Typing with Leftovers – A mechanization of Intuitionistic Linear Logic*

Guillaume Allais[1]

1 **Nijmegen Quantum Logic Group — Radboud University**
gallais@cs.ru.nl

Abstract

We start from a simple λ-calculus and introduce a bidirectional typing relation corresponding to an Intuitionistic Linear Logic. 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 could 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.

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`.

1998 ACM Subject Classification Dummy classification – please refer to `http://www.acm.org/about/class/ccs98-html`

Keywords and phrases Dummy keyword – please provide 1–5 keywords

Digital Object Identifier 10.4230/LIPIcs.CVIT.2016.23

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 [11] residualising specialised programs to erasure mechanisms –be they user-guided like Coq's extraction [17] which systematically removes all the purely logical proofs put in Prop by the developer or automated like Brady and Tejiščák's erased values [8, 9]– and including the library defining the State-Thread [16] 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 [7, 19] which, unlike ghc's Haskell [21], incorporate a hierarchy of universes in order to make certain 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. Idris has been extended with experimental support for uniqueness types inspired by Clean's [2] and Rust's ownership types [1], all of which stem from linear logic [14].

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.

* This work was partially supported by someone.

Notations

This whole development has been fully formalised in Agda. Rather than including Agda syntax, the results are reformulated in terms of definitions, lemmas, theorems, etc. 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 families' index is written as a subscript e.g. VAR_n.

2 The Calculus of Raw Terms

Following Altenkirch and Reus [6], we define the raw terms of our language not as an inductive type but rather as an inductive *family* [13]. 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 $\bot$ and fresh variables are introduced by wrapping the index in a `Maybe` type constructor[1], we index our terms by a natural number instead. The VAR type family defined below represents the de Bruijn indices [12] corresponding to the $n-1$ free variables present in a scope n.

$$\frac{n : \text{NAT}}{\text{VAR}_n : \mathit{Set}} \qquad \frac{}{\texttt{zero} : \text{VAR}_{1+n}} \qquad \frac{k : \text{VAR}_n}{\texttt{suc}(k) : \text{VAR}_{1+n}}$$

The calculus is presented in a bidirectional fashion [20]. This gives a clean classification of term formers as being either constructors of canonical values or eliminations corresponding to computations. This separation also characterises 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).

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

Figure 1 Grammar of the Language of Raw Terms

Two additional rules (`neu` and `cut` 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. The constructor `cut` takes an extra TYPE argument in order to guarantee the success of type-inference for INFER terms.

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

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 type family. A value of type PATTERN_n represents a 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 1 variable. 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.

$$\frac{n : \text{NAT}}{\text{PATTERN}_n : \mathit{Set}} \qquad \frac{}{\texttt{v} : \text{PATTERN}_1} \qquad \frac{p : \text{PATTERN}_m \qquad q : \text{PATTERN}_n}{p \texttt{,} q : \text{PATTERN}_{m+n}}$$

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 zero
            in prd (neu (var (suc zero))) (neu (var zero)))
```

▶ **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 zero))) (a ⊸ a)
           in prd (neu (var zero)) (neu (var (suc zero)))
```

▶ **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 zero)) (neu (var zero)))
```

3 Linear Typing Rules

These considerations lead us to the need for a typing relation describing the rules terms need to abide by in order to qualify as valid programs. A linear type system is characterised by the fact that all the resources available in a 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 (cf. Figure 2).

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 either extra inference rules to reorganise the context or a side condition to rules splitting the context so that it may be re-arranged on the fly. In the following example $\_ \approx \_$ stands for "bag-equivalence" of lists.

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 tries not 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 simple structural inductions.

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

Figure 2 Introduction rules for tensor (left: usual presentation, right: with reordering on the fly)

3.1 Usage Annotations

McBride's recent work [18] 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 `fresh` and `stale` respectively. The pointwise lifting of USAGE to contexts is called USAGES. The inference rules are:

$$\frac{\sigma : \text{TYPE}}{\text{USAGE}_\sigma : \mathit{Set}} \qquad \frac{}{\texttt{fresh}_\sigma : \text{USAGE}_\sigma} \qquad \frac{}{\texttt{stale}_\sigma : \text{USAGE}_\sigma}$$

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

3.2 Typing as Consumption Annotation

A Typing relation seen as a consumption annotation process describes what it means, given a context an its usage annotation, to ascribe a type to a 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:

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

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.

$$\frac{}{\Gamma \bullet \mathtt{fresh}_\sigma \vdash \mathtt{zero} \in \sigma \boxtimes \Gamma \bullet \mathtt{stale}_\sigma} \qquad \frac{\Gamma \vdash k \in \sigma \boxtimes \Delta}{\Gamma \bullet A \vdash \mathtt{suc}(k) \in \sigma \boxtimes \Delta \bullet A}$$

Figure 3 Typing rules for VAR

$$\frac{\Gamma \vdash k \in \sigma \boxtimes \Delta}{\Gamma \vdash \mathtt{var}(k) \in \sigma \boxtimes \Delta} \qquad \frac{\Gamma \vdash t \in \sigma \multimap \tau \boxtimes \Delta \qquad \Delta \vdash \sigma \ni u \boxtimes \Theta}{\Gamma \vdash \mathtt{app}(t,u) \in \tau \boxtimes \Theta}$$

$$\frac{\Gamma \vdash t \in \sigma \oplus \tau \boxtimes \Delta \qquad \begin{array}{l}\Delta \bullet \mathtt{fresh}_\sigma \vdash \nu \ni l \boxtimes \Theta \bullet \mathtt{stale}_\sigma \\ \Delta \bullet \mathtt{fresh}_\tau \vdash \nu \ni r \boxtimes \Theta \bullet \mathtt{stale}_\tau\end{array}}{\Gamma \vdash \mathtt{case}\ t\ \mathtt{return}\ \nu\ \mathtt{of}\ l\ \%\%\ r \in \nu \boxtimes \Theta} \qquad \frac{\Gamma \vdash \sigma \ni t \boxtimes \Delta}{\Gamma \vdash \mathtt{cut}(t,\sigma) \in \sigma \boxtimes \Delta}$$

Figure 4 Typing rules for INFER

$$\frac{\Gamma \bullet \mathtt{fresh}_\sigma \vdash \tau \ni b \boxtimes \Delta \bullet \mathtt{stale}_\sigma}{\Gamma \vdash \sigma \multimap \tau \ni \mathtt{lam}(b) \boxtimes \Delta} \qquad \frac{\Gamma \vdash t \in \sigma \boxtimes \Delta \qquad \sigma \ni p \rightsquigarrow \delta \qquad \Delta +\!\!+ \mathtt{fresh}_\delta \vdash \tau \ni u \boxtimes \Theta +\!\!+ \mathtt{stale}_\delta}{\Gamma \vdash \tau \ni \mathtt{let}\ p ::= t\ \mathtt{in}\ u \boxtimes \Theta}$$

$$\frac{\Gamma \vdash \sigma \ni t \boxtimes \Delta}{\Gamma \vdash \sigma \oplus \tau \ni \mathtt{inl}(t) \boxtimes \Delta} \qquad \frac{\Gamma \vdash \tau \ni t \boxtimes \Delta}{\Gamma \vdash \sigma \oplus \tau \ni \mathtt{inr}(t) \boxtimes \Delta}$$

$$\frac{\Gamma \vdash \sigma \ni a \boxtimes \Delta \qquad \Delta \vdash \tau \ni b \boxtimes \Theta}{\Gamma \vdash \sigma \otimes \tau \ni \mathtt{prd}(a,b) \boxtimes \Theta} \qquad \frac{\Gamma \vdash t \in \sigma \boxtimes \Delta}{\Gamma \vdash \sigma \ni \mathtt{neu}(t) \boxtimes \Delta}$$

Figure 5 Typing rules for CHECK

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

Figure 6 Typing rules for PATTERN

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 to that index if and only if the kth resource in context is `fresh`. In the resulting leftovers, the resource will have turned `stale`:

▶ **Definition 7.** The typing relation is presented in a sequent-style: $\Gamma \vdash k \in \sigma \boxtimes \Delta$ 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 3).

▶ 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 simply be given the same name as their VAR counterpart. 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 $(\gamma \bullet \sigma \bullet \tau)$ with usage annotation $(\Gamma \bullet \texttt{fresh}_\tau \bullet \texttt{fresh}_\sigma)$:

$$\frac{\overline{\Gamma \bullet \texttt{fresh}_\tau \vdash \texttt{zero} \in \tau \boxtimes \Gamma \bullet \texttt{stale}_\tau}}{\Gamma \bullet \texttt{fresh}_\tau \bullet \texttt{fresh}_\sigma \vdash \texttt{suc(zero)} \in \tau \boxtimes \Gamma \bullet \texttt{stale}_\tau \bullet \texttt{fresh}_\sigma}$$

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 • fresh τ • fresh σ ⊢ suc(zero) ∈ τ ⊠ Γ • stale τ • fresh σ
one = suc zero
```

3.2.2 Typing Terms

The key idea appearing in all the typing rules for compound expressions is to use the input USAGES to type one of the sub-expressions, collect the leftovers from that typing derivation and use them as the new input USAGES 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 USAGES 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. 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. Indeed, in both cases the type is inferred. $\Gamma \vdash t \in \sigma \boxtimes \Delta$ means that given Γ a USAGES_γ, and t an INFER, the type σ is inferred together with leftovers Δ, another USAGES_γ. The rules are listed in Figure 4.

For CHECK, the type σ comes first: $\Gamma \vdash \sigma \ni t \boxtimes \Delta$ means that given Γ a USAGES_γ, a type σ, the CHECK t can be checked to have type σ with leftovers Δ. The rules can be found in Figure 5.

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 obviously 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 6.

▶ **Example 10.** Given these rules, it is easy to see that the identity function can be checked at type $(\sigma \multimap \sigma)$ in an empty context:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{[] \bullet \texttt{fresh}_\sigma \vdash \texttt{zero} \in \sigma \boxtimes [] \bullet \texttt{stale}_\sigma}}{[] \bullet \texttt{fresh}_\sigma \vdash \texttt{var(zero)} \in \sigma \boxtimes [] \bullet \texttt{stale}_\sigma}}{[] \bullet \texttt{fresh}_\sigma \vdash \sigma \ni \texttt{neu(var(zero))} \boxtimes [] \bullet \texttt{stale}_\sigma}}{[] \vdash \sigma \multimap \sigma \ni \texttt{lam(neu(var(zero)))} \boxtimes []}$$

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 zero)) ⊠ []
identity = lam (neu (var zero))
```

▶ **Example 11.** It is also possible to revisit Example 1 to prove that it 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 quite a lot more readable:

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

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 usages 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 12.** A Typing Relation $\mathcal{T}$ for a NAT-indexed family T has the Framing Property if for all k a NAT, γ a CONTEXT$_k$, $\Gamma, \Delta, \Theta, \xi$ four USAGES$_\gamma$, t an element of T_k and σ a Type, if $\Gamma - \Delta \equiv \Theta - \xi$ and $\mathcal{T}(\Gamma, t, \sigma, \Delta)$ then $\mathcal{T}(\Theta, t, \sigma, \xi)$ also holds.

▶ **Definition 13.** The "consumption equivalence" characterises the pairs of an input and an output USAGES 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.

$$\dfrac{\Gamma, \Delta, \Theta, \xi : \text{USAGES}_\gamma}{\Gamma - \Delta \equiv \Theta - \xi : \mathit{Set}} \qquad \dfrac{}{[] - [] \equiv [] - []} \qquad \dfrac{\Gamma - \Delta \equiv \Theta - \xi}{(\Gamma \bullet A) - (\Delta \bullet A) \equiv (\Theta \bullet B) - (\xi \bullet B)}$$

$$\dfrac{\Gamma - \Delta \equiv \Theta - \xi}{(\Gamma \bullet \texttt{fresh}_\sigma) - (\Delta \bullet \texttt{stale}_\sigma) \equiv (\Theta \bullet \texttt{fresh}_\sigma) - (\xi \bullet \texttt{stale}_\sigma)}$$

▶ **Definition 14.** The "consumption partial order" $\Gamma \subseteq \Delta$ is defined as $\Gamma - \Delta \equiv \Gamma - \Delta$.

▶ **Lemma 15.** *1. The consumption equivalence is a partial equivalence*
2. The consumption partial order is a partial order
3. If there is a Usages "in between" two others according to the consumption partial order, then any pair of usages equal to these two can be split in a manner compatible with this middle element. Formally: Given Γ, Δ, Θ, ξ and χ such that $\Gamma - \Delta \equiv \Theta - \xi$, $\Gamma \subseteq \chi$ and $\chi \subseteq \Delta$, one can find ζ such that: $\Gamma - \chi \equiv \Theta - \zeta$ and $\chi - \Delta \equiv \zeta - \xi$.

▶ **Lemma 16** (Consumption). *The Typing Relations for VAR, INFER and CHECK all imply that if a typing derivation exists with input usages annotation Γ and output usages annotation Δ then $\Gamma \subseteq \Delta$.*

▶ **Theorem 17.** *The Typing Relations 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 generate the inclusion evidence and use it to split up the witness of consumption equivalence and distribute it appropriately in the induction hypotheses. ◀

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, in order to define substitution for instance, to be able to extend the underlying context a term is defined with respect to. 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 [10, 5]. 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. In this development, three variations on OPEs are actually needed: one for NATs to describe the embedding of a scope into a larger one, one for CONTEXT describing what types the extra variables in scope are assigned and finally one for their USAGES mentioning whether these variables are fresh or stale.

▶ **Definition 18.** An Order Preserving Embedding is an inductive family. Its constructors dubbed "moves" describe a strategy to realise the promise of an embedding. Figure 7 defines three OPEs at the same time.

The first column lists the names of the constructors associated to each one of the moves. The second column describes the OPE for NATs and it follows closely the traditional definition of OPEs. The two remaining columns are a bit different: they respectively define the OPE for CONTEXTs and the one for USAGES. However they do not mention the source and target sets in their indices; they are in fact generic enough to be applied to any context / usages of the right size. If a value of $k \leq l$ is seen as a *diff* between k and l then the OPEs on contexts and usages annotations only specify what to do for the variables introduced by the diff (i.e. the variables corresponding to an `insert` constructor). These diffs can then be applied to concrete contexts and usages respectively to transform them.

The first row defines the move `done` for each one of the OPEs. It is the strategy corresponding to the trivial embedding of a set into itself by the identity function. It 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 USAGES are purely structural: no additional content is required to be able to perform a `copy` move.

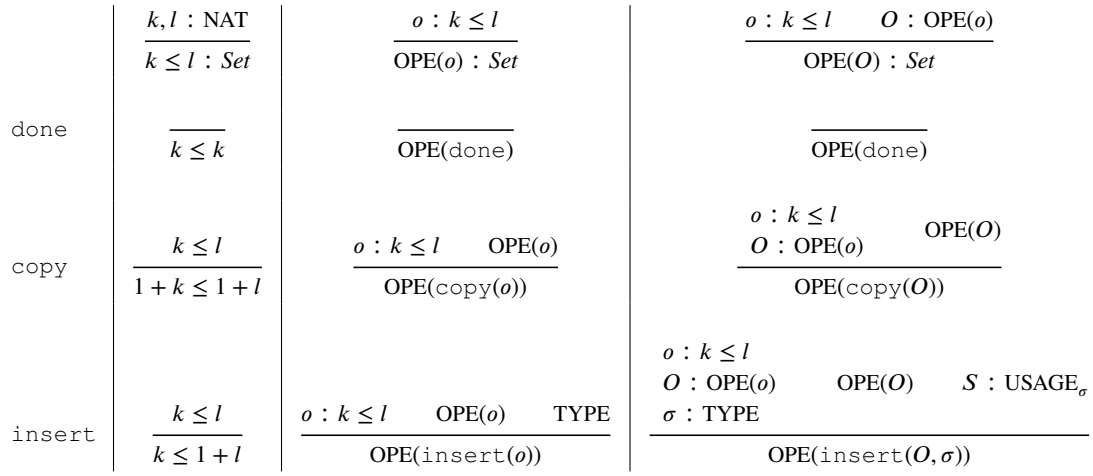

Figure 7 Order Preserving Embeddings for NAT, CONTEXT and USAGES

Last but not least, 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 USAGES takes a USAGE_σ (it will be the usage associated to that newly introduced variable of type σ).

▶ **Example 19.** We may define three embeddings corresponding to the introduction of a fresh variable for scopes, contexts and usages respectively. The body of these three declarations looks the same because we overload the constructors' names but they build different objects. As can be seen in the types, the latter ones depend on the former ones. This type of embedding is very much grounded in reality: it is precisely what pushing a substitution under a lambda abstraction calls for.

```
scopeWithFV : n ≤ suc n
scopeWithFV = insert done

contextWithFV : Type → Context.OPE scopeWithFV
contextWithFV σ = insert done σ

usagesWithFV : (σ : Type) → Usage σ → Usages.OPE (contextWithFV σ)
usagesWithFV σ S = insert done S
```

We leave out the definitions of the two `patch` functions applying the diff described by an OPE to respectively a context or a usages annotation. They are structural in the OPE. The interested reader will find them in the formal development in Agda. We also leave out the definition of weakening for raw terms. It is given by a simple structural induction on the terms themselves.

▶ **Definition 20.** 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 OPE(o) and $\mathcal{O}$ a proof that OPE(O) then for all γ a CONTEXT_k, Γ and Δ two USAGES_γ, t an element of T_k and σ a TYPE, if $\mathcal{T}(\Gamma, t, \sigma, \Delta)$ holds true then we also have $\mathcal{T}(\texttt{patch}(\mathcal{O}, \Gamma), \texttt{weak}_T(o, t), \sigma, \texttt{patch}(\mathcal{O}, \Delta))$.

▶ **Theorem 21.** *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. ◀

6 Substituting

Stability of the typing relation under substitution guarantees that the 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.

▶ **Definition 22.** 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 ($\bullet$`v`) and one to extend an environment with an INFER_l.

$$\frac{}{[] : \text{ENV}(0, l)} \qquad \frac{\rho : \text{ENV}(k, l)}{\rho \bullet \texttt{v} : \text{ENV}(\texttt{suc}(k), \texttt{suc}(l))} \qquad \frac{\rho : \text{ENV}(k, l) \qquad t : \text{INFER}_l}{\rho \bullet t : \text{ENV}(\texttt{suc}(k), l)}$$

Environment are carrying INFER elements because, being in the same syntactical class as VARs, they can be substituted for them without any issue.

▶ **Lemma 23.** *Raw terms are stable under substitutions: for all k and l, given t a term INFER_k (resp. CHECK_k) and ρ an environment $\text{ENV}(k, l)$, we can apply the substitution ρ to t and obtain an INFER_l (resp. CHECK_l).*

Proof. By mutual induction on the raw terms, using the $\bullet$`v` ENV constructor when going under a binder. The need for weakening or crafting fresh variables has not disappeared, it has been transferred to the auxiliary function looking up a value in ρ given a VAR_k. ◀

▶ **Definition 24.** The environments used when proving that Typing Relations are stable under substitution follow closely the ones for raw terms. $\text{ENV}(\Theta_1, \rho, \Theta_2, \Gamma)$ is a typing relation with input usages Θ_1 and output Θ_2 for the raw substitution ρ targeting the `fresh` variables in Γ. Unsurprisingly, the typing for the empty environment has the same input and output usages annotation. Formally:

$$\frac{\begin{array}{l}\theta : \text{CONTEXT}_l \\ \Theta_1, \Theta_2 : \text{USAGES}_\theta\end{array} \qquad \rho : \text{ENV}(k, l) \qquad \begin{array}{l}\gamma : \mathit{Context}_k \\ \Gamma : \text{USAGES}_\gamma\end{array}}{\text{ENV}(\Theta_1, \rho, \Theta_2, \Gamma) : \mathit{Set}} \qquad \frac{}{\text{ENV}(\Theta_1, [], \Theta_1, [])}$$

For `fresh` variables in Γ, there are two cases depending 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 the variable has been consumed.

$$\frac{\Theta_1 \vdash t \in \sigma \boxtimes \Theta_2 \qquad \mathrm{ENV}(\Theta_2, \rho, \Theta_3, \Gamma)}{\mathrm{ENV}(\Theta_1, \rho \bullet t, \Theta_3, \Gamma \bullet \mathtt{fresh}_\sigma)} \qquad \frac{\mathrm{ENV}(\Theta_1, \rho, \Theta_2, \Gamma)}{\mathrm{ENV}(\Theta_1 \bullet \mathtt{fresh}_\sigma, \rho \bullet \mathtt{v}, \Theta_2 \bullet \mathtt{stale}_\sigma, \Gamma \bullet \mathtt{fresh}_\sigma)}$$

For `stale` 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.

$$\frac{\mathrm{ENV}(\Theta_1, \rho, \Theta_2, \Gamma)}{\mathrm{ENV}(\Theta_1, \rho \bullet t, \Theta_2, \Gamma \bullet \mathtt{stale}_\sigma)} \qquad \frac{\mathrm{ENV}(\Theta_1, \rho, \Theta_2, \Gamma)}{\mathrm{ENV}(\Theta_1 \bullet \mathtt{stale}_\sigma, \rho \bullet \mathtt{v}, \Theta_2 \bullet \mathtt{stale}_\sigma, \Gamma \bullet \mathtt{stale}_\sigma)}$$

▶ **Definition 25.** A Typing Relation $\mathcal{T}$ for a NAT-indexed family T equipped with a function `subst` which for all NATs k, l, given an element T_k and an $\mathrm{ENV}(k, l)$ returns an element T_l is said to be stable under substitution if for all NATs k and l, γ a $\mathrm{CONTEXT}_k$, Γ and Δ two USAGES_γ, t an element of T_k, σ a Type, ρ an $\mathrm{ENV}(k, l)$, θ a $\mathrm{CONTEXT}_l$ and Θ_1 and Θ_3 two USAGES_θ such that $\mathcal{T}(\Gamma, t, \sigma, \Delta)$ and $\mathrm{ENV}(\Theta_1, \rho, \Theta_3, \Gamma)$ holds then there exists a Θ_2 of type USAGES_θ such that $\mathcal{T}(\Theta_1, \mathtt{subst}(t, \rho), \sigma, \Theta_2)$ and $\mathrm{ENV}(\Theta_2, \rho, \Theta_3, \Delta)$.

▶ **Theorem 26.** *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 readjust the usages annotations. ◀

7 Functionality

One thing we did not mention before because it was seldom used in the proofs is the fact that all of these typing relations are functional when seen as various (binary or ternary) relations. Which means that 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 27.** We say that a relation R of type $\Pi(ri : RI).\Pi(ii : II).O(ri) \to Set$ is functional if for all relevant inputs ri, all pairs of irrelevant inputs ii_1 and ii_2 and for all pairs of outputs 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 28.** *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 USAGES annotation and outputs the inferred TYPEs.*

▶ **Lemma 29.** *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 USAGES annotation or the other. This means that given a USAGES annotation (whether the input one or the output one) and the rest of the arguments, the other USAGES annotation is uniquely determined.*

8 Typechecking

▶ **Theorem 30** (Decidability of Typechecking)**.** *Type-inference for INFER and Type-checking for CHECK are decidable. In other words, given a NAT k, γ a $\mathrm{CONTEXT}_k$ and Γ a USAGES_γ,*

1. *for all* INFER_k t*, it is decidable whether there is a* TYPE σ *and* Δ *a* USAGES_γ *such that* $\Gamma \vdash t \in \sigma \boxtimes \Delta$
2. *for all* TYPE σ *and* CHECK_k*, it is decidable whether there is* Δ *a* USAGES_γ *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:

▶ **Example 31.** We can for instance check that the search procedure succeeds in finding the `swapTyped` derivation we had written down as Example 11. 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 Related Work

We have already mentioned McBride's work [18] 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.

We can find a very concrete motivation for a predicate similar to our USAGE in Robbert Krebbers' thesis [15]. 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.

10 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 [3]. 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 [4] 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 The Rust Programming Language – Ownership, 2016. URL: `https://doc.rust-lang.org/book/ownership.html`.

2 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.

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

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

5 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.

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

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

8 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.

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

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

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

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

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

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

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

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

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

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

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

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

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

Typing derivation for `swap`

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

$$\Pi = \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{[] \bullet \texttt{fresh}_\tau \vdash 0 \in \tau \boxtimes [] \bullet \texttt{stale}_\tau}}{[] \bullet \texttt{fresh}_\tau \bullet \texttt{fresh}_\sigma \vdash 1 \in \tau \boxtimes [] \bullet \texttt{stale}_\tau \bullet \texttt{fresh}_\sigma}}{[] \bullet \texttt{fresh}_\tau \bullet \texttt{fresh}_\sigma \vdash \texttt{var}(1) \in \tau \boxtimes [] \bullet \texttt{stale}_\tau \bullet \texttt{fresh}_\sigma}}{[] \bullet \texttt{fresh}_\tau \bullet \texttt{fresh}_\sigma \vdash \tau \ni \texttt{neu(var}(1)) \boxtimes [] \bullet \texttt{stale}_\tau \bullet \texttt{fresh}_\sigma} \quad \dfrac{\dfrac{}{[] \bullet \texttt{stale}_\tau \bullet \texttt{fresh}_\sigma \vdash 0 \in \sigma \boxtimes [] \bullet \texttt{stale}_\tau \bullet \texttt{stale}_\sigma}}{[] \bullet \texttt{stale}_\tau \bullet \texttt{fresh}_\sigma \vdash \texttt{var}(0) \in \sigma \boxtimes [] \bullet \texttt{stale}_\tau \bullet \texttt{stale}_\sigma}}{[] \bullet \texttt{fresh}_\tau \bullet \texttt{fresh}_\sigma \vdash \tau \otimes \sigma \ni \texttt{prd(neu(var}(1)), \texttt{neu(var}(0)) \boxtimes [] \bullet \texttt{stale}_\tau \bullet \texttt{stale}_\sigma}$$