41** (Elaboration of Let-bindings)**.** *Provided k a NAT, p a PATTERN$_k$, σ a TYPE and γ a CONTEXT$_k$ such that $\sigma \ni p \rightsquigarrow \gamma$, we have that for all δ and θ two LIST TYPE and τ a TYPE, if $\delta \vdash \sigma$ and $\gamma, \theta \vdash \tau$ then $\delta, \theta \vdash \tau$.*

We now have all the pieces to prove the soundness of our typing relations.

▶ **Theorem 42** (Soundness)**.** *The typing relations for VAR, INFER and CHECK are all sound.*

Proof. The proof is by mutual induction on the typing derivations. ILL's right rules are in direct correspondence with our introduction rules. The eliminators in our languages are translated by using ILL's *cut* together with left rules. The *mix* rule is crucial to rewrite the derivations' contexts on the fly. ◀

9.3 Completeness

Completeness is a trickier thing to prove: given a derivation in the traditional sequent calculus, we need to build a corresponding term and its typing derivation. However, unlike the soundness one it does not give us any insight as to what the meaning of USAGE and typing derivations is. So we only state the result and give an idea of the proof.

▶ **Theorem 43** (Completeness)**.** *Given γ a LIST TYPE and σ a type, from a proof $\gamma \vdash \sigma$ we can derive an INFER t and a proof that $\mathtt{f}_\gamma \vdash t \in \sigma \boxtimes \mathtt{s}_\gamma$.*

Proof. The proof is by induction over the derivation in ILL. The right rules match our introduction rules well enough that they do not pose any issue. Weakening lets us extend the USAGE of the sequents obtained by induction hypothesis in the case of multi-premises rules. Left rules are systematically translated as `cut` s.

Finally, the hardest rule to handle is the `mix` rule which reorganises the context. It is handled by a technical lemma which we have left out of this paper. Informally: it states that if the input and output USAGE of a typing derivation are obtained by the same interleaving of two distinct pairs of USAGE, then for any other interleaving we can find a term and a typing derivation for that term. ◀

10 Related Work

Benton, Bierman, de Paiva, and Hyland [8] did devise a term assignment system for Intuitionistic Linear Logic which was stable under substitution. Their system focuses on multiplicative linear logic only when ours also encompasses additive connectives *but* it gives a thorough treatment of the *!* modality. This is still an open problem for us because we do not want to have the explicit handling of *!*'s weakening, contraction, dereliction, and promotion rules pollute the raw terms.

Rand, Paykin and Zdancewic's work on modelling quantum circuits in Coq [34] necessarily includes a treatment of linearity as qbits cannot be duplicated. And because it is mechanised, they have to deal with the representation of contexts. Their focus is mostly on the quantum aspect and they are happy relying on Coq's scripting capabilities to cope with the extensional presentation.

Bob Atkey and James Wood [6] have been experimenting with using a deep embedding of a linear lambda calculus in Agda as a way to certify common algorithms. Being able to encode insertion sort as a term in this deep embedding is indeed sufficient to conclude that the output of the algorithm is a permutation of its input. They use on-the-fly re-ordering of contexts via explicit permutation proofs.

Polakow faced with the task of embedding a linear λ-calculus in Haskell [33] used a typed-tagless approach [23] and tried to get as much automation from typeclass resolution as possible. Seeing Haskell's typeclass resolution mechanism as a Prolog-style proof search engine, he opted for a relational description and thus an input-output presentation. This system can handle multiplicatives, additives and is even extended to a Dual Intuitionistic Linear Logic [7] to accomodate for values which can be duplicated. Focusing on the applications, it is not proven to be stable under substitution or that the typechecking process will always succeed.

The proof search community has been confronted with the inefficiency of randomly splitting up the multiset of assumption when applying a tensor-introduction rule. In an effort to combat this non-determinism, they have introduced alternative sequent calculi returning leftovers [14, 37]. However because they do not have to type a term living in a given context, they do not care about the structure of the context of assumptions: it is still modelled as a multiset.

We have already mentioned McBride's work [29] on (as a first approximation: the setup is actually more general) a type theory with a *dependent linear* function space as a very important source of inspiration. In that context it is indeed crucial to retain the ability to talk about a resource even if it has already been consumed. E.g. a function taking a boolean and deciding whether it is equal to `tt` or `ff` will have a type mentioning the function's argument twice. But in a lawful manner: $(x : \text{BOOL}) \multimap (x \equiv \texttt{tt}) \vee (x \equiv \texttt{ff})$. This leads to the need for a context *shared* across all subterms and consumption annotations ensuring that the linear resources are never used more than once.

Finally, we can find a very concrete motivation for a predicate similar to our USAGE in Robbert Krebbers' thesis [24]. In section 2.5.9, he describes one source of undefined behaviours in the C standard: the execution order of expressions is unspecified thus leaving the implementers with absolute freedom to pick any order they like if that yields better performances. To make their life simpler, the standard specifies that no object should be modified more than once during the execution of an expression. In order to enforce this invariant, Krebbers' memory model is enriched with extra information:

> [E]ach bit in memory carries a permission that is set to a special locked permission when a store has been performed. The memory model prohibits any access (read or store) to objects with locked permissions. At the next sequence point, the permissions of locked objects are changed back into their original permission, making future accesses possible again.

11 Conclusion

We have shown that taking seriously the view of linear logic as a logic of resource consumption leads, in type theory, to a well-behaved presentation of the corresponding type system for the lambda-calculus. The framing property claims that the state of irrelevant resources does not matter, stability under weakening shows that one may even add extra irrelevant assumptions to the context and they will be ignored whilst stability under substitution guarantees subject reduction with respect to the usual small step semantics of the lambda calculus. Finally, the decidability of type checking makes it possible to envision a user-facing language based on raw terms and top-level type annotations where the machine does the heavy lifting of checking that all the invariants are met whilst producing a certified-correct witness of typability.

Avenues for future work include a treatment of an *affine* logic where the type of substitution will have to be be different because of the ability to throw away resources without using them. Our long term goal is to have a formal specification of a calculus for Probabilistic and Bayesian Reasoning similar to the affine one described by Adams and Jacobs [2]. Another interesting question is whether these resource annotations can be used to develop a fully formalised proof search procedure for intuitionistic linear logic. The author and McBride have made an effort in such a direction [3] by designing a sound and complete search procedure for a fragment of intuitionistic linear logic with type constructors tensor and with. Its extension to lolipop is currently an open question.

References

1 Peter Achten, John Van Groningen, and Rinus Plasmeijer. High level specification of I/O in functional languages. In *Functional Programming, Glasgow 1992*, pages 1–17. Springer, 1993.

2 Robin Adams and Bart Jacobs. A type theory for probabilistic and bayesian reasoning. November 2015. URL: `http://arxiv.org/abs/1511.09230`.

3 Guillaume Allais and Conor McBride. Certified proof search for intuitionistic linear logic. 2015. URL: `http://gallais.github.io/proof-search-ILLWiL/`.

4 Thorsten Altenkirch, Martin Hofmann, and Thomas Streicher. Categorical reconstruction of a reduction free normalization proof. In *Category Theory and Computer Science*, pages 182–199. Springer, 1995.

5 Thorsten Altenkirch and Bernhard Reus. Monadic presentations of lambda terms using generalized inductive types. In *Computer Science Logic*, pages 453–468. Springer, 1999.

6 Robert Atkey and James Wood. Sorting types – permutation via linearity. `https://github.com/bobatkey/sorting-types`, 2013. Retrieved on 2017-11-06.

7 Andrew Barber and Gordon Plotkin. *Dual intuitionistic linear logic*. University of Edinburgh, Department of Computer Science, Laboratory for Foundations of Computer Science, 1996.

8 Nick Benton, Gavin Bierman, Valeria De Paiva, and Martin Hyland. A term calculus for intuitionistic linear logic. *Typed Lambda Calculi and Applications*, pages 75–90, 1993.

9 Richard S. Bird and Ross Paterson. de Bruijn notation as a nested datatype. *Journal of Functional Programming*, 9(1):77–91, 1999.

10 Samuel Boutin. Using reflection to build efficient and certified decision procedures. In *Theoretical Aspects of Computer Software*, pages 515–529. Springer, 1997.

11 Edwin Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming*, 23(05):552–593, 2013.

12 Edwin Brady, Conor McBride, and James McKinna. Inductive families need not store their indices. In *Types for proofs and programs*, pages 115–129. Springer, 2003.

13 Edwin Brady and Matúš Tejiščák. Practical erasure in dependently typed languages.

14 Iliano Cervesato, Joshua S Hodas, and Frank Pfenning. Efficient resource management for linear logic proof search. In *International Workshop on Extensions of Logic Programming*, pages 67–81. Springer, 1996.

15 James Maitland Chapman. *Type checking and normalisation*. PhD thesis, University of Nottingham, 2009.

16 Pierre-Evariste Dagand and Conor McBride. Transporting functions across ornaments. *Journal of Functional Programming*, 24(2-3):316–383, 2014.

17 Nils Danielsson. Bag equivalence via a proof-relevant membership relation. *Interactive Theorem Proving*, pages 149–165, 2012.

18 Olivier Danvy. *Type-Directed Partial Evaluation*. Springer, 1999.

19 Nicolaas Govert De Bruijn. Lambda calculus notation with nameless dummies. In *Indagationes Mathematicae (Proceedings)*, volume 75, pages 381–392. Elsevier, 1972.

20 The Rust Project Developers. The Rust Programming Language – Ownership. `https://doc.rust-lang.org/book/first-edition/ownership.html`, 2017. Retrieved on 2017-11-06.

21 Peter Dybjer. Inductive families. *Formal aspects of computing*, 6(4):440–465, 1994.

22 Jean-Yves Girard. Linear logic. *Theoretical computer science*, 50(1):1–101, 1987.

23 Oleg Kiselyov. Typed tagless final interpreters. In *Generic and Indexed Programming*, pages 130–174. Springer, 2012.

24 Robbert Krebbers. *The C standard formalized in Coq*. PhD thesis, Radboud University Nijmegen, 2015.

25 John Launchbury and Simon L Peyton Jones. Lazy functional state threads. In *ACM SIGPLAN Notices*, volume 29, pages 24–35. ACM, 1994.

26 Olivier Laurent and Laurent Regnier. Linear Logic Wiki – Intuitionistic Linear Logic. `http://llwiki.ens-lyon.fr/mediawiki/index.php/Intuitionistic_linear_logic`, 2009. Retrieved on 2017-11-06.

27 Pierre Letouzey. A new extraction for coq. In *Types for proofs and programs*, pages 200–219. Springer, 2002.

28 Conor McBride. Ornamental algebras, algebraic ornaments. *Journal of functional programming*, 2010.

29 Conor McBride. I got plenty o'nuttin'. In *A List of Successes That Can Change the World*, pages 207–233. Springer, 2016.

30 John C Mitchell. *Foundations for programming languages*, volume 1. MIT press, 1996.

31 Ulf Norell. Dependently typed programming in Agda. In *Advanced Functional Programming*, pages 230–266. Springer, 2009.

32 Benjamin C Pierce and David N Turner. Local type inference. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 22(1):1–44, 2000.

33 Jeff Polakow. Embedding a full linear lambda calculus in haskell. *ACM SIGPLAN Notices*, 50(12):177–188, 2016.

34 Robert Rand, Jennifer Paykin, and Steve Zdancewic. Qwire practice: Formal verification of quantum circuits in coq. In *Quantum Physics and Logic*, 2017.

35 Anne Sjerp Troelstra. *Lectures on linear logic*. 1991.

36 Stephanie Weirich, Justin Hsu, and Richard A Eisenberg. Towards dependently typed haskell: System FC with kind equality. Citeseer, 2013.

37 Michael Winiko and James Harland. Deterministic resource management for the linear logic programming language lygon. Technical report, Technical Report 94/23, Melbourne University, 1994.

A Fully-expanded Typing Derivation for `swap`

$$\dfrac{\dfrac{\dfrac{}{[\,] \bullet \mathtt{f}_{\sigma\otimes\tau} \vdash_{v} 0 \in \sigma \otimes \tau \boxtimes [\,] \bullet \mathtt{s}_{\sigma\otimes\tau}}}{[\,] \bullet \mathtt{f}_{\sigma\otimes\tau} \vdash \mathtt{var}(0) \in \sigma \otimes \tau \boxtimes [\,] \bullet \mathtt{s}_{\sigma\otimes\tau}} \qquad \dfrac{\dfrac{}{\sigma \ni \mathtt{v} \rightsquigarrow [\,] \bullet \sigma} \quad \dfrac{}{\tau \ni \mathtt{v} \rightsquigarrow [\,] \bullet \tau}}{\sigma \otimes \tau \ni (\mathtt{v},\ \mathtt{v}) \rightsquigarrow [\,] \bullet \tau \bullet \sigma} \qquad \Pi}{\dfrac{[\,] \bullet \mathtt{f}_{\sigma\otimes\tau} \vdash \tau \otimes \sigma \ni \begin{array}{l}\mathtt{let}\ (\mathtt{v},\ \mathtt{v})\ \mathtt{::=}\ \mathtt{var}\ 0\ \mathtt{in} \\ \mathtt{prd}(\mathtt{neu}(\mathtt{var}(1)),\ \mathtt{neu}(\mathtt{var}(0)))\end{array} \boxtimes [\,] \bullet \mathtt{s}_{\sigma\otimes\tau}}{[\,] \vdash (\sigma \otimes \tau) \multimap (\tau \otimes \sigma) \ni \mathtt{swap} \boxtimes [\,]}}$$

$$\Pi = \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{[\,] \bullet \mathtt{s}_{\sigma\otimes\tau} \bullet \mathtt{f}_{\tau} \vdash_{v} 0 \in \tau \boxtimes [\,] \bullet \mathtt{s}_{\sigma\otimes\tau} \bullet \mathtt{s}_{\tau}}}{[\,] \bullet \mathtt{s}_{\sigma\otimes\tau} \bullet \mathtt{f}_{\tau} \bullet \mathtt{f}_{\sigma} \vdash_{v} 1 \in \tau \boxtimes [\,] \bullet \mathtt{s}_{\sigma\otimes\tau} \bullet \mathtt{s}_{\tau} \bullet \mathtt{f}_{\sigma}}}{[\,] \bullet \mathtt{s}_{\sigma\otimes\tau} \bullet \mathtt{f}_{\tau} \bullet \mathtt{f}_{\sigma} \vdash \mathtt{var}(1) \in \tau \boxtimes [\,] \bullet \mathtt{s}_{\sigma\otimes\tau} \bullet \mathtt{s}_{\tau} \bullet \mathtt{f}_{\sigma}}}{[\,] \bullet \mathtt{s}_{\sigma\otimes\tau} \bullet \mathtt{f}_{\tau} \bullet \mathtt{f}_{\sigma} \vdash \tau \ni \mathtt{neu}(\mathtt{var}(1)) \boxtimes [\,] \bullet \mathtt{s}_{\sigma\otimes\tau} \bullet \mathtt{s}_{\tau} \bullet \mathtt{f}_{\sigma}} \qquad \Pi'}{[\,] \bullet \mathtt{s}_{\sigma\otimes\tau} \bullet \mathtt{f}_{\tau} \bullet \mathtt{f}_{\sigma} \vdash \tau \otimes \sigma \ni \mathtt{prd}(\mathtt{neu}(\mathtt{var}(1)),\ \mathtt{neu}(\mathtt{var}(0))) \boxtimes [\,] \bullet \mathtt{s}_{\sigma\otimes\tau} \bullet \mathtt{s}_{\tau} \bullet \mathtt{s}_{\sigma}}$$

$$\Pi' = \dfrac{\dfrac{\dfrac{}{[\,] \bullet \mathtt{s}_{\sigma\otimes\tau} \bullet \mathtt{s}_{\tau} \bullet \mathtt{f}_{\sigma} \vdash_{v} 0 \in \sigma \boxtimes [\,] \bullet \mathtt{s}_{\sigma\otimes\tau} \bullet \mathtt{s}_{\tau} \bullet \mathtt{s}_{\sigma}}}{[\,] \bullet \mathtt{s}_{\sigma\otimes\tau} \bullet \mathtt{s}_{\tau} \bullet \mathtt{f}_{\sigma} \vdash \mathtt{var}(0) \in \sigma \boxtimes [\,] \bullet \mathtt{s}_{\sigma\otimes\tau} \bullet \mathtt{s}_{\tau} \bullet \mathtt{s}_{\sigma}}}{[\,] \bullet \mathtt{s}_{\sigma\otimes\tau} \bullet \mathtt{s}_{\tau} \bullet \mathtt{f}_{\sigma} \vdash \sigma \ni \mathtt{neu}(\mathtt{var}(0)) \boxtimes [\,] \bullet \mathtt{s}_{\sigma\otimes\tau} \bullet \mathtt{s}_{\tau} \bullet \mathtt{s}_{\sigma}}$$