A Curry-Howard Correspondence for Linear, Reversible Computation

Kostia Chardonnet, Alexis Saurin, Benoît Valiron

A CURRY-HOWARD CORRESPONDENCE FOR LINEAR, REVERSIBLE COMPUTATION

KOSTIA CHARDONNET [a], ALEXIS SAURIN [b], AND BENOÎT VALIRON [c]

[a] Department of Computer Science and Engineering, University of Bologna, Italy
e-mail address: kostia.chardonnet@pm.me

[b] IRIF, Université Paris Cité, CNRS and Équipe INRIA Picube
e-mail address: alexis.saurin@irif.fr

[c] Université Paris-Saclay, CNRS, CentraleSupélec, ENS Paris-Saclay, Inria, LMF, 91190, Gif-sur-Yvette, France
e-mail address: benoit.valiron@universite-paris-saclay.fr

Abstract. In this paper, we present a linear and reversible programming language with inductive types and recursion. The semantics of the language is based on pattern-matching; we show how ensuring syntactical exhaustivity and non-overlapping of clauses is enough to ensure reversibility. The language allows to represent any Primitive Recursive Function. We then provide a Curry-Howard correspondence with the logic μMALL: linear logic extended with least and greatest fixed points allowing (co)inductive statements. The critical part of our work is to show how primitive recursion yields circular proofs that satisfy μMALL validity criterion and how the language simulates the cut-elimination procedure of μMALL.

1. Introduction

Computation and logic are two faces of the same coin. For instance, consider a proof s of $A \to B$ and a proof t of A. With the logical rule *Modus Ponens* one can construct a proof of B: Figure 1 features a graphical presentation of the corresponding proof. Horizontal lines stand for deduction steps —they separate conclusions (below) and hypotheses (above). These deduction steps can be stacked vertically up to axioms in order to describe complete proofs. In Figure 1 the proofs of A and $A \to B$ are symbolized with vertical ellipses. The ellipsis annotated with s indicates that s is a complete proof of $A \to B$ while t stands for a complete proof of A. From the perspective of a programmer, Figure 1 can however also be interpreted as the application of a function of type $A \to B$ to an argument for type A.

$$\dfrac{\begin{array}{cc} \overset{s}{\vdots} & \overset{t}{\vdots} \\ A \to B & A \end{array}}{B}$$

Figure 1: Modus Ponens

Key words and phrases: Linear Logic, Reversible Computation, Curry-Howard, Inductive types.

This is a extended version of the proceeding paper published in LIPICS [KP23, CSV23]. It contains all the missing proofs, together with additional discussions.

Preprint submitted to
Logical Methods in Computer Science

This connection is known as the *Curry-Howard correspondence* [Cur34, How80]. In this general framework, types correspond to formulas and programs to proofs, while program evaluation is mirrored with proof simplification (the so-called cut-elimination). The Curry-Howard correspondence formalizes the fact that the proof s of $A \to B$ can be regarded as a *function* —parametrized by an argument of type A— that produces a proof of B whenever it is fed with a proof of A. Therefore, the computational interpretation of Modus Ponens corresponds to the *application* of a function (i.e. s) of type $A \to B$ to an argument (i.e. t) of type A. When computing the corresponding program, one substitutes the parameter of the function with t and get a result of type B. On the logical side, this corresponds to substituting every axiom introducing A in the proof s with the full proof t of A. This yields a direct proof of B without any invocation of the "lemma" A.

Paving the way toward the verification of critical software, the Curry-Howard correspondence provides a versatile framework. It has been used to mirror first and second-order logics with dependent-type systems [BC13, Ler09], separation logics with memory-aware type systems [Rey02, JJKD17], resource-sensitive logics with differential privacy [GHH$^+$13], logics with monads with reasoning on side-effects [SHK$^+$16, MHRVM19], *etc.* One aspect that has not yet been covered by the Curry-Howard correspondence is the realm of *reversible computation*.

The idea of reversible computation comes from Landauer and Bennett [Lan61, Ben73], with the analysis of the link between the erasure of information and the dissipation of energy as heat [Lan61, BAP$^+$12]. A reversible process P can always be *reversed*, in the sense that there should be a process P' for which P followed by P' and P' followed by P both return the system to the initial state. Reversible computation has been described in many ways, for instance with the use of transition systems ensuring both *forward* and *backward* determinism, as in the case of reversible Turing Machines [MY07], but also with the use of reversible programming languages, both imperative and functional [GY23, YAG12, TA15, JSS14, SVV18, JKT18] and their semantics [CLV21, KR21].

In a (typed) reversible programming language, a function describes a bijection between the domain and the co-domain of the function: composing the function with its inverse yields an identity map. This connects to the notion of *type isomorphisms*, where we identify types and formulas. One then looks at when two types are "the same" according to the structure of some logic. For instance, in intuitionistic logic, there exists an isomorphism between the formulas $A \times B$ and $B \times A$, meaning that there is a pair of proofs $A \times B \vdash B \times A$ and $B \times A \vdash A \times B$ which, when *cut* together, reduces to the axiom rule on either $A \times B$ or $B \times A$, depending on the way the proofs were cut together.

Type isomorphisms have been studied in several contexts, such as finding suitable equations in various logical systems in order to characterize all isomorphisms between two types [BDCL92, BL85, SA97], and in higher-order type systems using game semantics [DL08]. On a practical side, the notion of type isomorphism has been at the root of code reuse by searching through libraries [Rit93]. Type isomorphisms have also been studied in the context of linear logic, for the multiplicative fragment [BDC99] and more recently for the multiplicative additive fragment [DGL23].

Although natural, the relationship between bijective functions and type isomorphisms has never been studied in details: this paper aims at initiating such an analysis.

On the language side, we base our study on the approach presented in [SVV18]. In this model, reversible computation is restricted to two main type constructs: the tensor, written $A \otimes B$ and the co-product, written $A \oplus B$. The former corresponds to the type of all pairs of elements of type A and elements of type B, while the latter represents the disjoint union of all elements of type A and elements of type B. For instance, a bit can be typed with $\mathbb{1} \oplus \mathbb{1}$, where $\mathbb{1}$ is a type with only one element. For expressivity, the type system is extended with the inductive type of lists, defining

$[A]$ with $\mathbb{1} \oplus (A \otimes [A])$. The language in [SVV18] offers the possibility to encode isomorphisms —reversible maps— with pattern matching, and features a term construction for building fixed points. The paper [SVV18] discusses how terminating functions indeed describe (total) bijections.

Although the type system hints at multiplicative additive linear logic (MALL), the connection has not formally been done. In this paper, we propose a correspondence with the logic μMALL [BM07, Bae12, BDS16, Dou17]: an extension of MALL with least and greatest fixed points allowing inductive and coinductive statements. This logic contains both a tensor and a co-product, and its strict linearity makes it a good fit for capturing the type system of [SVV18].

In the literature, multiple proof-systems have been considered for μMALL, some finitary proof system with explicit induction inferences à la Park [BM07, Bae12] as well as non-wellfounded proof systems which allow to build infinite derivations [BDS16, BDKS22]. The present paper focuses on the latter. In general, an infinite derivation is called a *pre-proof*. The ability to derive from infinite branches easily leads to inconsistency: to solve this problem, μMALL comes equipped with a *validity criterion*, describing when an infinite derivation can be considered as a logical proof.

Contributions. The main contribution of this work is a Curry-Howard correspondence between (a fragment of) μMALL and a purely reversible typed language in the style of [SVV18], with added generalized inductive types and terminating recursion, in which recursive functions must be structurally recursive. In particular, we show the following:

- Totality, reversibility. Every function that we can encode is reversible and total.
- Expressivity. The language captures the class of primitive recursive functions [RJ87].
- Curry-Howard correspondence. We show how well-typed encodable functions can be regarded as valid proofs of type isomorphisms between μMALL formulae.

Organization of the paper. The paper is organized as follows: in Section 2 we introduce the technical background needed for this work, split into two parts: Subsection 2.1 focuses on a reversible model of computation, called RPP [PPR20], while Subsection 2.2 introduces the logic μMALL. In Section 3 we introduce the language, its syntax, typing rules and operational semantics and show that any function that can be encoded in our language represents an isomorphism. We finally show in Section 4 the expressiveness of the language by encoding the language RPP into our language. Then in Section 5, we develop on the Curry-Howard correspondence part: we show how to translate a well-typed term from our language into a circular derivation of the logic μMALL, show that the given derivation respects the validity condition and show how our evaluation strategy simulates the cut-elimination procedure of the logic.

2. Technical Background

We introduce the technical background necessary for this work. The first part, in Subsection 2.1, focuses on the language of Recursive Primitive Permutations (RPP), which we use as a model of reversible computing to show the expressiveness of our language. The second part, in Subsection 2.2, focuses on the logic μMALL. We do not give all the details on μMALL but just the necessary definitions and intuitions. More details for RPP can be found in [PPR20] while [BM07, Bae12, BDS16, Dou17, BDKS22] contains more details on the logic μMALL.

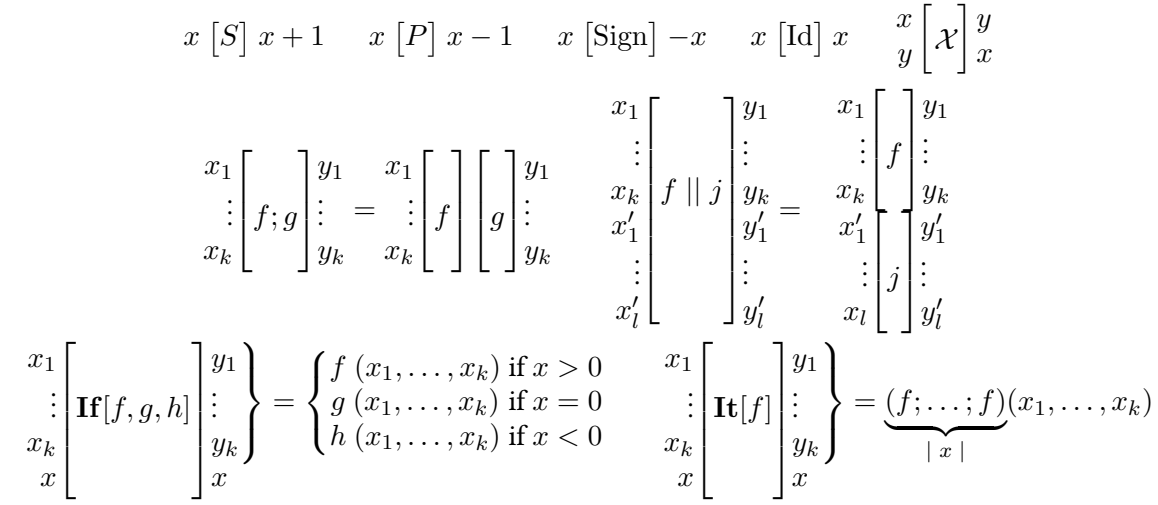


Figure 2: Generators of RPP

2.1. **Background on RPP.** Although reversible computation aims at capturing computability in a reversible setting, not all bijections are computable. Defining classes of such computable bijections have been the subject of several research programs, yielding reversible Turing machine [Ben73] and functional programming languages [GY23, YAG12, JSS14]. The main problem with a general approach towards computable functions is when we care about ruling out diverging functions. In the classical setting, *primitive recursive* functions (PRF) are an answer to this problem: primitive recursive functions are both *total* and computable. Although limited, this class nonetheless captures a sensible notion of computability.

The class of Recursive Primitive Permutations (RPP) [PPR20] is a class of total bijective functions, expressive enough to capture primitive recursion. Similar to what is done for RPF, RPP consists in a finite number of (total and bijective) generating functions and combinators. If for RPF, domains and codomains are products of $\mathbb{N}$, the elements of RPP are instead bijections over product of $\mathbb{Z}$.

2.1.1. *Generators and combinators.* More precisely, the generators are as follows. We first have the successor (S), predecessor (P), identity (Id) and sign-change (Sign), all acting on $\mathbb{Z}$, We then have the binary swap X acting on $\mathbb{Z} \times \mathbb{Z}$: $X(x,y) = (y,x)$. Finally given the RPP functions f, g, h acting on $\mathbb{Z}^k$ and a RPP function j acting on $\mathbb{Z}^l$, the following functions are also in RPP:

- The sequential composition $f;g$, defined by $(x_1,\ldots,x_k) \mapsto g(f(x_1,\ldots,x_k))$, bijection on $\mathbb{Z}^k$.
- The parallel composition $f \mid\mid j$, defined as the bijection $f \times j$ acting on $\mathbb{Z}^{k+l}$.
- The iterator $\mathbf{It}[f] \in \mathrm{RPP}^{k+1}$, which iterate f on the k first arguments, by the absolute value of its $(k+1)$th argument.
- The selection $\mathbf{If}[f,g,h] \in \mathrm{RPP}^{k+1}$, which applies either f, (resp. g or h) depending on whether its $(k+1)$th argument is strictly greater than 0 (resp. equal to 0 or strictly less than 0).

The semantics of generators and combinators is also given in Figure 2, using a graphical form where the left-hand-side variable of a diagram represent the input of the function and the right-hand-side its output.

Remark 2.1. Note that the class RPP is naturally graded: we can define RPP^k as the set of bijections where domain and codomain coincide. The class RPP is thus the union of all of the sub-classes RPP^k, when k ranges over the natural numbers.

Remark 2.2. In the original paper [PPR20], the class RPP is defined with two other kinds of generators: generalized permutations and weakening. They were only added for convenience: they can be derived from the ones we give. We therefore do not consider them here.

2.1.2. *Inversion.* For all k, the elements of RPP^k are indeed bijections over $\mathbb{Z}^k$. Moreover, the inverse of a given element can be inductively constructed as follows:

$$\begin{array}{ccc} \mathrm{Id}^{-1} = \mathrm{Id} & \mathrm{S}^{-1} = \mathrm{P} & \mathrm{P}^{-1} = \mathrm{S} \\ \mathrm{Sign}^{-1} = \mathrm{Sign} & \mathcal{X}^{-1} = \mathcal{X} & (g; f)^{-1} = f^{-1}; g^{-1} \\ (f \mid\mid g)^{-1} = f^{-1} \mid\mid g^{-1} & (\mathbf{It}[f])^{-1} = \mathbf{It}[f^{-1}] & (\mathbf{If}[f, g, h])^{-1} = \mathbf{If}[f^{-1}, g^{-1}, h^{-1}]. \end{array}$$

2.1.3. *Relationship with PRF.* Finally, and most importantly, RPP is PRF-Sound and Complete: it indeed captures the notion of primitive recursive permutations.

Theorem 2.3 (Soundness & Completeness [PPR20])**.** RPP *is PRF-Sound and PRF-Complete: it can represent any Primitive Recursive Function (PRF). Conversely, every function in* RPP *is indeed an element of the class RPF.* □

Remark 2.4. While RPP is expressive enough to encode any Primitive Recursive Function, it does so at the cost of auxiliary inputs and outputs. The canonical example requiring such extension is the Cantor Pairing, building a bijection between $\mathbb{N}$ and $\mathbb{N} \times \mathbb{N}$ [PPR20, Theorem 2 and Theorem 4]. Note however, that this is a standard trick in reversible models of computation.

2.2. **Background on μMALL.** Functional programming languages often feature the ability to encode *inductive types* and *coinductive types* as data structure. On a Curry-Howard point of view, how inductive and coinductive types and reasoning are related to Linear Logic is not directly clear.

From a proof theory point of view, inductive and coinductive reasoning have been studied for a long time. Most notably in the modal μ-calculus [DR79, Par69, Koz83], an extension of modal logic with fixed point operators. Modal logic was extended with a new formula of the form $\mu X.A$ where A is a formula, along with the dual operator $\nu X.A$. Already at this point in time, the μ and ν operators described the least and greatest fixed point operators. By the Curry-Howard correspondence, those new logics helped in modeling possibly infinite computation. Among those logics and derivations, *circular proofs* [San02, FS13], infinite proofs with finitely many subtrees, are of particular interest to represent recursive programs. As we are concerned with the particular case of Linear Logic, we look at the logic μMALL. In his PhD, Baelde [Bae08] extended linear logic with fixed points and showed the cut-elimination and the equivalence of provability with regards to the higher-order linear logic. Finally, Doumane et al [BDS16, Dou17] investigated more precisely the infinitary aspects of the derivations with a proof-theoretical approach, defining validity criterions and their properties.

2.2.1. *μMALL non-wellfounded derivations.* Baelde and his collaborators [Bae12, BDS16] introduced the logic μMALL, an extension of the additive and multiplicative fragment of linear logic [Gir87] with least and greatest fixed-points. The grammar for linear logic formulas, denoted by F, G, is extended with the construct $\mu X.F$ and its dual construct, $\nu X.F$ (where X is a type variable and μ, ν are variable binders), to be interpreted as the least and greatest fixed points of the operator $X \mapsto F$ respectively. These permit to form inductive and coinductive statements: one can for instance define the type of natural numbers as $\mathbb{N} = \mu X.\mathbb{1} \oplus X$ or of lists of type F as $[F] = \mu X.\mathbb{1} \oplus (F \otimes X)$.

Definition 2.5 (Pre-formulas)**.** Given an infinite set of variables $\mathcal{V} = \{X, Y, \dots\}$, the set of μMALL *pre-formulas* is inductively defined by the following grammar:

$$F, G ::= X \mid \mathbb{1} \mid \mathbb{0} \mid \top \mid \bot \mid F \otimes G \mid F \parr G \mid F \oplus G \mid F \,\&\, G \mid \mu X.F \mid \nu X.F.$$

In $\mu X.F$ (resp. $\nu X.F$), μ (resp. ν) binds the free occurrences of variable X in F.

A *formula* is a *closed* pre-formula (*i.e.* with no free variable).

The *(linear) negation* is the involution on pre-formulas defined as: $X^\perp = X, \mathbb{0}^\perp = \top, \mathbb{1}^\perp = \bot, (F \parr G)^\perp = F^\perp \otimes G^\perp, (F \oplus G)^\perp = F^\perp \,\&\, G^\perp, (\nu X.F)^\perp = \mu X.F^\perp$.

Given F, G two formulas, $F[X \leftarrow G]$ denotes the capture-free substitution of all the free-occurrences of variable X in F by G.

In the following, all derivations will manipulate formulas only. Setting $X^\perp = X$ is therefore harmless in the definition of negation. In the following, we shall consider one-sided sequents:

Definition 2.6 (μMALL sequents)**.** A *μMALL sequent* is a finite ordered list of μMALL formulas, usually denoted $\vdash \Gamma$. The concatenation of two lists of formulas Γ and Δ is simply written as Γ, Δ.

In the following, we will be interested in the non-wellfounded and circular proof systems used to derive μMALL sequents, that we shall simply refer to as μMALL sequent calculus in the following[1]: their inference rules are the usual MALL inference together with the addition of two new inference rules (in the one-sided sequent calculus) for the μ and ν connectives:

$$\frac{\vdash F[X \leftarrow \mu X.F], \Delta}{\vdash \mu X.F, \Delta}\ \mu \qquad \frac{\vdash F[X \leftarrow \nu X.F], \Delta}{\vdash \nu X.F, \Delta}\ \nu$$

Definition 2.7 (μMALL inference rules and pre-proofs)**.** A μMALL *pre-proof* of conclusion Γ is a sequent derivation tree coinductively generated by the rules of Table 1 such that the root is labelled with $\vdash \Gamma$. A *circular pre-proof* is a pre-proof having only finitely many distinct subtrees.

A formula is *principal* when it is the formula to which the rule is being applied. Given an inference rule r, we denote by $\mathrm{premiss}(r)$ the set of its premiss sequents (i.e. above the line).

Each inference rule is given together with an *immediate sub-occurrence* relation, relating a formula of the conclusion sequent with formulas of the premisses, depicted by the green and blue lines in the proof system. The reflexive transitive closure of this relation is written $F \sqsubseteq G$ when F is a formula occurrence (that is a position in a given sequent occurrence) that is above G in a proof and related by any number of immediate sub-occurrence relations.

The suboccurrence relation will most often be left implicit since it can usually be reconstructed from the context or the way the sequents are written. A case when we shall make explicit this suboccurrence relation is in the threading relation used to distinguish pre-proofs from valid proofs in what follows.

[1]Note that, in the literature, the sequent calculus we consider here is often called $\mu\mathsf{MALL}^\infty$ to distinguish it from Baelde and Miller's calculus but since we shall not consider the latter logical system in the present work, this introduces no confusion.

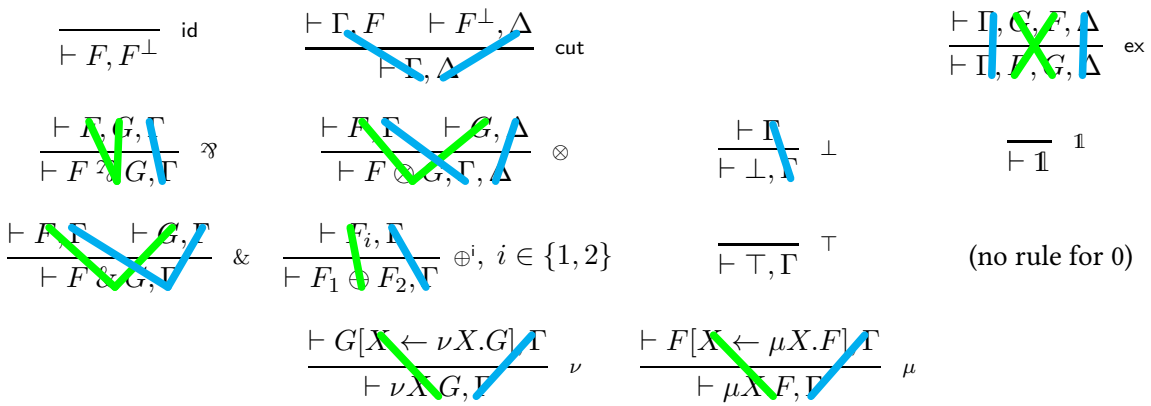


Table 1: Rules of μMALL. The sub-occurrence relation is graphically depicted with colored lines between formulas of the conclusion and premisses sequents, in green for the principal formula, in blue for the other formulas.

Note also that in what follows, we will most often use the exchange rule implicitly, denoting with an inference rule name, say $\otimes$, all the derived rules obtained by applying an exchange rule on top of each premise of the rule and below the conclusion of the rule. For example, we can freely write (here, $\mathsf{ex}^\star$ refers to a sequence of exchange rules):

$$\frac{\vdash \Gamma_1, F, \Gamma_2 \qquad \vdash \Delta_1, G, \Delta_2}{\vdash \Gamma_1, \Delta_2, F \otimes G, \Gamma_2, \Delta_1}\otimes \qquad \text{for} \qquad \frac{\dfrac{\vdash \Gamma_1, F, \Gamma_2}{\vdash F, \Gamma_1, \Gamma_2}\,\mathsf{ex}^\star \qquad \dfrac{\vdash \Delta_1, G, \Delta_2}{\vdash G, \Delta_1, \Delta_2}\,\mathsf{ex}^\star}{\dfrac{\vdash F \otimes G, \Gamma_1, \Gamma_2, \Delta_1, \Delta_2}{\vdash \Gamma_1, \Delta_2, F \otimes G, \Gamma_2, \Delta_1}\,\mathsf{ex}^\star}\otimes$$

What follows is an important notational convention that is used in most of the technical development of the paper when relating isos and μMALL proofs:

Notational convention 1. *The two-sided notation* $\Gamma \vdash \Delta$ *will denote sequent* $\vdash \Gamma^\perp, \Delta$.

For example we will write $\dfrac{\mu X.X \vdash F}{\mu X.X \vdash F}\,\nu$ for $\dfrac{\vdash \nu X.X, F}{\vdash \nu X.X, F}\,\nu$. (Note in particular that we do not modify the rule labels.)

Remark 2.8. This two-sided notation helps clarifying the computational reading of a derivation by distinguishing between input type and output type (even though in classical linear logic they are interchangeable) and it will also make lighter the notation for translating isos into μMALL proofs. In particular, we will essentially use notation $\Gamma \vdash \Delta$ in a specific case when the formulas in Γ and Δ are positive formulas (that is, are built from $\otimes, \oplus, \mu, \mathbb{1}, \mathbb{0}$ and atoms). This will allow for a convenient correspondence between *left formulas* (in the two sided-presentation) and *negative formulas* (in the one-sided presentation) that we will heavily rely on in the following sections.

Note that an alternative would have been to present μMALL in the two-sided notation all along: apart from being essentially superficial notational distinction (at least for our present concerns), this would have departed from most of the literature on μMALL.

For instance, taking the type of natural numbers that we defined earlier, it is possible to encode any natural number as a derivation of μMALL.

$$\dfrac{\dfrac{\vdots}{\vdash \nu X.X}\,\nu \qquad \dfrac{\vdots}{\vdash \mu X.X, \Gamma}\,\mu}{\vdash \Gamma}\,\texttt{cut}$$

Figure 3: Invalid pre-proof

Example 2.9. Representing the type of natural numbers by the pre-formula $\mathbb{N} = \mu X.\mathbb{1} \oplus X$, one can define the proof π_n, encoding any natural number n. We give this encoding, by induction on n as:

$$\pi_0 = \dfrac{\dfrac{\dfrac{}{\vdash \mathbb{1}}\,\mathbb{1}}{\vdash \mathbb{1} \oplus \mathbb{N}}\,\oplus^1}{\vdash \mu X.\mathbb{1} \oplus X}\,\mu \qquad\qquad \pi_n = \dfrac{\dfrac{\dfrac{\pi_{n-1}}{\vdash \mathbb{N}}}{\vdash \mathbb{1} \oplus \mathbb{N}}\,\oplus^2}{\vdash \mu X.\mathbb{1} \oplus X}\,\mu$$

The particularity is that the derivations are generated **coinductively** over the set of rules, allowing for infinite derivation trees, called pre-proofs.

Their name comes from the fact that, if we consider that any derivation is a *proof*, then we can prove any statement ψ using the cut-rule, as shown in Figure 3. This is why μMALL comes with a validity criterion, separating *pre-proofs* from actual *proofs*.

The usual cut-elimination reduction rules of MALL [Gir87] is extended with a new reduction rule and two commutation rules (here, $\sigma \in \{\mu, \nu\}$):

$$\dfrac{\dfrac{\vdash F^\perp[X \leftarrow \mu X.F^\perp], \Gamma}{\vdash \mu X.F^\perp, \Gamma}\,\mu \qquad \dfrac{\vdash F[X \leftarrow \nu X.F], \Delta}{\vdash \nu X.F, \Delta}\,\nu}{\vdash \Gamma, \Delta}\,\text{cut} \quad \rightsquigarrow \quad \dfrac{\vdash F^\perp[X \leftarrow \nu X.F^\perp], \Gamma \qquad \vdash F[X \leftarrow \nu X.F], \Delta}{\vdash \Gamma, \Delta}\,\text{cut}$$

$$\dfrac{\dfrac{\vdash F[X \leftarrow \sigma X.F], \Gamma, C}{\vdash \sigma X.F, \Gamma, C}\,\sigma \qquad \vdash C^\perp, \Delta}{\vdash \sigma X.F, \Gamma, \Delta}\,\text{cut} \quad \rightsquigarrow \quad \dfrac{\vdash F[X \leftarrow \sigma X.F], \Gamma, C \qquad \vdash C^\perp, \Delta}{\dfrac{\vdash F[X \leftarrow \sigma X.F], \Gamma, \Delta}{\vdash \sigma X.F, \Gamma, \Delta}\,\sigma}\,\text{cut}$$

2.2.2. *Bouncing Validity.* The validity criterion requires that each infinite branch can be justified by a form of coinductive reasoning. The criterion also ensures that the cut-elimination procedure holds. In the following, we will rely on a validity criterion recently introduced by Baelde *et al.*[BDKS22] in a slightly simplified reformulation.

This criterion is based on Girard's Geometry of Interaction where some data (namely a *thread*) moves through the derivation, following a subformula and collecting information (its *weight*). Then, one can analyze the collected information and determine whether or not it is *valid*. More formally, a *thread* [BDS16, BDKS22] is an infinite sequence of tuples of formulas, sequents and directions (either up or down) written $(F; \vdash \Phi; d)$. Intuitively, these threads follow some formula (here F) starting from a sequent of the derivation and starting by going up. The thread has the possibility to *bounce* on axioms and cuts and change its direction, either going back-down on an axiom or back-up on a cut. A thread will be called *valid* when it is non-stationary (does not follow a formula that is never a principal formula of a rule), and when in the set of formulas appearing infinitely often, the minimum formula (according to the subformula ordering) is a ν formula. We will then say that a derivation is valid if any infinite branch is inhabited by a valid thread.[2] The

[2]Note that in μMALL one needs an additional notion of *slices* for the additive part of the logic. However, since we will only consider a fragment of μMALL this notion will not be required in our work.

system of thread previously mentioned is developed in [BDS16] and extended in [BDKS22] and called the *bouncing-thread-validity*. This is the criterion that we present here and that we will use in Section 5.

We recall the formalism of bouncing validity from [BDKS22], adapting it to sequents as lists of formulas instead of locative occurrences [BDKS22, Def 2.5].

As mentioned earlier, not all derivations are indeed proofs. To answer this problem, μMALL comes with a validity criterion for derivations. This makes use of the notion of *bouncing-threads*: paths that travel along the infinite derivation and collect some information along the way. In order to formally define bouncing-threads we first need to introduce some notations: given an alphabet Σ, we denote by Σ^ω the set of infinite words over Σ and $\Sigma^\infty = \Sigma^* \cup \Sigma^\omega$. The letter ϱ will denote ordinals in $\omega + 1$. Finally, we use a special concatenation: given $u = (u_i)_{i \leq n < \omega}$ and $v = (v_i)_{i \in \varrho}$ such that $u_n = v_0$, we define $u \odot v$ as the concatenation of u and v without the first element of v, i.e. : $u \cdot (v_i)_{i \in \varrho \setminus \{0\}}$. For instance $aab \odot bac = aabac$.

We begin with the definition of *pre-thread* : the basic construction of a path along an infinite tree that follows a formula occurrence. Then, we only look at some special ones called *threads*, and finally define the notion of *valid thread* that validates an infinite branch as being valid. A pre-proof is a proof whenever all of its infinite branches are valid.

Definition 2.10 (Pre-thread [BDKS22])**.** A pre-thread is a sequence $(F_i, s_i, d_i)_{i \in \varrho \in \omega+1}$ of tuples of a formula (occurrence), a sequent (occurrence) and a direction $d \in \{\uparrow, \downarrow\}$ such that for all $i \in \varrho$ and $i + 1 \in \varrho$ one of the following holds:

- $d_i = d_{i+1} = \uparrow$, $s_{i+1} \in \text{premiss}(s_i)$ and $F_{i+1} \sqsubseteq F_i$
- $d_i = d_{i+1} = \downarrow$, $s_i \in \text{premiss}(s_{i+1})$ and $F_i \sqsubseteq F_{i+1}$
- $d_i = \downarrow, d_{i+1} = \uparrow$, s_i and s_{i+1} are the two premisses of the same cut rule and $F_i = F_{i+1}^\perp$
- $d_i = \uparrow, d_{i+1} = \downarrow$ and $s_i = s_{i+1} = \vdash F_i, F_{i+1}$ (or $\vdash F_{i+1}, F_i$) is the conclusion of an axiom rule.

Bouncing validity is a condition expressed on pre-threads of a specific form. To state it, we need to first recall the notion of *weight* of a pre-thread from [BDKS22]:

Definition 2.11 (Weight of a pre-thread)**.** Consider $t = (F_j, s_j, d_j)_{j \in \varrho \in \omega+1}$ a pre-thread. The weight of t, noted $w(t)$, is a word over $(w_j)_{j \in \varrho \in \omega+1} \in \{l, r, i, \overline{l}, \overline{r}, \overline{i}, \mathcal{W}, \mathcal{A}, \mathcal{C}\}^\infty$ such that for every $j \in \varrho$ one of the following holds:

- thread going up ($d_j = \uparrow$):
  - $w_j = l$ if $F_j = F_{j+1} \star G$ is principal in s_j, with $\star \in \{\parr, \otimes, \oplus, \&\}$;
  - $w_j = r$ if $F_j = G \star F_{j+1}$ is principal in s_j, with $\star \in \{\parr, \otimes, \oplus, \&\}$;
  - $w_j = i$ if $F_j = \sigma X.F$ is principal in s_j and $F_{j+1} = F[X \leftarrow \sigma X.F]$;
  - $w_j = \mathcal{W}$ if F_j is not principal in s_j; $w_j = \mathcal{A}$ if $d_{j+1} = \downarrow$.
- thread going down ($d_j = \downarrow$):
  - $w_j = \overline{l}$ if $F_{j+1} = F_j \star G$ is principal in s_{j+1}, with $\star \in \{\parr, \otimes, \oplus, \&\}$;
  - $w_j = \overline{r}$ if $F_{j+1} = G \star F_j$ is principal in s_{j+1}, with $\star \in \{\parr, \otimes, \oplus, \&\}$;
  - $w_j = \overline{i}$ if $F_{j+1} = \sigma X.F$ is principal in s_{j+1} and $F_j = F[X \leftarrow \sigma X.F]$;
  - $w_j = \mathcal{W}$ if F_{j+1} is not principal in s_{j+1};
  - $w_j = \mathcal{C}$ if $d_{j+1} = \uparrow$.

Weights of pre-thread allow us to characterize certain sets of pre-threads of interest.

Definition 2.12 (b-paths and h-paths)**.** We define two sets of words $\mathfrak{B}$ and $\mathfrak{H}$ inductively as follows:

$$\mathfrak{B} ::= \mathcal{C} \mid \mathfrak{B}\mathcal{W}^*\mathcal{A}\mathcal{W}^*\mathfrak{B} \mid \overline{x}\mathcal{W}^*\mathfrak{B}\mathcal{W}^*x \qquad\qquad \mathfrak{H} ::= \epsilon \mid \mathcal{A}\mathcal{W}^*\mathfrak{B}$$

A finite pre-thread t is called a *b-path* if $w(t) \in \mathfrak{B}$, and it is called a *h-path* if $w(t) \in \mathfrak{H}$.

We are now ready to state the notion of bouncing thread [BDKS22]:

Definition 2.13 ((Bouncing) thread)**.** A pre-thread t is a *(bouncing) thread* when it can be decomposed as $\bigodot_{i\in 1+\varrho}(H_i \odot V_i)$ where for all $i \in 1+\varrho$:

- $w(V_i) \in \{l, r, i, \mathcal{W}\}^\infty$, and it is non-empty if $i \neq \varrho$
- $w(H_i) \in \not{\mathbb{Y}}$, and it is non-empty if $i \neq 0$

The decomposition can be read as a thread initially going up, before going back down after encountering an axiom, accumulating debt in the form of the alphabet $\{\overline{l}, \overline{r}, \overline{i}\}$, which will need to be repaid by their opposites, $\{l, r, i\}$, when going back up after encountering a cut. Those dual alphabets correspond to steps of the cut-elimination: making sure that the correct formulas will at some point interact by a cut.

Such a decomposition is unique, and we call $(V_i)_{i\in 1+\varrho}$ the *visible part* of t and $(H_i)_{i\in 1+\varrho}$ the *hidden part*. A thread is *stationary* when its visible part is a finite sequence (of finite words) or when there exists $k \in 1+\varrho$ such that $w(V_i) \in \{\mathcal{W}\}^\infty$ for all $k \leq i \in 1+\varrho$.

We can now define the validity criterion on threads:

Definition 2.14 (Valid threads [BDKS22])**.** If we take the sequence of formulas followed by a non-stationary thread on its visible part and skipping the steps corresponding to $\mathcal{W}$ weights, we obtain an infinite sequence of formulas where each formula is an immediate subformula or an unfolding of the previous formula. The formulas appearing infinitely often admit a minimum with regard to the subformula ordering, such a formula is the *minimal formula of the thread*.

A non-stationary thread is *valid* if its minimal formula is a ν formula.

This notion of bouncing valid threads induces an associated notion of valid proof. Note that in the definition below, we use a simpler notion than that of [BDKS22], which we justify immediately after the definition and example.

Definition 2.15 (Validity of μMALL)**.** We say that an infinite branch is valid if there exists a valid thread starting from one of its sequents, whose visible part is contained in this branch.

A μMALL pre-proof π is *valid* if every infinite branch of π is valid.

Example 2.16. The infinite derivation $\dfrac{\dfrac{\vdots}{\vdash \mu X.X}\,\mu}{\vdash \mu X.X}\,\mu$ is not valid as the only existing thread $(\mu X.X; \vdash \mu X.X; \uparrow)^\infty$ has as sole minimal formula a μ formula.

This validity condition ensures cut-elimination for μMLL derivations, as proved in [BDKS22].

The reader will notice a difference in the notion of μMALL validity between [BDKS22] and our work. Indeed, bouncing validity as stated in [BDKS22] relies on the notion of *slice* (which comes from LL proof-nets). However, as we shall only be interested in a fragment of μMALL pre-proofs in the following, that is the target of the translations from *iso*s to proofs, we will not need to reason about slices (this will be further detailed in Section 5).

Remark 2.17. Original cut-elimination results on non-wellfounded μMALL [BDS16, BDKS22] dealt with sequents as sets of (locative) occurrences rather than sequents as lists as we consider here. However, it was shown recently by the second author [Sau23] how to transfer those cut-elimination results to sequent calculi manipulating lists, in a uniform way.

Example 2.18. Remember that $\mathbb{N} = \mu X.\mathbb{1} \oplus X$ and hence $\mathbb{N}^\perp = \nu X.\bot \,\&\, X$. We can then define the successor function as:

$$\pi_{\text{succ}} = \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash \mathbb{1}}\,\mathbb{1}}{\vdash \mathbb{1} \oplus \mathbb{N}}\,\oplus^1}{\vdash \mathbb{N}}\,\mu}{\mathbb{1} \vdash \mathbb{N}}\,\bot \qquad \dfrac{\dfrac{\dfrac{\pi_{\text{succ}}}{\mathbb{N} \vdash \mathbb{N}}}{\mathbb{N} \vdash \mathbb{1} \oplus \mathbb{N}}\,\oplus^2}{\mathbb{N} \vdash \mathbb{N}}\,\mu}{\mathbb{1} \oplus \mathbb{N} \vdash \mathbb{N}}\,\&}{\mathbb{N} \vdash \mathbb{N}}\,\nu$$

We can show that this derivation is valid, by following the two possible thread (painted in red and blue) in the infinite branch.

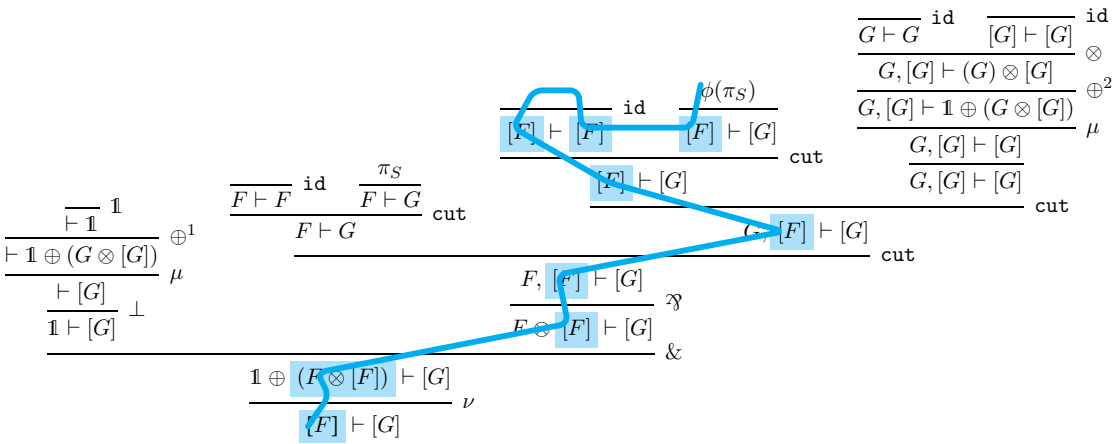
The red-thread t_l have for weight $(irWW)^\infty$ and the set of its formulas encountered infinitely often is $\{\nu X.\bot \,\&\, X, \bot \,\&\, \nu X.\bot \,\&\, X\}$. The smallest formula is a ν formula, validating the branch.

On the other hand, the blue-thread has weight $(WWir)^\infty$ and for smallest formula a μ formula and hence is not valid. Since we only require *one* valid thread per infinite branch, the whole derivation is therefore valid and a proof.

Example 2.19. Recall the type of lists of type ψ as $[\psi] = \mu X.\mathbb{1} \oplus (\psi \otimes X)$. Consider the μMALL proof π_S of $A \otimes B \vdash B \otimes A$, which swaps the input of type $A \otimes B$, defined as:

$$\pi_S = \dfrac{\dfrac{\overline{A \vdash A}\,\texttt{id} \qquad \overline{B \vdash B}\,\texttt{id}}{A, B \vdash B \otimes A}\,\otimes}{A \otimes B \vdash B \otimes A}\,\parr$$

We can use this proof to define the infinite, circular proof $\phi(\pi_S)$, which swap the elements of a given input list, where $F = A \otimes B$ and $G = B \otimes A$:



We painted in blue the pre-thread that will validate the proof. The pre-thread starts at the root on the formula $[F]$ and follows a sub-formula when some rules (here ν, $\&$, $\parr$) are applied to it. The pre-thread is then inactive during the multiple *cut* rules until it reaches the *id* rule, where the pre-thread bounces and starts going down before bouncing back up again in the *cut* rule, into the infinite branch, where the behavior of the pre-thread will repeat itself. One can then show that this pre-thread is indeed a thread, according to [BDKS22] and that it is valid: among the formulas it visits infinitely often on its visible part, the minimal formula is $[F]$, which is a ν formula.

Notice that the sub-derivation containing the `cut` and the proof π_S could have been simplified without the `cut` and by putting π_S directly. However, as we will see in Section 5, this derivation will correspond exactly to the translation of a term.

2.2.3. *Circular Representation.* As mentioned earlier, μMALL allows us to build infinite derivations and the fragment of *circular* proofs is of a special interest as it allows to be finitely represented and therefore subject to algorithmic treatment. We now introduce circular representations which provide a finite mean to deal with circular proofs with the help of *back-edges*. Back-edges are arrows in the derivation that represent a repetition of the derivation as a (strict) sub-derivation. Derivations with back-edges are represented with the addition of sequents marked by a back-edge label, noted $\vdash^f$, and an additional rule, $\dfrac{}{\vdash \Gamma}\ be(f)$, which represents a back-edge pointing to the sequent $\vdash^f$. We take the convention that from the root of the derivation to rule $be(f)$ there must be *exactly one* sequent annotated by f.

Example 2.20. An infinite derivation and three different circular representations with back-edges:

$$\dfrac{\dfrac{\vdots}{\vdash \mu X.X}\ \mu}{\vdash \mu X.X}\ \mu \qquad \dfrac{\dfrac{}{\vdash \mu X.X}\ be(f)}{\vdash^f \mu X.X}\ \mu \qquad \dfrac{\dfrac{\dfrac{}{\vdash \mu X.X}\ be(f)}{\vdash^g \mu X.X}\ \mu}{\vdash^f \mu X.X}\ \mu \qquad \dfrac{\dfrac{\dfrac{}{\vdash \mu X.X}\ be(g)}{\vdash^g \mu X.X}\ \mu}{\vdash^f \mu X.X}\ \mu$$

While a(n infinite) circular proof has infinitely many circular representations (depending on where the back-edge is placed), an unfolding operation allows to map those representations to the circular derivation:

Definition 2.21 (Unfolding)**.** We define the unfolding of a circular representation P with a valuation v from back-edge labels to circular-derivations along with a valuation by:

- $\mathcal{U}\left(P : \dfrac{P_1 \ \dots \ P_n}{\vdash \Gamma}\ r\ , v\right) = \dfrac{\mathcal{U}(P_1, v) \ \dots \ \mathcal{U}(P_n, v)}{\vdash \Gamma}\ r$
- $\mathcal{U}(be(f), v) = \mathcal{U}(v(f))$
- $\mathcal{U}\left(P : \dfrac{P_1 \ \dots \ P_n}{\vdash^f \Gamma}\ r\ , v\right) = \left(\dfrac{\mathcal{U}(P_1, v') \ \dots \ \mathcal{U}(P_n, v')}{\vdash \Gamma}\ r\right)$ with $v'(g) = \begin{cases} (P, v) & \text{if } g = f, \\ v(g) & \text{else.} \end{cases}$

3. First-order Isos

Sabry et al. introduced in [SVV18] a typed, functional, reversible programming language with the type of lists as the only infinite data-type. Although their paper eventually extends the framework to quantum computing, in our paper we only consider the reversible aspect. In particular, we focus on the first-order fragment, extended with both a more general rewriting system and more general inductive type. We consider arbitrary inductive types, making it possible to not only build list but also e.g. trees. We start by presenting the syntax of the language before moving on to its rewriting system and then its typing system.

3.1. Terms and Types.

The language is first-order: it consists of basic types, i.e. types for values, patterns, expressions, and general terms, and isos types, i.e. function types. The grammar for these notions is given as follows.

$$\begin{array}{llll} \text{(Base types)} & A, B & ::= & \mathbb{1} \mid A \oplus B \mid A \otimes B \mid \mu X.A \mid X \\ \text{(Isos types)} & T & ::= & A \leftrightarrow B \\ \text{(Values)} & v & ::= & () \mid x \mid \mathtt{inj}_l\ v \mid \mathtt{inj}_r\ v \mid \langle v_1, v_2 \rangle \mid \mathtt{fold}\ v \end{array}$$

$$
\begin{array}{llll}
\text{(Pattern)} & p & ::= & x \mid \langle p_1, p_2 \rangle \\
\text{(Expressions)} & e & ::= & v \mid \mathtt{let}\, p_1 = \omega\, p_2\, \mathtt{in}\, e \\
\text{(Isos)} & \omega & ::= & f \mid \{v_1 \leftrightarrow e_1 \mid \ldots \mid v_n \leftrightarrow e_n\} \mid \\
 & & & \mathbf{fix}\, f.\{v_1 \leftrightarrow e_1 \mid \ldots \mid v_n \leftrightarrow e_n\} \\
\text{(Terms)} & t & ::= & () \mid x \mid \mathtt{inj}_l\, t \mid \mathtt{inj}_r\, t \mid \langle t_1, t_2 \rangle \mid \\
 & & & \mathtt{fold}\, t \mid \omega\, t \mid \mathtt{let}\, p = t_1\, \mathtt{in}\, t_2
\end{array}
$$

Remark 3.1. The language makes use of free and bound variables. In order to avoid conflicts between variables we will always work up to α-conversion and use Barendregt's convention [Bar84, p.26] which consists in keeping all bound and free variable names distinct, even when this remains implicit.

3.1.1. *Values, patterns, expressions, terms and basic types.* They allow us to construct first-order terms. The constructors $\mathtt{inj}_l$ and $\mathtt{inj}_r$ represent the choice between either the left or right-hand side of a type of the form $A \oplus B$; the constructor $\langle , \rangle$ builds pairs of elements, with the corresponding type constructor $\otimes$; $\mathtt{fold}$ represents inductive constructor of the inductive type $\mu X.A$, where μ is a binder, binding the type-variable X in A. A value can serve both as a result and as a pattern in the defining clause of an iso. We write $(x_1, \ldots, x_n)$ for $\langle x_1, \langle \ldots, x_n \rangle \rangle$ or $\overrightarrow{x}$ when n is clear from the context and $A_1 \otimes \cdots \otimes A_n$ for $A_1 \otimes (\cdots \otimes A_n)$ and A^n for $A \otimes \cdots \otimes A$, the n-th tensor of A. We assume that every top-level type is closed.

Remark 3.2. In the original paper [SVV18], the type for lists of type A is defined equationally as $[A] = \mathbb{1} \oplus (A \otimes [A])$. In our language, we instead use the inductive type constructor $\mu X.A$ to define the type of lists as $[A] \triangleq \mu X.\mathbb{1} \oplus (A \otimes X)$. The list constructors becomes

- $[\,] \triangleq \mathtt{fold}\, (\mathtt{inj}_l\, ())$ for the empty list;
- $h :: t \triangleq \mathtt{fold}\, (\mathtt{inj}_r\, \langle h, t \rangle)$ for the addition of a head h to a list t.

3.1.2. *Isos types.* An iso of type $A \leftrightarrow B$ acts on terms of basic types. An iso is a function with inputs of type A and outputs of type B and defined as a set of clauses: $\{v_1 \leftrightarrow e_1 \mid \ldots \mid v_n \leftrightarrow e_n\}$. In each clause, the token v_i stands for an open value while e_i is an expression. For each i, v_i and e_i share the same variables: the variables of v_i can be seen as a binder for the free variables of e_i.

The construction $\mathbf{fix}\, g.\{v_1 \leftrightarrow e_1 \mid \ldots \mid v_n \leftrightarrow e_n\}$ represents the creation of a recursive function. The intuition (enforced by the operational semantics in Subsection 3.3) is that $\mathbf{fix}\, g.\{v_1 \leftrightarrow e_1 \mid \ldots \mid v_n \leftrightarrow e_n\}$ is identified with $\{v_1 \leftrightarrow e_1 \mid \ldots \mid v_n \leftrightarrow e_n\}$, where g is replaced with the fixed point $\mathbf{fix}\, g.\{v_1 \leftrightarrow e_1 \mid \ldots \mid v_n \leftrightarrow e_n\}$.

3.2. **Typing System.** The typing system is two-fold. We define typing judgments for values, patterns, terms and expressions, denoted with $\Theta; \Psi \vdash_e t : A$, and typing judgments for isos, denoted with $\Psi \vdash_i \omega : A \leftrightarrow B$. In the judgments, the (linear) contexts Θ are sets of pairs that consist of a term-variable and a base type, where each variable can only occur once. The (non-linear) context Ψ is a set of size at most one. It can only contain a pair of an iso-variable and an iso-type.

Based on [SVV18], the typing system is akin to the rules for μMALL. It requires two criteria developed below: (i) that every recursive iso terminates—we require structural recursion—and (ii) that every iso is exhaustive and non-overlapping on the left and on the right—to enforce the totality and bijectivity of isos.

3.2.1. *Typing of Terms and Expressions.* We say that a term t has type A under contexts Θ and Ψ if it can be inferred inductively from the following rules. We start by first defining the typing rules for values:

$$\frac{}{\emptyset; \Psi \vdash_e () : \mathbb{1}} \quad \frac{}{x : A; \Psi \vdash_e x : A} \quad \frac{\Theta; \Psi \vdash_e t : A}{\Theta; \Psi \vdash_e \mathtt{inj}_l\ t : A \oplus B} \quad \frac{\Theta; \Psi \vdash_e t : B}{\Theta; \Psi \vdash_e \mathtt{inj}_r\ t : A \oplus B}$$

$$\frac{\Theta_1; \Psi \vdash_e t_1 : A \quad \Theta_2; \Psi \vdash_e t_2 : B}{\Theta_1, \Theta_2; \Psi \vdash_e \langle t_1, t_2 \rangle : A \otimes B} \qquad \frac{\Theta; \Psi \vdash_e t : A[X \leftarrow \mu X.A]}{\Theta; \Psi \vdash_e \mathtt{fold}\ t : \mu X.A}$$

Lemma 3.3 (Linearity of term-variables)**.** *Let v be a value and $\Theta; \Psi \vdash_e v : A$, be a valid typing judgment. Then all the term-variables of v occus in Θ. Moreover, each such variable appears once and only once in v.*

Proof. The proof is done by structural induction on the derivation of $\Theta; \Psi \vdash_e v : A$. Notice how in all derivation rules, whenever the terms at the root are values, so are the terms appearing in the hypotheses. Moreover, notice how for the (only) branching rule for the product, the linear context Θ is partitioned into Θ_1 and Θ_2. □

Lemma 3.4 (Inversion)**.** *Given a well-typed value v of type A, either $v = x$ or one of the following is true:*

- $v = ()$ *and* $A = \mathbb{1}$*;*
- $v = \langle v_1, v_2 \rangle$ *where* v_1 *is of type* A_1 *and* v_2 *is of type* A_2 *and* $A = A_1 \otimes A_2$*;*
- $v = \mathtt{inj}_l\ v_1$ *and* v_1 *is of type* A_1 *and* $A = A_1 \oplus A_2$*;*
- $v = \mathtt{inj}_r\ v_2$ *and* v_2 *is of type* A_2 *and* $A = A_1 \oplus A_2$*;*
- $v = \mathtt{fold}\ v'$ *and* v' *is of type* $A'[X \leftarrow A']$ *and* $A = \mu X.A'$*.*

Proof. This can be proved directly by case analysis on v. □

With this Lemma, we can define a notion of *flattening* of a pattern p of type A, noted $|(p, A)|$ and which gives us a list of the variables in p as well as their corresponding types. The flattening is defined inductively as: $|(x, A)| = ([x], [A]), |(\langle p_1, p_2 \rangle, A_1 \otimes A_2)| = |(p_1, A_1)| + +|(p_2, A_2)|$ where $++$ is the pointwise list concatenation.

We can now define the typing derivation for general terms, where in the typing rule for the `let`, we have that $|(p, A)| = ([x_1, \ldots, x_n], [A_1, \ldots, A_n])$. By abuse of notation we shall later identify $|(p, A)|$ with the corresponding typing context $x_1 : A_1, \ldots x_n : A_n$. Note how this imposes that p is of type A in the context $|(p, A)|$.

$$\frac{\Psi \vdash_i \omega : A \leftrightarrow B \quad \Theta; \Psi \vdash_e t : A}{\Theta; \Psi \vdash_e \omega\ t : B}$$

$$\frac{\Theta_1; \Psi \vdash_e t_1 : A \qquad \Theta_2, x_1 : A_1, \ldots, x_n : A_n; \Psi \vdash_e t_2 : B}{\Theta_1, \Theta_2; \Psi \vdash_e \mathtt{let}\, p = t_1\ \mathtt{in}\ t_2 : B}$$

Note how the context Θ is used linearly while the context Ψ is not. By abuse of notation, we will write $\Theta \vdash_e t : A$ for $\Theta; \emptyset \vdash_e t : A$

3.2.2. *Exhaustivity, non-overlapping.* In order to apply an iso to a term, the iso must be of type $A \leftrightarrow B$ and the argument of type A. Since we want our language to represent *isomorphisms* we impose isos of shape $\{v_1 \leftrightarrow e_1 \mid \ldots \mid v_n \leftrightarrow e_n\}$ and type $A \leftrightarrow B$ to be exhaustive and non-overlapping. *Exhaustivity*, means that the expressions on the left (resp. on the right) of the clauses describe all possible values for the type A (resp. the type B). *Non-overlapping* means that two expressions cannot match the same value. For instance, the left and right injections $\texttt{inj}_l\ v$ and $\texttt{inj}_r\ v'$ are non-overlapping while a variable x is always exhaustive. In order to formally define these two notions, we first need to characterize *pattern-matching*, as follows.

Definition 3.5 (Pattern-Matching)**.** We say that a value v matches against a value v' if there exists a substitution σ, that is, a mapping from variables to (closed) values such that v under the substitution σ is equal to v'. This is noted $\sigma[v] = v'$ and is defined inductively over the following rules.

$$\frac{\sigma[e] = e'}{\sigma[\texttt{inj}_l\ e] = \texttt{inj}_l\ e'} \quad \frac{\sigma[e] = e'}{\sigma[\texttt{inj}_r\ e] = \texttt{inj}_r\ e'} \quad \frac{\sigma = \{x \mapsto e\}}{\sigma[x] = e} \quad \frac{\sigma[e] = e'}{\sigma[\texttt{fold}\ e] = \texttt{fold}\ e'}$$

$$\frac{\sigma_1[e_1] = e_1' \quad \sigma_2[e_2] = e_2' \quad \mathrm{supp}(\sigma_1) \cap \mathrm{supp}(\sigma_2) = \emptyset \quad \sigma = \sigma_1 \cup \sigma_2}{\sigma[\langle e_1, e_2\rangle] = \langle e_1', e_2'\rangle} \quad \frac{}{\sigma[()] = ()}$$

The *support* of a substitution σ is defined as $\mathrm{supp}(\sigma) = \{x \mid (x \mapsto v) \in \sigma\}$.

Definition 3.6 (Exhaustivity)**.** We call a set S of well-typed values of type A (where each value can be typed under a different context) *exhaustive* if, for every closed values v of type A there exists $v' \in S$ and σ such that $\sigma[v'] = v$.

Definition 3.7 (Non-Overlapping)**.** We call a set S of values of type A (where each value can be typed under a different context) *non-overlapping* if, for every closed values v of type A there exists at most one $v' \in S$ and σ such that $\sigma[v'] = v$.

In order to define the typing of isos in Subsubsection 3.2.3, we define the predicate $\texttt{OD}_A$, used to ensure that isos are exhaustive and non-overlapping, and that they indeed represent isomorphisms.

Definition 3.8 (Orthogonal Decomposition)**.** We say that a list of well-typed values S of type A satisfies $\texttt{OD}_A$, noted $\texttt{OD}_A(S)$, if it can be inferred inductively from the following rules.

$$\frac{}{\texttt{OD}_A(\{x\})}\ \texttt{OD-}var \qquad \frac{}{\texttt{OD}_{\mathbb{1}}(\{()\})}\ \texttt{OD-}\mathbb{1} \qquad \frac{\texttt{OD}_A(S) \qquad \texttt{OD}_B(T)}{\texttt{OD}_{A\oplus B}(\{\texttt{inj}_l\ v \mid v \in S\} \cup \{\texttt{inj}_r\ v \mid v \in T\})}\ \texttt{OD-}\oplus$$

$$\frac{\texttt{OD}_{A[X \leftarrow \mu X.A]}(S)}{\texttt{OD}_{\mu X.A}(\{\texttt{fold}\ v \mid v \in S\})}\ \texttt{OD-}\mu \qquad \frac{\texttt{OD}_A(\pi_1(S)), \forall v \in \pi_1(S), \texttt{OD}_B(S_v^1)}{\texttt{OD}_{A\otimes B}(S)}\ \texttt{OD-}\otimes_1$$

$$\frac{\texttt{OD}_B(\pi_2(S)), \forall v \in \pi_2(S), \texttt{OD}_A(S_v^2)}{\texttt{OD}_{A\otimes B}(S)}\ \texttt{OD-}\otimes_2$$

In the rules $\texttt{OD-}\otimes_1$ and $\texttt{OD-}\otimes_2$, we have $S = \{\langle v_1, v_1'\rangle, \ldots, \langle v_n, v_n'\rangle\}$ and the sets $\pi_1(S)$ and $\pi_2(S)$ are respectively $\{v \mid \langle v, w\rangle \in S\}$ and $\{w \mid \langle v, w\rangle \in S\}$. The sets S_v^1 and S_v^2 are respectively $\{w \mid \langle v, w\rangle \in S\}$ and $\{w \mid \langle w, v\rangle \in S\}$.

Remark 3.9. The definition of $\texttt{OD}_A(S)$ is inspired from [SVV18]. The main difference is the rule for the tensor, and the new rule $\texttt{OD-}\mu$ for inductive types. The problem in the original definition comes from the impossibility to type important isos. Indeed, the system in [SVV18] has the following rule for the tensor:

$$\frac{\texttt{OD}_A(S_1) \qquad \texttt{OD}_B(S_2)}{\texttt{OD}_{A\otimes B}\{\langle v_1, v_2\rangle \mid v_1 \in S_1, v_2 \in S_2, \mathrm{FV}(v_1) \cap \mathrm{FV}(v_2) = \emptyset\}.} \tag{3.1}$$

Consider the following encoding of the so-called Toffoli gate:

$$\left\{ \begin{array}{ccc} \langle \mathtt{inj}_l\ (), x\rangle & \leftrightarrow & \langle \mathtt{inj}_l\ (), x\rangle \\ \langle \mathtt{inj}_r\ (), \mathtt{inj}_l\ ()\rangle & \leftrightarrow & \langle \mathtt{inj}_r\ (), \mathtt{inj}_r\ ()\rangle \\ \langle \mathtt{inj}_r\ (), \mathtt{inj}_r\ ()\rangle & \leftrightarrow & \langle \mathtt{inj}_r\ (), \mathtt{inj}_l\ ()\rangle \end{array} \right\} : (\mathbb{1} \oplus \mathbb{1})^2 \leftrightarrow (\mathbb{1} \oplus \mathbb{1})^2$$

This iso does not satisfy the criterion shown in Equation 3.1. Indeed, the variable x on the right-hand-side of the pair overlap with the values $\mathtt{inj}_l\ ()$ and $\mathtt{inj}_r\ ()$. Meanwhile, this set of clauses satisfy our definition of $\mathtt{OD}$.

We can already show that $\mathtt{OD}$ is sound:

Lemma 3.10 (Soundness of $\mathtt{OD}_A(S)$)**.** *Given a set S of well-typed values of type A such that $\mathtt{OD}_A(S)$ holds, then S is exhaustive and non-overlapping, i.e. for all closed values v of type A, there exists a unique $v' \in S$ and a unique σ such that $\sigma[v'] = v$.*

Proof. By induction on a derivation of $\mathtt{OD}_A(S)$:

- $\mathtt{OD}_A(\{x\})$ is direct, and $\mathtt{OD}_{\mathbb{1}}(\{()\})$ follows from Lemma 3.4.
- Assume that the root of the derivation is
$$\frac{\mathtt{OD}_A(S_A) \qquad \mathtt{OD}_B(S_B)}{\mathtt{OD}_{A\oplus B}(\{\mathtt{inj}_l\ v \mid v \in S_A\} \cup \{\mathtt{inj}_r\ v \mid v \in S_B\})}\ \mathtt{OD}\text{-}\oplus.$$
In this case, the closed value v is of type $A \oplus B$. By Lemma 3.4 we know that either $v = \mathtt{inj}_l\ \tilde{v}$ where $\tilde{v}$ is a closed value of type A or $v = \mathtt{inj}_r\ \tilde{v}$ where $\tilde{v}$ is a closed value of type B. Both cases are proved similarly: let us focus on the first case. By induction hypothesis on $\mathtt{OD}_A(S_A)$ we know that there exists a unique $v' \in S_A$ and a unique σ' such that $\sigma'[v'] = \tilde{v}$. Therefore, we know that $\sigma'[\mathtt{inj}_l\ v'] = \mathtt{inj}_l\ \tilde{v}$. By definition of the pattern-matching, we know that there is no term of the shape $(\mathtt{inj}_r\ -)$ and no substitution δ such that $\delta[\mathtt{inj}_l\ v'] = (\mathtt{inj}_r\ -)$, therefore $\mathtt{inj}_l\ v'$ is the only value in $(\{\mathtt{inj}_l\ v \mid v \in S\} \cup \{\mathtt{inj}_r\ v \mid v \in T\})$ that matches $\mathtt{inj}_l\ \tilde{v}$ with the unique substitution σ'.
- Assume that the root of the derivation is
$$\frac{\mathtt{OD}_{A[X\leftarrow \mu X.A]}(S)}{\mathtt{OD}_{\mu X.A}(\{\mathtt{fold}\ v \mid v \in S\})}\ \mathtt{OD}\text{-}\mu.$$
By Lemma 3.4 we know that $v = \mathtt{fold}\ \tilde{v}$ where $\tilde{v}$ is a closed value of type $A[X \leftarrow \mu X.A]$. By induction hypothesis on $\mathtt{OD}_{A[X\leftarrow \mu X.A]}(S)$, we know that there exists a unique $v' \in S$ and a unique σ' such that $\sigma'[v'] = \tilde{v}$. Therefore, we know that $\sigma'[\mathtt{fold}\ v'] = \mathtt{fold}\ \tilde{v}$. Since by induction hypothesis we know that v' and σ' are unique for $\tilde{v}$, then $\mathtt{fold}\ v'$ and σ' are also unique for $\mathtt{fold}\ \tilde{v}$.
- Assume that the root of the derivation is
$$\frac{\mathtt{OD}_A(\pi_1(S)), \forall v \in \pi_1(S), \mathtt{OD}_B(S^1_v)}{\mathtt{OD}_{A\otimes B}(S)}\ \mathtt{OD}\text{-}\otimes_1.$$
Recall that $S = \{\langle v_1, v'_1\rangle, \ldots, \langle v_n, v'_n\rangle\}$. By Lemma 3.4 we know that $v = \langle \tilde{v}, \tilde{v}'\rangle$, where $\tilde{v}$ is a closed value of type A and $\tilde{v}'$ is a closed value of type B. By induction hypothesis on $OD_A(\pi_1(S))$ we know that there exists a unique v_i (for $i \in \{1, \ldots, n\}$) and a unique σ_i such that $\sigma_i[v_i] = \tilde{v}$. Similarly, there exists a unique w in $S^1_{v_i}$ and a unique σ_w such that $\sigma_w[w] = \tilde{v}'$. Due to the fact that the free variables of well-typed values are all distinct (by Lemma 3.3) the support of σ_i and σ_w are disjoint. We can then conclude that $(\sigma_i \cup \sigma_w)[\langle v_i, w\rangle] = \langle \tilde{v}, \tilde{v}'\rangle$.
- The case for $\mathtt{OD}\text{-}\otimes_2$ is similar. □

Remark 3.11. While we have shown that the definition of the OD predicate is sound, we can notice that it is non-complete: there exists some set of clauses that are exhaustive and non-overlapping, but that do not match the OD predicate. For instance, the following set of clauses of type $((A \oplus B) \oplus C) \otimes ((D \oplus E) \oplus F)$:

$$\left\{ \begin{array}{lcl} \langle \mathtt{inj}_r\ (x), \mathtt{inj}_l\ (\mathtt{inj}_r\ (y)) \rangle & \leftrightarrow & \dots \\ \langle \mathtt{inj}_r\ (x), \mathtt{inj}_l\ (\mathtt{inj}_l\ (y)) \rangle & \leftrightarrow & \dots \\ \langle \mathtt{inj}_r\ (x), \mathtt{inj}_r\ (y) \rangle & \leftrightarrow & \dots \\ \langle \mathtt{inj}_l\ (x), \mathtt{inj}_r\ (y) \rangle & \leftrightarrow & \dots \\ \langle \mathtt{inj}_l\ (\mathtt{inj}_l\ (x)), \mathtt{inj}_l\ (y) \rangle & \leftrightarrow & \dots \\ \langle \mathtt{inj}_l\ (\mathtt{inj}_r\ (x)), \mathtt{inj}_l\ (y) \rangle & \leftrightarrow & \dots \end{array} \right\}.$$

While this set of clauses is indeed exhaustive and non-overlapping and would be accepted in a typical functional programming language such as OCaml or Haskell, it does not check the OD predicate: both OD-$\otimes_1$ and OD-$\otimes_2$ are not satisfied. In the first case, the clauses $\mathtt{inj}_l\ (x)$ and $\mathtt{inj}_l\ (\mathtt{inj}_l\ (x))$ are overlapping, and similarly for OD-$\otimes_2$. However, for each closed value of type $((A \oplus B) \oplus C) \otimes ((D \oplus E) \oplus F)$, there is a single clause that matches. Although this could be considered as a limitation, for the purpose of the paper it is expressive enough, in the sense of Section 4.

3.2.3. *Typing of Isos.* We can now define the typing of isos. An iso will be required to satisfy OD both on the left and the right-hand-sides of the set of clauses. Moreover, since we allow isos to be recursive we need to make sure that on a given input an iso always terminates. Indeed, otherwise it would not necessarily be a total function. For that we introduce a notion of *structural recursion*, stating that a recursive call can only be used on a strict subterm of the input.

Formally, we say that an iso ω has type T under some context Ψ, noted $\Psi \vdash_i \omega$ if it can be inferred inductively by the following rules:

$$\frac{}{f : A \leftrightarrow B \vdash_i f : A \leftrightarrow B}$$

$$\frac{\begin{array}{cccc} \Theta_1 \vdash_e v_1 : A & \dots & \Theta_n \vdash_e v_n : A & \mathtt{OD}_A(\{v_1, \dots, v_n\}) \\ \Theta_1; \Psi \vdash_e e_1 : B & \dots & \Theta_n; \Psi \vdash_e e_n : B & \mathtt{OD}_B(\{\mathit{Val}(e_1), \dots, \mathit{Val}(e_n)\}) \end{array}}{\Psi \vdash_i \{v_1 \leftrightarrow e_1 \mid \dots \mid v_n \leftrightarrow e_n\} : A \leftrightarrow B.}$$

$$\frac{f : A \leftrightarrow B \vdash_i \{v_1 \leftrightarrow e_1 \mid \dots \mid v_n \leftrightarrow e_n\} : A \leftrightarrow B \qquad \textbf{fix}\ f.\omega \text{ is structurally recursive}}{\Psi \vdash_i \textbf{fix}\ f.\{v_1 \leftrightarrow e_1 \mid \dots \mid v_n \leftrightarrow e_n\} : A \leftrightarrow B}$$

There, $\mathit{Val}(e)$ is defined as $\mathit{Val}(\mathtt{let}\, p = \omega\ p'\ \mathtt{in}\ e) = \mathit{Val}(e)$, and $\mathit{Val}(v) = v$ otherwise. In the second rule, the term variables of the $\Theta_1, \dots, \Theta_n$ are bound by the pattern-matching construction: they are not visible outside the iso, thus not appearing anymore in the typing context.

In the last rule, we furthermore ask **fix** $f.\omega$ to be *structurally recursive* :

Definition 3.12 (Structurally Recursive)**.** We say that an iso **fix** $f.\{v_1 \leftrightarrow e_1 \mid \dots \mid v_n \leftrightarrow e_n\} : A \leftrightarrow B$ is structurally recursive when $A = A_1 \otimes \dots \otimes A_m$ and $B = B_1 \otimes \dots \otimes B_l$, and when $A_j = \mu X.C_1$ and $B_k = \mu X.C_2$ for some $1 \leq j \leq m$ and $1 \leq k \leq l$. Moreover, we require that for all $i \in \{1, \dots, n\}$ the value v_i is of the form $(v_i^1, \dots, v_i^m)$ and that v_i^j is either:

- A closed value, in which case f does not occurs in e_i and $\mathit{Val}(()e_i) = (v'^1_i, \dots, v'^l_i)$ is such that v'^k_i is also a closed value.

- An open value, in which case for all subterms of the form $(\texttt{let}\, p' = f\ p\ \texttt{in}\ \ldots)$ in e_i we have $p' = (y_1, \ldots, y_l)$ and $p = (x_1, \ldots, x_m)$ where $x_j : \mu X.B$ is a strict subterm of v_i^j. We also ask that $\mathit{Val}(()e_i) = (v'^1_i, \ldots, v'^m_i)$, where v'^k_i contains the value y_k as a strict subterm.

Finally, we ask that there is at least one clause where v_i^j is a closed value. We call the value v_i^j (resp. the variable x_j) the *decreasing argument* (resp. the *focus*) of the structurally recursive criterion.

Remark 3.13. Note that, given a well typed closed term $\emptyset; \emptyset \vdash t : A$, along its typing derivation, the iso-context will be either empty or a singleton.

Remark 3.14. We impose a simple notion of structural recursion: the typing rules of isos allow to have at most one iso-variable in the context. Indeed, the last typing rule of isos erases the context Ψ and replaces it with the new context containing only a single iso-variable. Also, we cannot have intertwined recursive calls. This mean that for an iso of the form $\mathbf{fix}\, f.\{v_1 \leftrightarrow e_1 \mid \ldots \mid v_n \leftrightarrow e_n\}$, if e_i is

$$\begin{array}{l}\texttt{let}\, p_{i,1} = \omega_{i,1}\ p'_{i,1}\ \texttt{in}\\ \ldots\\ \texttt{let}\, p_{i,n} = \omega_{i,n}\ p'_{i,n}\ \texttt{in}\ v'_i,\end{array}$$

for each $j \in \{1, \ldots, n\}$, either $\omega_{i,j} = f$ or f does not appear at all inside $\omega_{i,j}$.

Example 3.15. We can define the iso of type $: A \oplus (B \oplus C) \leftrightarrow C \oplus (A \oplus B)$, where a, b and c are variables as:

$$\left\{\begin{array}{ll}\texttt{inj}_l\ (a) & \leftrightarrow \texttt{inj}_r\ (\texttt{inj}_l\ (a))\\ \texttt{inj}_r\ (\texttt{inj}_l\ (b)) & \leftrightarrow \texttt{inj}_r\ (\texttt{inj}_r\ (b))\\ \texttt{inj}_r\ (\texttt{inj}_r\ (c)) & \leftrightarrow \texttt{inj}_l\ (c)\end{array}\right\}$$

The first clause is typed under context $\{a : A\}$, the second clause under context $\{b : B\}$ and the third clause under context $\{c : C\}$. Notice that the `OD` predicate holds for both the right and left-hand-side clauses.

Example 3.16. Consider a (closed) iso $\vdash \omega : A \leftrightarrow B$, and recall the list construction of Remark 3.2. Let us define the operation $\mathit{map}(\omega) : [A] \leftrightarrow [B]$ as follows.

$$\mathit{map}(\omega) : [A] \leftrightarrow [B] = \mathbf{fix}\ f. \left\{\begin{array}{ll}[\,] & \leftrightarrow [\,]\\ h :: t & \leftrightarrow \texttt{let}\, h' = \omega\ h\ \texttt{in}\\ & \quad\ \texttt{let}\, t' = f\ t\ \texttt{in}\\ & \quad\ h' :: t'\end{array}\right\}$$

Note how the iso is indeed structurally recursive. Note also how the left and right-hand-side of the $\leftrightarrow$ respect both the criteria of exhaustivity—every value of each type is being covered by at least one expression—and non-overlapping—no two expressions cover the same value.

Example 3.17. There are of course fixed points that do not respect the structural recursive constraint, such as e.g. $\mathbf{fix}\, f.\{x \leftrightarrow \texttt{let}\, y = f\ x\ \texttt{in}\ y\}$.

3.3. **Operational Semantics.** From now on, we will only consider well-typed terms. Our language is equipped with a rewriting system $\rightarrow$ on terms, that follows a deterministic call-by-value strategy: each argument of a function is fully evaluated before applying the β-reduction. This is done through the use of an evaluation context $C[]$, which consists of a term with a hole (where $C[t]$ is C where the hole has been filled with t). Due to the deterministic nature of the strategy we directly obtain the unicity of the normal forms. The evaluation of an iso applied to a value

relies on pattern-matching, as discussed in Definition 3.5. Because we ensure exhaustivity and non-overlapping (Lemma 3.10), the pattern-matching always succeeds on closed, well-typed values.

Definition 3.18 (Substitution)**.** Applying a substitution σ to an expression t, written $\sigma(t)$, is defined as: $\sigma(()) = (), \sigma(x) = v$ if $\{x \mapsto v\} \subseteq \sigma, \sigma(\mathtt{inj}_r\ (t)) = \mathtt{inj}_r\ (\sigma(t)), \sigma(\mathtt{inj}_l\ (t)) = \mathtt{inj}_l\ (\sigma(t)), \sigma(\mathtt{fold}\ (t)) = \mathtt{fold}\ (\sigma(t)), \sigma(\langle t, t'\rangle) = \langle \sigma(t), \sigma(t')\rangle, \sigma(\omega\, t) = \omega\, \sigma(t)$ and $\sigma(\mathtt{let}\, p = t_1\ \mathtt{in}\ t_2) = (\mathtt{let}\, p = \sigma(t_1)\ \mathtt{in}\ \sigma(t_2))$ where the variables from p do not occur in the domain of σ (this assumption can always be fulfilled thanks to α-renaming).

Remark 3.19. Notice the difference between $\sigma[v] = t$ and $\sigma(t)$. The first one defines the fact that v and t matches under the substitution σ, while the second one defines the actual substitution. In fact if $\sigma[v] = t$ then $\sigma(v) = t$. Notice that if $\sigma[v] = t$ then the support of σ is the set of free variables of v.

As we mentioned, the rewriting system is defined through the use of evaluation contexts and with the help of the pattern-matching and substitution from Definition 3.5 and Definition 3.18 respectively.

Definition 3.20 (Evaluation Contexts)**.** The evaluation contexts C are defined as:

$$C ::= [\,] \mid \mathtt{inj}_l\ C \mid \mathtt{inj}_r\ C \mid \omega\ C \mid \mathtt{let}\, p = C\ \mathtt{in}\ t \mid \langle C, t\rangle \mid \langle v, C\rangle \mid \mathtt{fold}\ C$$

Definition 3.21 (Evaluation relation $\to$)**.** We define $\to$ the rewriting system of our language as follows:

$$\frac{t_1 \to t_2}{C[t_1] \to C[t_2]}\ \text{Cong} \qquad \frac{\sigma[p] = v}{\mathtt{let}\, p = v\ \mathtt{in}\ t \to \sigma(t)}\ \text{LetE} \qquad \frac{}{(\mathbf{fix}\ f.\omega)\ v \to (\omega[f := (\mathbf{fix}\ f.\omega)])\ v}\ \text{IsoRec}$$

$$\frac{\sigma[v_i] = v}{\{v_1 \leftrightarrow e_1 \mid \ldots \mid v_n \leftrightarrow e_n\}\ v \to \sigma(e_i)}\ \text{IsoApp}$$

As usual, we denote $\to^*$ for the reflexive transitive closure of $\to$.

3.4. **Property of the typed language.** The language features the standard properties of typed languages, namely progress and subject reduction. As there are two classes of variables: term and iso variables, we have two substitution lemmas.

Lemma 3.22 (Substitution Lemma for term variables)**.** *Assume that for all i, we have $\Theta_i; \Psi \vdash_e v_i : A_i$. Furthermore, assume that $\Sigma, x_1 : A_1, \ldots, x_n : A_n; \Psi \vdash_e t : B$. Then, provided that $\sigma = \{x_1 \mapsto v_1, \ldots, x_n \mapsto v_n\}$ we have $\Sigma, \Theta_1, \ldots, \Theta_n; \Psi \vdash_e \sigma(t) : B$.*

Proof. By induction on t.

- Case x, then $n = 1$ and $x_1 = x$ and $\Sigma = \emptyset$ and we have $\sigma = \{x_1 \mapsto v_1\}$ for some v_1 of type B under some context Θ_1, then we get $\Theta_1; \Psi \vdash_e \sigma(x) : B$ which leads to $\Theta_1; \Psi \vdash_e v : B$ which is typable by our hypothesis.
- Case $()$, the result follows immediately from the fact that Σ is empty, $n = 0$ and $\sigma(()) = ()$.
- Case $\mathtt{inj}_l\ t'$. By induction hypothesis on t' we know that $\Sigma, \Theta_1, \ldots, \Theta_n; \Psi \vdash_e \sigma(t') : A$ is typable and therefore by applying the typing rule for the $\mathtt{inj}_l\ -$, and by definition of the substitution, we get that $\Sigma, \Theta_1, \ldots, \Theta_n; \Psi \vdash_e \sigma(\mathtt{inj}_l\ t') : A \oplus B$ is typable.
- Case $\mathtt{inj}_r\ t', \mathtt{fold}\ t', \omega\ t'$ are similar.

- Case $\langle t_1, t_2 \rangle$, by typing we get that we can split Σ into Σ_1, Σ_2 and the variables $x_1, \dots, x_n$ are split into two parts for typing both t_1 or t_2 depending on whenever or they occur freely in t_1 or t_2, w.l.o.g. say that $x_1, \dots, x_l$ are free in t_1 and $x_{l+1}, \dots, x_n$ are free in t_2 then we get:
$$\frac{\Sigma_1, x_1 : A_1, \dots, x_l : A_l; \Psi \vdash_e t_1 : B_1 \qquad \Sigma_2, x_{l+1} : A_{l+1}, \dots, x_n : A_n; \Psi \vdash_e t_2 : B_2}{\Sigma_1, \Sigma_2, x_1 : A_1, \dots, x_l : A_l, x_{l+1} : A_{l+1}, \dots, x_n : A_n; \Psi \vdash_e \langle t_1, t_2 \rangle : B_1 \otimes B_2}$$
By substitution, we get that $\sigma(\langle t_1, t_2 \rangle) = \langle \sigma(t_1), \sigma(t_2) \rangle$, so we get the following typing derivation which is completed by induction hypothesis on the subterms:
$$\frac{\Sigma_1, \Theta_1, \dots, \Theta_l; \Psi \vdash_e \sigma_1(t_1) : B_1 \qquad \Sigma_2, \Theta_{l+1}, \dots, \Theta_n; \Psi \vdash_e \sigma_2(t_2) : B_2}{\Sigma_1, \Sigma_2, \Theta_1, \dots, \Theta_l, \Theta_{l+1}, \dots, \Theta_n; \Psi \vdash_e \langle \sigma_1(t_1), \sigma_2(t_2) \rangle : B_1 \otimes B_2}$$
- Case $\texttt{let}\, p = t_1 \,\texttt{in}\, t_2$ Similar to the case of the tensor. □

Lemma 3.23 (Substitution Lemma for Isos)**.** *The following substitution properties hold:*

- *If* $\Theta; f : T_1 \vdash_e t : A$ *and* $g : T_2 \vdash_i \omega : T_1$ *then* $\Theta; g : T_2 \vdash_e t[f \leftarrow \omega] : A$.
- *If* $f : T_1 \vdash_i \omega_1 : T_2$ *and* $h : T_3 \vdash_i \omega_2 : T_1$ *then* $h : T_3 \vdash_i \omega_1[f \leftarrow \omega_2] : T_2$.

Proof. We prove those two propositions by mutual induction on t and ω_1.

Terms, by induction on t.

- If $t = x$ or $t = ()$ then in the first case $\Theta = x : A$ and in the second case $\Theta = \emptyset$ and $A = \mathbb{1}$ by Lemma 3.4, and in both cases we have that $t[f \leftarrow \omega] = t$. Therefore, we have to type $x : A; g : T_2 \vdash x : A$ in one case and $\emptyset; g : T_2 \vdash_e () : \mathbb{1}$ in the other. Both are possible by definition of the typing system.
- If $t = \texttt{inj}_l\, t'$ or $\texttt{inj}_r\, t'$ or $\texttt{fold}\, t'$ or $\langle t_1, t_2 \rangle$ or $\texttt{let}\, p = t_1 \,\texttt{in}\, t_2$, then similarly to the proof of Lemma 3.22 the substitution goes to the subterms and we can apply the induction hypothesis.
- If $t = \omega'\ t'$. In that case, the substitution goes to both subterms: $t[f \leftarrow \omega] = (\omega'[f \leftarrow \omega])\ (t'[f \leftarrow \omega])$. We can then conclude by induction hypothesis on t' and by the induction hypothesis on isos.

Isos, by induction on ω_1.

- If $\omega_1 = f$, then we get $h : T_3 \vdash_i f[f \leftarrow \omega_2] : T_2$ which is typable by hypothesis.
- If $\omega_1 = g \neq f$ is impossible by our typing hypothesis.
- If $\omega_1 = \mathbf{fix}\ g.\omega$, then by typing f does not occur in ω_1 so nothing happens.
- If $\omega_1 = \{v_1 \leftrightarrow e_1 \mid \dots \mid v_n \leftrightarrow e_n\}$, then, by definition of the substitution we have that
$$\begin{aligned}&\{v_1 \leftrightarrow e_1 \mid \dots \mid v_n \leftrightarrow e_n\}[f \leftarrow \omega_2]\\ &= \{v_1[f \leftarrow \omega_2] \leftrightarrow e_1[f \leftarrow \omega_2] \mid \dots \mid v_n[f \leftarrow \omega_2] \leftrightarrow e_n[f \leftarrow \omega_2]\}\\ &= \{v_1 \leftrightarrow e_1[f \leftarrow \omega_2] \mid \dots \mid v_n \leftrightarrow e_n[f \leftarrow \omega_2]\}\end{aligned}$$
in which case we apply the induction hypothesis of isos on terms. □

We can then deduce subject reduction and progress, as follows.

Lemma 3.24 (Subject Reduction)**.** *If* $\Theta; \Psi \vdash_e t : A$ *and* $t \to t'$ *then* $\Theta; \Psi \vdash_e t' : A$.

Proof. By induction on $t \to t'$ and direct by Lemma 3.22 and Lemma 3.23 □

Lemma 3.25 (Progress)**.** *If* $\vdash_e t : A$ *then, either* t *is a value, or* $t \to t'$.

Proof. Direct by induction on $\vdash_e t : A$. The two possible reduction cases, $\omega\ v$ and $\texttt{let}\, p = v \,\texttt{in}\, t$ always reduce by typing, pattern-matching and by Lemma 3.10. □

3.5. **Inversion.** One thing to note is that, in many models of reversible computing, such as Reversible Turing Machines [Ben73, MY07, AG11], assembly code [Vie95], imperative or functional programming languages [GY23, YAG12], each step of the evaluation is reversible. It is a local property. This is not the case in our language. When a term t reduces to another term t', one cannot directly inverse the rewriting system to go from t' to t. Instead, reversibility should be understood in the broader context of the design of the language, in which any isos can be inverted. Therefore, given an iso $\omega : A \leftrightarrow B$ one can build its inverse $\omega^\perp : B \leftrightarrow A$. The inverse operation is defined inductively on ω and is given in Definition 3.26.

Definition 3.26 (Inversion)**.** Given an iso ω, we define its dual $\omega^\perp$ as: $f^\perp = f, (\mathbf{fix}\ f.\omega)^\perp = \mathbf{fix}\ f.\omega^\perp, \{(v_i \leftrightarrow e_i)_{i\in I}\}^\perp = \{((v_i \leftrightarrow e_i)^\perp)_{i\in I}\}$ and the inverse of a clause as:

$$\begin{pmatrix} v_1 \leftrightarrow \mathtt{let}\, p_1 = \omega_1\, p_1'\, \mathtt{in} \\ \quad \cdots \\ \quad \mathtt{let}\, p_n = \omega_n\, p_n'\, \mathtt{in}\ v_1' \end{pmatrix}^\perp := \begin{pmatrix} v_1' \leftrightarrow \mathtt{let}\, p_n' = \omega_n^\perp\, p_n\, \mathtt{in} \\ \quad \cdots \\ \quad \mathtt{let}\, p_1' = \omega_1^\perp\, p_1\, \mathtt{in}\ v_1 \end{pmatrix}.$$

We can show that the inverse is well-typed and behaves as expected:

Lemma 3.27 (Inversion is well-typed)**.** *If $f : C \leftrightarrow D \vdash_i \ \omega : A \leftrightarrow B$, then $f : D \leftrightarrow C \vdash_i \ \omega^\perp : B \leftrightarrow A$.*

Proof. The proof is done by structural induction on the typing derivation of $f : C \leftrightarrow D \vdash_i \ \omega : A \leftrightarrow B$.

- The case where $\omega = f$ is direct.
- For the case where the root of the typing derivation is $\omega = \{v_1 \ \leftrightarrow\ e_1 \ \mid \ \ldots \ \mid \ v_m \ \leftrightarrow\ e_m\}$, recall the typing rule of isos:

$$\frac{\begin{array}{cccc} \Theta_1 \vdash_e v_1 : A & \ldots & \Theta_m \vdash_e v_m : A & \mathtt{OD}_A(\{v_1, \ldots, v_m\}) \\ \Theta_1; f : C \leftrightarrow D \vdash_e e_1 : B & \ldots & \Theta_m; f : C \leftrightarrow D \vdash_e e_m : B & \mathtt{OD}_B(\{\mathit{Val}(e_1), \ldots, \mathit{Val}(e_m)\}) \end{array}}{f : C \leftrightarrow D \vdash_i \ \{v_1 \ \leftrightarrow\ e_1 \ \mid \ \ldots \ \mid \ v_m \ \leftrightarrow\ e_m\} : A \leftrightarrow B.}$$

First, notice that the predicate $\mathtt{OD}$ still holds for $\omega^\perp$ as values do not change. We then need to check that if one clause is typable, then its dual is also typable. Without lost of generality we consider the case of the first clause, the other being similar:

$$\begin{pmatrix} v_1 \leftrightarrow \mathtt{let}\, p_1 = \omega_1\, p_1'\, \mathtt{in} \\ \quad \cdots \\ \quad \mathtt{let}\, p_n = \omega_n\, p_n'\, \mathtt{in}\ v_1' \end{pmatrix}^\perp := \begin{pmatrix} v_1' \leftrightarrow \mathtt{let}\, p_n' = \omega_n^\perp\, p_n\, \mathtt{in} \\ \quad \cdots \\ \quad \mathtt{let}\, p_1' = \omega_1^\perp\, p_1\, \mathtt{in}\ v_1 \end{pmatrix}.$$

Without loss of generality, using Remark 3.1 we can assume that all of the term variables introduced in the p_i are fresh. By typing we know that $\Theta_1; f : C \leftrightarrow D \vdash_e v_1 : A$ and

$$\Theta_1; f : C \leftrightarrow D \vdash_e \mathtt{let}\, p_1 = \omega_1\, p_1'\, \mathtt{in} \ldots v_1' : B.$$

Writing Ψ for $f : C \leftrightarrow D$, a typing derivation for the latter, written π_1, starts with

$$\dfrac{\dfrac{\begin{array}{cc} \pi_{\omega_1} & \pi_{p_1'} \\ \vdots & \vdots \\ \Psi \vdash_i \omega_1 : A_1 \leftrightarrow B_1 & \Theta_1'; \Psi \vdash_e p_1' : A_1 \end{array}}{\Theta_1'; \Psi \vdash_e \omega_1\, p_1' : B_1} \qquad \begin{array}{c} \pi_2 \\ \vdots \\ \Theta_1'', \Sigma_1; \Psi \vdash_e \mathtt{let}\, p_2 = \omega_2\, p_2'\, \mathtt{in} \ldots v_1' : B \end{array}}{\Theta_1', \Theta_1''; \Psi \vdash_e \mathtt{let}\, p_1 = \omega_1\, p_1'\, \mathtt{in} \ldots v_1' : B}$$

where Θ_1 is decomposed as Θ'_1, Θ''_1. By linearity we have $|(p'_1, A_1)| = \Theta'_1$, and $|(p_1, B_1)| = \Sigma_1$. At each level $1 < i < n$, the typing derivation π_i is similar:

$$\dfrac{\dfrac{\begin{array}{c}\pi_{\omega_i}\\ \vdots \\ \Psi \vdash_i \omega_i : A_i \leftrightarrow B_i\end{array} \quad \begin{array}{c}\pi_{p'_i}\\ \vdots \\ \Theta'_i; \Psi \vdash_e p'_i : A_i\end{array}}{\Theta'_i; \Psi \vdash_e \omega_i\ p'_i : B_i} \quad \begin{array}{c}\pi_{i+1}\\ \vdots \\ \Theta''_i, \Sigma_i; \Psi \vdash_e \mathtt{let}\, p_{i+1} = \omega_{i+1}\ p'_{i+1}\ \mathtt{in} \ldots v'_1 : B\end{array}}{\Theta''_{i-1}, \Sigma_{i-1}; \Psi \vdash_e \mathtt{let}\, p_i = \omega_i\ p'_i\ \mathtt{in} \ldots v'_1 : B}$$

where $\Theta''_{i-1}, \Sigma_{i-1}$ is decomposed as Θ'_i, Θ''_i. By linearity we have $|(p'_i, A_i)| = \Theta'_i$, and $|(p_i, B_i)| = \Sigma_i$. At the level n, there is only one let-term left, and the proof π_n is

$$\dfrac{\dfrac{\begin{array}{c}\pi_{\omega_n}\\ \vdots \\ \Psi \vdash_i \omega_n : A_n \leftrightarrow B_n\end{array} \quad \begin{array}{c}\pi_{p'_n}\\ \vdots \\ \Theta'_n; \Psi \vdash_e p'_n : A_n\end{array}}{\Theta'_n; \Psi \vdash_e \omega_n\ p'_n : B_n} \quad \begin{array}{c}\pi_{v'_1}\\ \vdots \\ \Theta''_n, \Sigma_n; \Psi \vdash_e v'_1 : B\end{array}}{\Theta''_{n-1}, \Sigma_{n-1}; \Psi \vdash_e \mathtt{let}\, p_n = \omega_n\ p'_n\ \mathtt{in}\ v'_1 : B} \tag{3.2}$$

where $\Theta''_{n-1}, \Sigma_{n-1}$ is decomposed as Θ'_n, Θ''_n, and where by linearity we have $|(p'_n, A_n)| = \Theta'_n$, and $|(p_n, B_n)| = \Sigma_n$.

Let us now build a typing derivation for the judgment

$$\Theta''_n, \Sigma_n; \Psi \vdash_e \mathtt{let}\, p'_n = \omega_n^{\perp}\ p_n\ \mathtt{in} \ldots \mathtt{let}\, p'_1 = \omega_1^{\perp}\ p_1\ \mathtt{in}\ v_1 : A$$

starting from the top. First, we can build the typing derivation $\pi_1^{\perp}$

$$\dfrac{\dfrac{\begin{array}{c}\pi^{\perp}_{\omega_1}\\ \vdots \\ \Psi^{\perp} \vdash_e \omega_1^{\perp} : B_1 \leftrightarrow A_1\end{array} \quad \begin{array}{c}\pi_{p_1}\\ \vdots \\ \Sigma_1; \Psi^{\perp} \vdash_e p_1 : B_1\end{array}}{\Sigma_1; \Psi^{\perp} \vdash_e \omega_1^{\perp}\ p_1 : A_1} \quad \begin{array}{c}\pi_{v_1}\\ \vdots \\ \Theta'_1, \Theta''_1; \Psi^{\perp} \vdash_e v_1 : B\end{array}}{\Theta''_1, \Sigma_1; \Psi^{\perp} \vdash_e \mathtt{let}\, p'_1 = \omega_1^{\perp}\ p_1\ \mathtt{in}\ v_1 : A}$$

Remember how Θ'_1 corresponds to p'_1 and Σ_1 corresponds to p_1 and $\Psi^{\perp}$ is $f : D \leftrightarrow C$, but it is not used in π_{p_1} nor π_{v_1}. The derivation $\pi^{\perp}_{\omega_1}$ is built by invoking the induction hypothesis, and π_{p_1} by recalling that Σ_1 is $|(p_1, B_1)|$. The derivation π_{v_1} comes from the hypothesis that $\Theta_1; \Psi \vdash_e v_1 : A$, that $\Theta_1 = \Theta'_1, \Theta''_1$ and that f is not used in v_1, so that we can safely change its type to $D \leftrightarrow C$ in the typing context.

We now iteratively derive the proof π_i of the intermediate typing judgments

$$\Theta''_i, \Sigma_i; \Psi^{\perp} \vdash_e \mathtt{let}\, p'_i = \omega_i^{\perp}\ p_i\ \mathtt{in} \ldots \mathtt{let}\, p'_1 = \omega_1^{\perp}\ p_1\ \mathtt{in}\ v_1 : A$$

for incremental $1 < i \leq n$, assuming that we already have π_{i-1}. The typing derivation π_i is as follows:

$$\dfrac{\dfrac{\begin{array}{c}\pi^{\perp}_{\omega_i}\\ \vdots \\ \Psi^{\perp} \vdash_e \omega_i^{\perp} : B_i \leftrightarrow A_i\end{array} \quad \begin{array}{c}\pi_{p_i}\\ \vdots \\ \Sigma_i; \Psi^{\perp} \vdash_e p_i : B_i\end{array}}{\Sigma_i; \Psi^{\perp} \vdash_e \omega_i^{\perp}\ p_i : A_i} \quad \begin{array}{c}\pi^{\perp}_{i-1}\\ \vdots \\ \Theta''_{i-1}, \Sigma_{i-1}; \Psi^{\perp} \vdash_e \mathtt{let}\, p'_{i-1} = \omega_{i-1}^{\perp}\ p_{i-1}\ \mathtt{in} \ldots v_1 : A\end{array}}{\Theta''_i, \Sigma_i; \Psi^{\perp} \vdash_e \mathtt{let}\, p'_i = \omega_i^{\perp}\ p_i\ \mathtt{in} \ldots v_1 : A.}$$

Remember how $\Theta''_{i-1}, \Sigma_{i-1}$ and Θ''_i, Θ'_i are two decompositions of the same typing context. The typing derivation π_{p_i} comes from the fact that Σ_i is $|(p_i, B_i)|$, and $\pi^{\perp}_{\omega_i}$ comes from the induction hypothesis.

When i reaches n, we get a derivation for the typing judgment

$$\Theta''_n, \Sigma_n; \Psi^{\perp} \vdash_e \texttt{let}\, p'_n = \omega_n^{\perp}\ p_n\ \texttt{in} \ldots \texttt{let}\, p'_1 = \omega_1^{\perp}\ p_1\ \texttt{in}\ v_1 : A.$$

On the other hand, in Equation 3.2 we have a typing derivation $\pi_{v'_1}$ for

$$\Theta''_n, \Sigma_n; \Psi^{\perp} \vdash_e v'_1 : B.$$

As f is not used in v'_1 (since it is a value), this also gives us a typing derivation

$$\Theta''_n, \Sigma_n \vdash_e v'_1 : B.$$

The reasoning we did on the clause $\{v_1 \leftrightarrow \texttt{let}\, p_1 = \omega_1\ p'_1\ \texttt{in} \ldots \texttt{let}\, p_n = \omega_n\ p'_n\ \texttt{in}\ v'_1\}$ is general and can be reproduced for all clauses $\{v_i \leftrightarrow e_i\}$. We can then derive a proof of the desired judgment

$$f : D \leftrightarrow C \vdash_i \ \{v_1 \leftrightarrow e_1 | \ldots | v_m \leftrightarrow e_m\}^{\perp} : B \leftrightarrow A.$$

- For the case of a recursive iso $\mathbf{fix}\ f.\omega$ of type $A \leftrightarrow B$, one can directly invoke the inductive hypothesis on ω and rely on the symmetry of the criterion for structural recursion. □

Example 3.28. Recall the iso $\mathit{map}(\omega)$ defined in Example 3.16. Its inverse $\mathit{map}(\omega)^{\perp}$ is

$$\mathit{map}(\omega)^{\perp} = \mathbf{fix}\ f. \left\{ \begin{array}{ll} [\,] & \leftrightarrow [\,] \\ h' :: t' & \leftrightarrow \texttt{let}\, t = f\ t'\ \texttt{in} \\ & \quad\ \texttt{let}\, h = \omega^{\perp} h'\ \texttt{in} \\ & \quad\ h :: t \end{array} \right\}$$

It can indeed be typed with $[B] \leftrightarrow [A]$.

We are left to show that our isos indeed represent isomorphisms, meaning that given some iso $\vdash_i \ \omega : A \leftrightarrow B$ and some value $\vdash_e v : A$, then $\omega^{\perp}(\omega\ v) \to^* v$. For that we need the following lemma:

Lemma 3.29 (Commutativity of substitution)**.** *Let σ_1, σ_2 and v, such that $\sigma_1 \cup \sigma_2$ closes v and $\texttt{supp}(\sigma_1) \cap \texttt{supp}(\sigma_2) = \emptyset$ then $\sigma_1(\sigma_2(v)) = \sigma_2(\sigma_1(v))$*

Proof. Direct by induction on v as σ_1 and σ_2 have disjoint support: In the case where $v = x$ then either $\{x \mapsto v'\} \in \sigma_1$ or $\{x \mapsto v'\} \in \sigma_2$ and hence $\sigma_1(\sigma_2(x)) = v' = \sigma_2(\sigma_1(x))$. All the other cases are by direct induction hypothesis as the substitutions enter the subterms. □

Lemma 3.30. *For all well-typed isos $\vdash_i \ \omega : A \leftrightarrow B$, and for all well-typed values $\vdash_e v : A$, if $\omega\ v \to^* v'$ a value, then $\omega^{\perp}\ v' \to^* v$.*

Proof. Write X for the number of IsoRec rules being used in the rewrite sequence

$$\omega\ v \to^* v'. \tag{3.3}$$

We prove the result by induction on the lexicographical order on (X, size of ω). Let us proceed by case distinction on ω.

- ω cannot be an iso-variable since the typing context of $\vdash_i \omega : A \leftrightarrow B$ is empty.

- Suppose that ω is of the form $\mathbf{fix}\ f.\omega'$. Then $\omega^\perp$ is $\mathbf{fix}\ f.\omega'^\perp$. The rewriting in Equation 3.3 starts with an IsoRec rules as follows:

$$((\mathbf{fix}\ f.\omega')\ v) \to ((\omega'[f := \mathbf{fix}\ f.\omega'])\ v) \to^* v_0$$

As the reduction from $(\omega'[f := \mathbf{fix}\ f.\omega'])\ v$ to v_0 takes one less number of IsoRec rules we can apply the induction hypothesis and we get that

$$(\omega'[f := \mathbf{fix}\ f.\omega'])^\perp v_0 \to^* v \tag{3.4}$$

Notice that we have

$$(\omega'[f := \mathbf{fix}\ f.\omega'])^\perp = \omega'^\perp[f := \mathbf{fix}\ f.\omega'^\perp] \tag{3.5}$$

We can now show that $\omega^\perp v_0 \to^* v$. By definition of the inverse and the rewriting system we have

$$\omega^\perp v_0 = (\mathbf{fix}\ f.\omega'^\perp)v_0 \to_{\mathrm{IsoRec}} \omega'^\perp[f := \mathbf{fix}\ f.\omega'^\perp]v_0$$

By using Equation 3.5 and Equation 3.4 we can conclude that

$$\omega'^\perp[f := \mathbf{fix}\ f.\omega'^\perp]v_0 \to^* v$$

- Consider the case where ω is of the form $\{v_1 \leftrightarrow e_1 \mid \ldots \mid v_n \leftrightarrow e_n\}$. Without loss of generality, we consider that the clause $\{v_1 \leftrightarrow e_1\}$ matches with v and therefore there is a substitution σ_0 such that $\sigma_0[v_1] = v$, and

$$\omega\ v \to \sigma_0(e_1). \tag{3.6}$$

Assume that e_1 is

$$\mathtt{let}\, p_1 = \omega_1\ p_1'\ \mathtt{in}\ \ldots \mathtt{let}\, p_n = \omega_n\ p_n'\ \mathtt{in}\ v_1'.$$

By linearity, we can decompose σ_0 into $\sigma_0^1, \ldots, \sigma_0^n, \sigma_0^{n+1}$ such that, after substitution we obtain

$$\begin{array}{rcl} & & \mathtt{let}\, p_1 = \omega_1\, \sigma_0^1(p_1')\ \mathtt{in} \\ \sigma_0(e_1) & = & \ldots \\ & & \mathtt{let}\, p_n = \omega_n\, \sigma_0^n(p_n')\ \mathtt{in}\ \sigma_0^{n+1}(v_1') \end{array}$$

By Lemma 3.25, each `let` construction will reduce, and by the rewriting strategy we will first rewrite $\mathtt{let}\, p_1 = \omega_1\ p_1'\ \mathtt{in} \ldots$ before rewriting the other `let`. We have that

$$\omega_1\ \sigma_0^1(p_1') \to^* \overline{v_1}$$

for some $\overline{v_1}$. At this point, a pattern-matching occurs between p_1 and $\overline{v_1}$, generating a new substitution σ_1: we have $\sigma_1[p_1] = \overline{v_1}$. Again, due to linearity this new substitution can be split into $\sigma_1^2, \ldots, \sigma_1^n, \sigma_1^{n+1}$, where the support of σ_1^i are the free variables of $\sigma_0^i(p_i')$ appearing in p_1 for $i \in \{2, \ldots, n\}$ and the support of σ_1^{n+1} is the free variables of v_1' appearing in p_1. We get

$$\begin{array}{l} \mathtt{let}\, p_1 = \omega_1\, \sigma_0^1(p_1')\ \mathtt{in} \\ \mathtt{let}\, p_2 = \omega_2\, \sigma_0^2(p_2')\ \mathtt{in} \\ \mathtt{let}\, p_3 = \omega_3\, \sigma_0^3(p_3')\ \mathtt{in} \\ \ldots \\ \mathtt{let}\, p_n = \omega_n\, \sigma_0^n(p_n')\ \mathtt{in} \\ \sigma_0^{n+1}(v_1') \end{array} \to^* \begin{array}{l} \mathtt{let}\, p_1 = \overline{v_1}\ \mathtt{in} \\ \mathtt{let}\, p_2 = \omega_2\, \sigma_0^2(p_2')\ \mathtt{in} \\ \mathtt{let}\, p_3 = \omega_3\, \sigma_0^3(p_3')\ \mathtt{in} \\ \ldots \\ \mathtt{let}\, p_n = \omega_n\, \sigma_0^n(p_n')\ \mathtt{in} \\ \sigma_0^{n+1}(v_1') \end{array} \to \begin{array}{l} \mathtt{let}\, p_2 = \omega_2\, \sigma_1^2(\sigma_0^2(p_2'))\ \mathtt{in} \\ \mathtt{let}\, p_3 = \omega_3\, \sigma_1^3(\sigma_0^3(p_3'))\ \mathtt{in} \\ \ldots \\ \mathtt{let}\, p_n = \omega_n\, \sigma_1^n(\sigma_0^n(p_n'))\ \mathtt{in} \\ \sigma_1^{n+1}(\sigma_0^{n+1}(v_1')) \end{array} \tag{3.7}$$

The operation can be repeated, and to each p_k is associated a value $\overline{v_k}$ and a substitution σ_k with $\sigma_k[p_k] = \overline{v_k}$ and that can splitted in $\sigma_k^{k+1}, \ldots, \sigma_k^{n+1}$. They verify

$$\omega_k(\sigma_{k-1}^k \ldots \sigma_1^k \sigma_0^k(p_k')) \to^* \overline{v_k}. \tag{3.8}$$

Continuing the rewriting started in Equation 3.6 and Equation 3.7, we then have

$$\omega\ v \to^* \sigma_n^{n+1} \dots \sigma_1^{n+1}\sigma_0^{n+1}(v_1')$$

We were looking for $\omega^\perp(\omega\ v)$: at the stage

$$\omega^\perp(\sigma_n^{n+1} \dots \sigma_1^{n+1}\sigma_0^{n+1}(v_1')),$$

the clause to fire is

$$\begin{pmatrix} v_1 \leftrightarrow \texttt{let}\, p_1 = \omega_1\ p_1'\ \texttt{in} \\ \quad \cdots \\ \quad \texttt{let}\, p_n = \omega_n\ p_n'\ \texttt{in}\ v_1' \end{pmatrix}^\perp := \begin{pmatrix} v_1' & \leftrightarrow & \texttt{let}\, p_n' = \omega_n^\perp\ p_n\ \texttt{in} \\ & & \cdots \\ & & \texttt{let}\, p_1' = \omega_1^\perp\ p_1\ \texttt{in}\ v_1 \end{pmatrix}.$$

Let $\Sigma = \sigma_n^{n+1} \dots \sigma_1^{n+1}\sigma_0^{n+1}$, we are therefore left to evaluate

$$\begin{pmatrix} v_1' & \leftrightarrow & \texttt{let}\, p_n' = \omega_n^\perp\ p_n\ \texttt{in} \\ & & \cdots \\ & & \texttt{let}\, p_1' = \omega_1^\perp\ p_1\ \texttt{in}\ v_1 \end{pmatrix}\ \Sigma(v_1')$$

We get $\Sigma[v_1'] = \Sigma(v_1')$. Hence, after applying the substitution we have :

$$\begin{array}{l} \texttt{let}\, p_n' = \omega_n^\perp\ \Sigma(p_n)\ \texttt{in} \\ \cdots \\ \texttt{let}\, p_1' = \omega_1^\perp\ \Sigma(p_1)\ \texttt{in}\ \Sigma(v_1) \end{array} \tag{3.9}$$

Note that in Σ the only substitution acting on p_n is σ_n^{n+1} the last one in the composition (since all the others can only have the variables in the p_j for $j < n$ in their support). But $\sigma_n^{n+1}(p_n)$ is exactly $\sigma_n(p_n) = \overline{v_n}$.

Recall that the rewriting shown in Equation 3.8 is part of the sequence of reduction in Equation 3.3. Then: either the number of IsoRec unfolding is smaller, or the size of the term is smaller. In both cases we can apply the induction hypothesis and deduce that

$$\omega_n^\perp \Sigma(p_n) = \omega_n^\perp \overline{v_n} \to^* (\sigma_{n-1}^n \dots \sigma_1^n\sigma_0^n(p_n')) \tag{3.10}$$

Therefore, rewriting Equation 3.9 eventually yields

$$\begin{array}{l} \texttt{let}\, p_{n-1}' = \omega_{n-1}^\perp\ (\sigma_{n-1}^n \dots \sigma_1^n\sigma_0^n\sigma_n^{n+1} \dots \sigma_1^{n+1}\sigma_0^{n+1})(p_{n-1})\ \texttt{in} \\ \dots \\ \texttt{let}\, p_1' = \omega_1^\perp\ \Sigma(p_1)\ \texttt{in}\ \Sigma(v_1) \end{array} \tag{3.11}$$

Note how in the sequence of compositions

$$\sigma_{n-1}^n \dots \sigma_1^n\sigma_0^n\sigma_n^{n+1} \dots \sigma_1^{n+1}\sigma_0^{n+1} \tag{3.12}$$

the only substitution with support matching the free variables of p_{n-1} are σ_{n-1}^n and σ_{n-1}^{n+1}: The composition in Equation 3.12 is σ_{n-1}. The term in Equation 3.11 is then

$$\begin{array}{l} \texttt{let}\, p_{n-1}' = \omega_{n-1}^\perp\ (\sigma_{n-1}(p_{n-1}))\ \texttt{in} \\ \dots \\ \texttt{let}\, p_1' = \omega_1^\perp\ \Sigma(p_1)\ \texttt{in}\ \Sigma(v_1). \end{array}$$

We can iterate the process by reducing $\omega_{n-1}^\perp\ (\sigma_{n-1}(p_{n-1})) = \omega_{n-1}^\perp\ \overline{v_{n-1}}$ using the induction hypothesis, and continue until we reach v_1. When we reach this level, the substitution is then σ_0 and we retrive $\sigma_0(v_1) = v$, as expected. □

Theorem 3.31 (Isos are isomorphisms)**.** *For all well-typed isos* $\vdash_i \omega : A \leftrightarrow B$*, and for all well-typed values* $\vdash_e v : A$*, if* $(\omega^\perp\ (\omega\ v)) \to^* v'$ *then* $v = v'$*.*

Proof. According to the evaluation strategy, we can decompose the rewriting of $\omega^\perp\ (\omega\ v)$ into two steps:

$$\omega^\perp\ (\omega\ v) \to^* \omega^\perp v_0 \tag{3.13}$$

followed by

$$\omega^\perp v_0 \to^* v'. \tag{3.14}$$

In Equation 3.13, we have $\omega\ v \to^* v_0$ a value. We can invoke Lemma 3.30 and get Equation 3.14 with $v' = v$, relying on the determinism of the rewriting procedure. □

4. Computational Content

In this section, we study the computational content of our language. Specifically, if we identify types with the set of their closed values —associating for instance $(\mathbb{1} \oplus \mathbb{1}) \otimes \mathbb{1}$ with the set of values $\{\langle \mathtt{inj}_l\ (), () \rangle, \langle \mathtt{inj}_r\ (), () \rangle\}$, closed isos $A \leftrightarrow B$ can be regarded as bijective maps between the corresponding sets of values. In the case of a closed iso $A \leftrightarrow B$ between types not involving the type constructor $\mu X.A$, any bijection between values of type A and values of type B can be represented with a pattern matching, one pattern for each closed value. However, with (infinite) inductive types, the expressivity becomes less clear as a bijection does not have a finite representation anymore. In this section, we show that we can encode the class of Recursive Primitive Permutations [PPR20] (RPP), implying that, although restricted, our language can express all primitive recursive functions [RJ87].

4.1. From RPP to Isos.

As the language RPP considers integers, we need to define a type to represent them. We start by defining the type of strictly positive natural numbers, $\mathtt{npos}$, as $\mathtt{npos} = \mu X.\mathbb{1} \oplus X$. We then define $\underline{n}$, the encoding of a positive natural number n into a value of type $\mathtt{npos}$ inductively, as

$$\underline{1} = \mathtt{fold}\ (\mathtt{inj}_l\ ()), \quad \underline{n+1} = \mathtt{fold}\ (\mathtt{inj}_r\ (\underline{n})).$$

Finally, we define the type of integers as $Z = \mathbb{1} \oplus (\mathtt{npos} \oplus \mathtt{npos})$ along with $\overline{z}$ the encoding of any $z \in \mathbb{Z}$ into a value of type Z defined as:

$$\begin{aligned}
&\overline{0} = \mathtt{inj}_l\ (), \\
&\overline{z} = \mathtt{inj}_r\ (\mathtt{inj}_l\ (\underline{z})) && \text{for } z \text{ positive,} \\
&\overline{z} = \mathtt{inj}_r\ (\mathtt{inj}_r\ (\underline{-z})) && \text{for } z \text{ negative.}
\end{aligned}$$

Given some function $f \in \mathrm{RPP}^k$, we show in the remainder of the section how to build an iso $\mathrm{isos}(f) : Z^k \leftrightarrow Z^k$ realizing f. The construction is defined inductively on the structure of f, as presented in Subsection 2.1.

4.1.1. *Encoding of Primitives.*

- The Successor is

$$\left\{ \begin{array}{ccc} \mathtt{inj}_l\ () & \leftrightarrow & \mathtt{inj}_r\ (\mathtt{inj}_l\ (\mathtt{fold}\ (\mathtt{inj}_l\ ()))) \\ \mathtt{inj}_r\ (\mathtt{inj}_l\ x) & \leftrightarrow & \mathtt{inj}_r\ (\mathtt{inj}_l\ (\mathtt{fold}\ (\mathtt{inj}_r\ x))) \\ \mathtt{inj}_r\ (\mathtt{inj}_r\ (\mathtt{fold}\ (\mathtt{inj}_l\ ()))) & \leftrightarrow & \mathtt{inj}_l\ () \\ \mathtt{inj}_r\ (\mathtt{inj}_r\ (\mathtt{fold}\ (\mathtt{inj}_r\ x))) & \leftrightarrow & \mathtt{inj}_r\ (\mathtt{inj}_r\ x) \end{array} \right\} : Z \leftrightarrow Z$$

- The Sign-change is

$$\left\{ \begin{array}{ccc} \mathtt{inj}_r\ (\mathtt{inj}_l\ x) & \leftrightarrow & \mathtt{inj}_r\ (\mathtt{inj}_r\ x) \\ \mathtt{inj}_r\ (\mathtt{inj}_r\ (x)) & \leftrightarrow & \mathtt{inj}_r\ (\mathtt{inj}_l\ x) \\ \mathtt{inj}_l\ () & \leftrightarrow & \mathtt{inj}_l\ () \end{array} \right\} : Z \leftrightarrow Z$$

- The identity is $\{x \leftrightarrow x\} : Z \leftrightarrow Z$.
- The Swap is $\{(x, y) \leftrightarrow (y, x)\} : Z^2 \leftrightarrow Z^2$.
- The Predecessor is the inverse of the Successor.

4.1.2. *Encoding of Horizontal and Vertical Composition.* Consider f and g two permutations in RPP^j, and write $\omega_f = \mathrm{isos}(f)$ and $\omega_g = \mathrm{isos}(g)$ for the isos encoding f and g. We encode the composition $\mathrm{isos}(f; g)$ of f and g with type $Z^j \leftrightarrow Z^j$ as:

$$\mathrm{isos}(f; g) = \left\{ \begin{array}{ccc} & & \mathtt{let}\ (y_1, \ldots, y_j) = \omega_f\ (x_1, \ldots, x_j)\ \mathtt{in} \\ (x_1, \ldots, x_j) & \leftrightarrow & \mathtt{let}\ (z_1, \ldots, z_j) = \omega_g\ (y_1, \ldots, y_j)\ \mathtt{in} \\ & & (z_1, \ldots, z_j) \end{array} \right\}$$

If now $f \in \mathrm{RPP}^j$ and $g \in \mathrm{RPP}^k$, and if $\omega_f = \mathrm{isos}(f)$ and $\omega_g = \mathrm{isos}(g)$, we encode the parallel composition $\mathrm{isos}(f \mid\mid g)$ of f and g with type $Z^{j+k} \leftrightarrow Z^{j+k}$ as:

$$\mathrm{isos}(f \mid\mid g) = \left\{ \begin{array}{ccc} & & \mathtt{let}\ (x'_1, \ldots, x'_j) = \omega_f\ (x_1, \ldots, x_j)\ \mathtt{in} \\ (x_1, \ldots, x_j, y_1, \ldots, y_k) & \leftrightarrow & \mathtt{let}\ (y'_1, \ldots, y'_k) = \omega_g\ (y_1, \ldots, y_k)\ \mathtt{in} \\ & & (x'_1, \ldots, x'_j, y'_1, \ldots, y'_k) \end{array} \right\}$$

4.1.3. *Encoding of Finite Iteration.* Consider $f \in \mathrm{RPP}^k$ and $\omega_f = \mathrm{isos}(f)$ its encoding. We encode the finite iteration $\mathbf{It}[f] \in \mathrm{RPP}^{k+1}$ with the help of the auxiliary iso ω_{aux}, of type $Z^k \otimes \mathtt{npos} \leftrightarrow Z^k \otimes \mathtt{npos}$ doing the finite iteration using $\mathtt{npos}$, defined as:

$$\omega_{\mathrm{aux}} = \mathbf{fix}\ g. \left\{ \begin{array}{ccc} (\overrightarrow{x}, \mathtt{fold}\ (\mathtt{inj}_l\ ())) & \leftrightarrow & \mathtt{let}\ \overrightarrow{y} = \omega_f\ \overrightarrow{x}\ \mathtt{in} \\ & & (\overrightarrow{y}, \mathtt{fold}\ (\mathtt{inj}_l\ ())) \\ (\overrightarrow{x}, \mathtt{fold}\ (\mathtt{inj}_r\ n)) & \leftrightarrow & \mathtt{let}\ (\overrightarrow{y}) = \omega_f\ (\overrightarrow{x})\ \mathtt{in} \\ & & \mathtt{let}\ (\overrightarrow{z}, n') = g\ (\overrightarrow{y}, n)\ \mathtt{in} \\ & & (\overrightarrow{z}, \mathtt{fold}\ (\mathtt{inj}_r\ n')) \end{array} \right\}$$

Given a pair (x, n), this iso iterates f $n+1$ times on x. We can now properly define $\mathrm{isos}(\mathbf{It}[f])$ of type $Z^{k+1} \leftrightarrow Z^{k+1}$ as:

$$\mathrm{isos}(\mathbf{It}[f]) = \left\{ \begin{array}{ccc} (\overrightarrow{x}, \mathtt{inj}_l\ ()) & \leftrightarrow & (\overrightarrow{x}, \mathtt{inj}_l\ ()) \\ (\overrightarrow{x}, \mathtt{inj}_r\ (\mathtt{inj}_l\ z)) & \leftrightarrow & \mathtt{let}\ (\overrightarrow{y}, z') = \omega_{aux}(\overrightarrow{x}, z)\ \mathtt{in} \\ & & (\overrightarrow{y}, \mathtt{inj}_r\ (\mathtt{inj}_l\ z')) \\ (\overrightarrow{x}, \mathtt{inj}_r\ (\mathtt{inj}_r\ z)) & \leftrightarrow & \mathtt{let}\ (\overrightarrow{y}, z') = \omega_{aux}(\overrightarrow{x}, z)\ \mathtt{in} \\ & & (\overrightarrow{y}, \mathtt{inj}_r\ (\mathtt{inj}_r\ z')) \end{array} \right\}$$

We simply have to perform a case distinction on whether the number of iterations is given as a positive, negative, or null value, and run ω_{aux} accordingly.

4.1.4. *Encoding of Selection.* Consider $f, g, h \in \mathrm{RPP}^k$ and their corresponding encoding $\omega_f = \mathrm{isos}(f), \omega_g = \mathrm{isos}(g)$ and $\omega_h = \mathrm{isos}(h)$. We define $\mathrm{isos}(\mathbf{If}[f, g, h])$ of type $Z^{k+1} \leftrightarrow Z^{k+1}$ as:

$$\mathrm{isos}(\mathbf{If}[f, g, h]) = \left\{ \begin{array}{ccc} (\overrightarrow{x}, \mathtt{inj}_r\ (\mathtt{inj}_l\ z)) & \leftrightarrow & \mathtt{let}\ \overrightarrow{x'} = \omega_f(\overrightarrow{x})\ \mathtt{in}\ (\overrightarrow{x'}, \mathtt{inj}_r\ (\mathtt{inj}_l\ z)) \\ (\overrightarrow{x}, \mathtt{inj}_l\ ()) & \leftrightarrow & \mathtt{let}\ \overrightarrow{x'} = \omega_g(\overrightarrow{x})\ \mathtt{in}\ (\overrightarrow{x'}, \mathtt{inj}_l\ ()) \\ (\overrightarrow{x}, \mathtt{inj}_r\ (\mathtt{inj}_r\ z)) & \leftrightarrow & \mathtt{let}\ \overrightarrow{x'} = \omega_h(\overrightarrow{x})\ \mathtt{in}\ (\overrightarrow{x'}, \mathtt{inj}_r\ (\mathtt{inj}_r\ z)) \end{array} \right\}$$

4.1.5. *Soundness of the Encoding.* In order to make sure that our encoding is sound, we need to make sure of two things: first that it is well-typed, and then that the semantics is preserved.

Theorem 4.1 (The encoding is well-typed)**.** *Let $f \in \mathrm{RPP}^k$, then $\vdash_i\ \mathrm{isos}(f) : Z^k \leftrightarrow Z^k$.*

Proof. By induction on f, for the two compositions, iteration, and selection the variables $\overrightarrow{x}$ are all of type Z, while for ω_{aux} the last argument is of type `npos`. The predicate $\mathtt{OD}_Z$ is always satisfied as the left columns of the n-fold tensor are always variables and the right-most argument on each isos always satisfies $\mathtt{OD}_Z$. □

Theorem 4.2 (Simulation)**.** *Let $f \in \mathrm{RPP}^k$ and $n_1, \ldots, n_k$ elements of $\mathbb{Z}$ such that $f(n_1, \ldots, n_k) = (m_1, \ldots, m_k)$ then* $\mathrm{isos}(f)(\overline{n_1}, \ldots, \overline{n_k}) \to^* (\overline{m_1}, \ldots, \overline{m_k})$

Proof. By induction on f.

- Direct for the identity, swap and sign-change.
- For the Successor:

$$\omega = \left\{ \begin{array}{ccc} \mathtt{inj}_l\ () & \leftrightarrow & \mathtt{inj}_r\ (\mathtt{inj}_l\ (\mathtt{fold}\ (\mathtt{inj}_l\ ()))) \\ \mathtt{inj}_r\ (\mathtt{inj}_l\ (x)) & \leftrightarrow & \mathtt{inj}_r\ (\mathtt{inj}_l\ (\mathtt{fold}\ (\mathtt{inj}_r\ (x)))) \\ \mathtt{inj}_r\ (\mathtt{inj}_r\ (\mathtt{fold}\ (\mathtt{inj}_l\ ()))) & \leftrightarrow & \mathtt{inj}_l\ () \\ \mathtt{inj}_r\ (\mathtt{inj}_r\ (\mathtt{fold}\ (\mathtt{inj}_r\ x))) & \leftrightarrow & \mathtt{inj}_r\ (\mathtt{inj}_r\ (x)) \end{array} \right\}$$

  we do it by case analysis on the sole input n.
  - Case $n = 0$ then $\overline{0} = \mathtt{inj}_l\ ()$ and $\omega\ \mathtt{inj}_l\ () \to \mathtt{inj}_r\ (\mathtt{inj}_l\ (\mathtt{fold}\ (\mathtt{inj}_l\ ()))) = \overline{1}$.
  - Case $n = -1$ then $\overline{-1} = \mathtt{inj}_r\ (\mathtt{inj}_r\ (\mathtt{fold}\ (\mathtt{inj}_l\ ())))$, so the term reduces to $\mathtt{inj}_l\ () = \overline{0}$.
  - Case $n < -1$, we have $\overline{n} = \mathtt{inj}_r\ (\mathtt{inj}_r\ (\mathtt{fold}\ (\mathtt{inj}_r\ \underline{n'})))$ with $\underline{n'} = \underline{-(n+1)}$, by pattern-matching we get $\mathtt{inj}_r\ (\mathtt{inj}_r\ \underline{n'})$ which is $\overline{-(n+1)}$.
  - Case $n > 1$ is similar.
- The Predecessor is the dual of the Successor.

- **Composition & Parallel composition**: Direct by induction hypothesis on ω_f and ω_g.: for the composition, ω_f is first applied on all the input and then ω_g on the result of ω_f. For the parallel composition, ω_f is applied on the first j arguments and ω_g on the argument $j+1$ to k before concatenating the results from both isos.
- **Finite Iteration**: $\mathrm{It}[f]$.

  We need the following lemma: $\omega_{\text{aux}}(\bar{x_1}, \dots, \bar{x_n}, \underline{z}) \to^* (\bar{z_1}, \dots, \bar{z_n}, \underline{z})$ where z is a non-zero integer and $(z_1, \dots, z_n) = f^{|z|}(x_1, \dots, x_n)$ which can be shown by induction on $\mid z \mid$: the case $z = 1$ and $\bar{z} = \texttt{fold inj}_l\ ()$ is direct by induction hypothesis on ω_f. Then if $z = n+1$ we get it directly by induction hypothesis on both ω_f and our lemma.

  Then, for $\mathrm{isos}\,\mathrm{It}[f]$ we do it by case analysis on the last argument: when it is $\bar{0}$ then we simply return the result, if it is $\bar{z}$ for z (no matter if strictly positive or strictly negative) then we enter ω_{aux}, and apply the previous lemma.
- **Conditional**: $\mathrm{If}[f, g, h]$. Direct by case analysis of the last value and by induction hypothesis on $\omega_f, \omega_g, \omega_h$. □

Theorem 2.3 and Theorem 4.2 tell us that any primitive recursive function can effectively be encoded as a well-typed iso with the appropriate semantics. It is in particular possible to encode the Cantor-Pairing [PPR20, Theorem 2 and Theorem 4], although with auxiliary arguments. Another representation of the Cantor Pairing, without auxiliary arguments, would be possible in our language but would require a more lax notion of structural recursion. This is discussed in Section 6.

Also note that $\mathrm{isos}(f)^\perp \neq \mathrm{isos}(f^{-1})$. Indeed, $\mathrm{isos}(f)^\perp$ inverses the order of the `let` constructions, whereas this will not happen for $\mathrm{isos}(f^{-1})$. The two can nonetheless be considered equivalent up to a permutation of `let` constructions and renaming of variables. Theorem 4.2 also confirms that they do indeed have the same behavior.

5. Proof-Theoretical Content

In the present section, we want to relate our language of isos to proofs in a μMALL. As mentioned earlier, an iso $\vdash_i\ \omega : A \leftrightarrow B$ corresponds to both a computation sending a value of type A to a result of type B and a computation sending a value of type B to a result of type A, inverse of each other. Under the principles of Curry-Howard correspondence, an iso should therefore correspond to a *proof* isomorphism : the data of two proofs, π of $A \vdash B$ and $\pi^\perp$ of $B \vdash A$, which are inverse of each other in the sense that, when cut together behave like an (expansion of the) axiom rule, in the sense that cut with any proof π' of $\vdash A$ (resp. $\vdash B$), cut-elimination returns the proof π' unchanged, corresponding to Theorem 3.31.

We will rely on the infinite derivations available in μMALL to obtain a proper representation of the recursive behaviours available in our iso. The main difficulty will consist in ensuring that any pre-proof of μMALL obtained from an iso will indeed satisfy μMALL validity criterion (Subsection 2.2).

5.1. Translating isos into μMALL.

We start by giving the translation from isos to pre-proofs, and then show that they are actually proofs, therefore obtaining a *static* correspondence between reversible programs and μMALL proofs. We then show that our translation entails the expected *dynamic* correspondence between the evaluation procedure of our language and μMALL cut-elimination procedure. The derivations we obtain are *circular*, and we therefore translate an iso ω directly into

a finite derivation with back-edges, written $\texttt{Circ}(\omega)$. A translation to infinite derivations can straightforwardly be obtained by composing $\texttt{Circ}(-)$ with the unfolding operation.

In this translation, we need to manage a correspondence between typing contexts which are sets of variables with associated types and sequent contexts which are lists of formulas. In order to do so, we assume fixed once and for all a linear order on variables (for instance given by an arbitrary enumeration of the set of variables) and will consider that to a typing context Θ one associates the list made of the formulas appearing in the typing contexts following the order given by the variable ordering, written $\overline{\Theta}$.

5.1.1. *Definition of the Translation.* Given an iso $\omega : A \leftrightarrow B$, its translation into a μMALL derivation of $A \vdash B$ is described with three separate phases:

***Iso Phase*:** The first phase consists in traveling through the syntactical definition of an iso, keeping as information the last encountered iso-variable bounded by a **fix** -construction, if any. When an iso of the shape $\{v_1 \leftrightarrow e_1 \mid \ldots \mid v_n \leftrightarrow e_n\}$ of type $A \leftrightarrow B$ is encountered, the negative phase starts creating a sequent $A \vdash^f B$ labeled with the last encountered iso-variable. One will resume this phase when encountering another iso in one of the e_i; a back-edge is created in case of an iso variable.

***Negative Phase*:** The negative phase is guided by a list of formulas to be decomposed using the reversible rules of the logic beginning with a singleton list $[A]$ that corresponds to the decomposition of formula A according to how values of type A on the left-hand-side of ω are pattern-matched. The negative phase relies on the $\texttt{Neg}(\cdot)$ function having 5 arguments:

- A set of pairs of lists of values and typing derivation, written (l, ξ) where each element of the set corresponds to one clause $v \leftrightarrow e$ of the given iso and ξ is the typing derivation of e. The list of values corresponds to what is left to be decomposed in the left-hand-side of the clause (for instance if v is a pair $\langle v_1, v_2\rangle$ the list will have two elements to decompose).
- A list of formulas Γ containing the formulas being decomposed. Intuitively, the ith formula of the list corresponds to the type of the ith value of the list l.
- A context Δ, containing the formulas that have already been decomposed and will no longer be decomposed during the negative phase. This context can only grow during the negative phase: once a formula is in it, it will no longer be modified until the end of the negative phase.
- A formula A which corresponds to the right-hand-side formula of the conclusion sequent (or rather to the single positive formula) of the derivation that we are currently building, which will not be modified during the negative phase.
- a (potentially) annotated sequent $\vdash^f$, which will be used as the target of a back-edge.

Therefore $\texttt{Neg}(\{(l_i, \xi_i) \mid i \in I\}, \Gamma, \Delta, A, \vdash^a)$ will build a derivation $\Gamma, \Delta \vdash^a A$ by decomposing the formulas in Γ accordingly to the values present in l. The negative phase ends when the list is empty and hence when $\Gamma = []$. When it is the case, we can start translating ξ and the *positive phase* start. The negative phase is defined inductively on the first element of the list of every set, which are known by typing to have the same prefix, and is given in Figure 4.

***Positive phase*:** The translation of an expression is pretty straightforward: each *let* and iso-application is represented by two `cut` rules, as usual in Curry-Howard correspondence. The second argument is a list of formulas for which we maintain the following invariant at each call to $\texttt{Pos}(\xi, \Gamma)$: Γ is a list of formulas corresponding to an ordering of the types declared in the typing context of ξ. This is useful not only to generate the μMALL derivation but also to

$$\texttt{Neg}(\{(\texttt{inj}_l\ v_j :: l_j, \xi_j) \mid j \in J\} \cup \{(\texttt{inj}_r\ v_k :: l_k, \xi_k) \mid k \in K\}, A_1 \oplus A_2 :: \Gamma, \Delta, B, \vdash^\Psi) =$$

$$\frac{\texttt{Neg}(\{(v_j :: l_j, \xi_j) \mid j \in J\}, A_1 :: \Gamma, \Delta, B, \vdash) \qquad \texttt{Neg}(\{(v_k :: l_k, \xi_k) \mid k \in K\}, A_2 :: \Gamma, \Delta, B, \vdash)}{A_1 \oplus A_2, \Gamma, \Delta \vdash^\Psi B}\ \&$$

$$\texttt{Neg}(\{(() :: l_i, \xi_i) \mid i \in I\}, \mathbb{1} :: \Gamma, \Delta, B, \vdash^\Psi) = \frac{\texttt{Neg}(\{(l_i, \xi_i) \mid i \in I\}, \Gamma, \Delta, B, \vdash)}{\mathbb{1}, \Gamma, \Delta \vdash^\Psi B}\ \bot$$

$$\texttt{Neg}(\{(\langle v_i^1, v_i^2\rangle :: l_i, \xi_i) \mid i \in I\}, A_1 \otimes A_2 :: \Gamma, \Delta, B, \vdash^\Psi) = \frac{\texttt{Neg}(\{(v_i^1 :: v_i^2 :: l_i, \xi_i) \mid i \in I\}, A_1 :: A_2 :: \Gamma, \Delta, B, \vdash)}{A_1 \otimes A_2, \Gamma, \Delta \vdash^\Psi B}\ \parr$$

$$\texttt{Neg}(\{(\texttt{fold}\ v_i :: l_i, \xi_i) \mid i \in I\}, \mu X.A :: \Gamma, \Delta, B, \vdash^\Psi) = \frac{\texttt{Neg}(\{(v_i :: l_i, \xi_i) \mid i \in I\}, A[X \leftarrow \mu X.A] :: \Gamma, \Delta, B, \vdash)}{\mu X.A, \Gamma, \Delta \vdash^\Psi B}\ \nu$$

$$\texttt{Neg}(\{(x :: l_i, \xi_i) \mid i \in I\}, A :: \Gamma, \Delta, B, \vdash^\Psi) = \frac{\texttt{Neg}(\{(l_i, \xi_i) \mid i \in I\}, \Gamma, A :: \Delta, B, \vdash)}{A, \Gamma, \Delta \vdash^\Psi B}\ ex$$

$$\texttt{Neg}(\{([], \xi)\}, [], \Delta, B, \vdash^\Psi) = \texttt{Pos}(\xi, \Delta)$$

Figure 4: Negative Phase.

manage the sequent contexts correctly when needing to split it, for the $\otimes$ rule for instance. The definition of the positive phase is given in Figure 5, where in the last rule we have $|(p, A)| = ([x_1, \ldots, x_n], [A_1, \ldots, A_n])$ as defined in Subsubsection 3.2.1.

Definition 5.1 (Translation of Isos into μMALL)**.** Given a well-typed iso $\omega : A \leftrightarrow B$, its translation into μMALL, $\texttt{Circ}(\omega)$ is defined altogether with the functions `Neg` and `Pos` by mutual recursion and produces a circular derivation of μMALL. $\texttt{Circ}(\omega)$ is defined below while `Neg` and `Pos` are defined in Figure 4 and Figure 5 respectively:

- $\texttt{Circ}(f : A \leftrightarrow B \vdash f : A \leftrightarrow B) = \dfrac{}{A \vdash B}\ be(f)$
- $\texttt{Circ}(\Psi \vdash_i \{(v_i \leftrightarrow e_i)_{i \in I}\} : A \leftrightarrow B) = \texttt{Neg}(([v_i], \xi_i)_{i \in I}, [A], \emptyset, B, \vdash)$
- $\texttt{Circ}(\Psi \vdash_i \textbf{fix}\ f.\{(v_i \leftrightarrow e_i)_{i \in I}\} : A \leftrightarrow B) = \texttt{Neg}(([v_i], \xi_i)_{i \in I}, [A], \emptyset, B, \vdash^f)$

  Where the ξ_i are the typing derivations of the e_i.

Example 5.2. The translation $\pi = \texttt{Circ}(\omega)$ of the iso ω from Example 3.15 is:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{A \vdash A}\ \texttt{id}}{A \vdash A \oplus B}\ \oplus^1}{A \vdash C \oplus (A \oplus B)}\ \oplus^2 \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{B \vdash B}\ \texttt{id}}{B \vdash A \oplus B}\ \oplus^2}{B \vdash C \oplus (A \oplus B)}\ \oplus^2 \qquad \dfrac{\dfrac{}{C \vdash C}\ \texttt{id}}{C \vdash C \oplus (A \oplus B)}\ \oplus^1}{B \oplus C \vdash C \oplus (A \oplus B)}\ \&}{A \oplus (B \oplus C) \vdash C \oplus (A \oplus B)}\ \&$$

Example 5.3. Considering the iso swap of type $A \otimes B \leftrightarrow B \otimes A$ defined as $\text{swap} = \{\langle x, y\rangle \leftrightarrow \langle y, x\rangle\}$, its corresponding proof is the proof π_S given in Example 2.19 whereas the iso $\text{map}(\text{swap})$ as defined by Example 3.16 have for corresponding proof the proof $\phi(\pi_S)$ from Example 2.19. Notice that the blue thread follows the focus of the structurally recursive criterion. The *negative phase* consist of the $\nu, \&, \parr$ and $\bot$ rules, where the pre-thread is going up, while the *positive phase* consist of the multiple cut-rules where the pre-thread is not active.

Lemma 5.4. *Given an iso $\vdash_i\ \omega : A \leftrightarrow B$, and given $\pi = \texttt{Circ}(\omega)$, for each infinite branch of π, only one sequent tagged by an iso-variable is visited infinitely often.*

$$\mathtt{Pos}\left(\ \overline{\vdash_e () : \mathbb{1}, \emptyset}\ \right) = \overline{\vdash \mathbb{1}}\ \mathbb{1}$$

$$\mathtt{Pos}\left(\ \overline{x : A \vdash_e x : A, [A]}\ \right) = \overline{A \vdash A}\ \mathtt{id}$$

$$\mathtt{Pos}\left(\ \frac{\xi : (\Theta \vdash_e t : A_1)}{\Theta \vdash_e \mathtt{inj}_l\ t : A_1 \oplus A_2, \Gamma}\ \right) = \frac{\mathtt{Pos}(\xi, \Gamma)}{\Gamma \vdash A_1 \oplus A_2}\ \oplus^1$$

$$\mathtt{Pos}\left(\ \frac{\xi : (\Theta \vdash_e t : A_2)}{\Theta \vdash_e \mathtt{inj}_r\ t : A_1 \oplus A_2, \Gamma}\ \right) = \frac{\mathtt{Pos}(\xi, \Gamma)}{\Gamma \vdash A_1 \oplus A_2}\ \oplus^2$$

$$\mathtt{Pos}\left(\ \frac{\xi : (\Theta \vdash_e t : A[X \leftarrow \mu X.A])}{\Theta \vdash_e \mathtt{fold}\ t : \mu X.A}, \Gamma\ \right) = \frac{\mathtt{Pos}(\xi, \Gamma)}{\Gamma \vdash \mu X.A}\ \mu$$

$$\mathtt{Pos}\left(\ \frac{\xi_1 : (\Theta_1 \vdash_e t_1 : A_1) \qquad \xi_2 : (\Theta_2 \vdash_e t_2 : A_2)}{\Theta_1, \Theta_2 \vdash_e \langle t_1, t_2 \rangle : A_1 \otimes A_2}, \Gamma\ \right) = \frac{\dfrac{\mathtt{Pos}(\xi_1, \overline{\Theta_1}) \qquad \mathtt{Pos}(\xi_2, \overline{\Theta_2})}{\overline{\Theta_1}, \overline{\Theta_2} \vdash A_1 \otimes A_2}\ \otimes}{\Gamma \vdash A_1 \otimes A_2}\ \mathsf{ex}^\star$$

$$\mathtt{Pos}\left(\ \frac{\Psi \vdash_i \omega : A \leftrightarrow B \qquad \xi : (\Theta \vdash_e t : A)}{\Theta; \Psi \vdash_e \omega\ t : B}, \Gamma\ \right) = \frac{\mathtt{Pos}(\xi, \Gamma) \qquad \mathtt{Circ}(\omega)}{\Gamma \vdash B}\ \mathtt{cut}$$

$$\mathtt{Pos}\left(\ \frac{\xi_1 : (\Theta_1 \vdash_e t_1 : A) \qquad \xi_2; (\Theta_2, x_1 : A_1, \ldots, x_n : A_n \vdash_e t_2 : B)}{\Theta_1, \Theta_2 \vdash_e \mathtt{let}\, p = t_1\ \mathtt{in}\ t_2 : B}, \Gamma\ \right) =$$

$$\frac{\dfrac{\mathtt{Pos}(\xi_1, \overline{\Theta_1}) \qquad \mathtt{Neg}(\{([x_1, \ldots, x_n], \xi_2)\}, [A_1 \otimes \cdots \otimes A_n], \overline{\Theta_2}, B, \vdash)}{\overline{\Theta_1}, \overline{\Theta_2} \vdash B}\ \mathtt{cut}}{\Gamma \vdash B}\ \mathsf{ex}^\star$$

Figure 5: Positive Phase.

Proof. Since we have at most one iso-variable, we never end up in the case that between an annotated sequent $\vdash^f$ and a back-edge pointing to f we encounter another annotated sequent. □

Among the terms that we translate, the translation of a value yields what we call a *purely positive proof* which is trivially a valid pre-proof.

Definition 5.5 (Purely Positive Proof)**.** A *purely positive proof* is a finite, cut-free proof whose rules are only $\oplus^i, \otimes, \mu, \mathsf{ex}, \mathbb{1}$, *id* for $i \in \{1, 2\}$.

Lemma 5.6 (Values are Purely Positive Proofs)**.** *Given* $\Theta = \{x_1 : A_1, \ldots, x_n : A_n\}$, $\xi : (\Theta \vdash v : A)$ *and* Δ *a list of formulas corresponding to* Θ *then* $\mathtt{Pos}(\xi, \Delta)$ *is a purely positive proof.*

Proof. By induction on the structure of ξ:

- $x : A \vdash_e x : A$ then the derivation is $\overline{A \vdash A}\ \mathtt{id}$, which is a purely positive proof;
- $\vdash () : \mathbb{1}$ then the derivation is $\overline{\vdash \mathbb{1}}\ \mathbb{1}$, which is a purely positive proof;
- $\Theta_1, \Theta_2 \vdash \langle v_1, v_2 \rangle : A \otimes B$ then we get $\dfrac{\dfrac{\dfrac{\pi_1}{\overline{\Theta_1} \vdash A} \qquad \dfrac{\pi_2}{\overline{\Theta_2} \vdash B}\ \otimes}{\overline{\Theta_1}, \overline{\Theta_2} \vdash A \otimes B}}{\Delta \vdash A \otimes B}\ \mathsf{ex}^\star$ and then by induction hypothesis on the typing derivations ξ_1 and ξ_2 of v_1 and v_2 we get the expected result;
- $\Theta \vdash \mathtt{inj}_l\ v : A \oplus B$ then the derivation is $\dfrac{\dfrac{\pi}{\Delta \vdash A}}{\Delta \vdash A \oplus B}\ \oplus^1$ then by induction hypothesis on the typing derivation ξ of v we get the desired result;
- The proof is similar for $\mathtt{inj}_r\ v$ and $\mathtt{fold}\ v$. □

The converse is also true: any purely positive proof describes a unique value of the language (up to α-equivalence).

The well-definedness of $\texttt{Circ}(\omega)$ relies on guaranteeing the following invariant upon each call to function $\texttt{Pos}(\xi, \Gamma)$: Γ *is a list corresponding to some ordering of the formulas occurring in the typing context of* ξ.

This property is trivially preserved upon each recursive call to $\texttt{Pos}(\cdot)$ in Figure 5. Therefore, the only thing to check is that at the initial call to $\texttt{Pos}(\cdot)$, in the last case of the definition of $\texttt{Neg}(\cdot)$ in Figure 4, the invariant is satisfied.

Proposition 5.7. *Let* $\omega = \{v_1 \leftrightarrow e_1 \mid \ldots \mid v_n \leftrightarrow e_n\} : A \leftrightarrow B$ *(resp.* $\omega = \textbf{fix}\ f.\{v_1 \leftrightarrow e_1 \mid \ldots \mid v_n \leftrightarrow e_n\} : A \leftrightarrow B$*) be a well-typed iso of typing derivation* ξ_ω *and let* $(\Theta_i)_{1\leq i\leq n}$ *be the typing contexts given by* ξ_ω *such that* $\Theta_i \vdash_e v_i : A$ *and* $\Theta_i \vdash_e e_i : B$ *for* $1 \leq i \leq n$*. Then for any* $(\{([v_i^1, \ldots, v_i^k], \xi_i), i \in I\}, [F_1, \ldots F_k], \Gamma, B, \vdash^{\Psi'})$ *that is the argument of a recursive call to* $\texttt{Neg}(\cdot)$ *from* $\texttt{Neg}(\{([v_i], \xi_i), 1 \leq i \leq n\}, [\underline{A}], \emptyset, B, \vdash^{\Psi})$ *is such that there exists typing contexts* $\Theta_i^j, 1 \leq j \leq k$ *such that* $\Theta_i^j \vdash v_i^j : F_j$ *and* $\Gamma, \overline{\Theta_i^1}, \ldots, \overline{\Theta_i^n}$ *is a reordering of the list* $\overline{\overline{\Theta_i}}$.

Proof. In the following proof, when this does not introduce ambiguity, we will indulge ourselves in a slight abuse of notation, writing Θ instead of $\overline{\Theta}$ to keep the notations light. The property is proved by induction on the number n of recursive calls needed to reach

$$(\{([v_i^1, \ldots, v_i^k], \xi_i), i \in I\}, [F_1, \ldots F_k], \Gamma, B)$$

in the (finite) tree of recursive calls from

$$\texttt{Neg}(\{([v_i], \xi_i), 1 \leq i \leq n\}, [A], \emptyset, B).$$

- If $n = 0$, then the property is trivial since $k = 1, \Gamma = \emptyset$: it is sufficient to set $\Theta_i^1 = \Theta_i$.
- Assuming that the property holds for any $i \leq n$ and assuming $\texttt{Neg}(\{([v_i^1, \ldots, v_i^k], \xi_i), i \in I\}, [F_1, \ldots F_k], \Gamma, B, \vdash^{\Psi'})$ is a subtree of height $n + 1$. We reason by case on the last rule the derivation π of which $\texttt{Neg}(\{([v_i^1, \ldots, v_i^k], \xi_i), i \in I\}, [F_1, \ldots F_k], \Gamma, B, \vdash^{\Psi'})$ is a premise.
  - Case of $\parr$: π is some derivation to which the induction hypothesis applies and, by definition of $\texttt{Neg}(\cdot)$ it is of the form $\texttt{Neg}(\{([\langle v_i^1, v_i^2\rangle, v_i^3, \ldots v_i^k], \xi_i), i \in I\}, [F_1 \otimes F_2, F_3, \ldots F_k], \Gamma, B, \vdash^{\Psi'})$. By induction hypothesis, there exist Θ^0 and $\Theta_i^j, 3 \leq j \leq k$ such that (i) $\Theta^0 \vdash \langle v_i^1, v_i^2\rangle : F_1 \otimes F_2$, (ii) $\Theta_i^j \vdash v_i^j : F_j$ for $3 \leq j \leq k$ and $\Gamma, \Theta^0, \Theta_i^3, \ldots, \Theta_i^n$ is a reordering of Θ_i.
    Since $\langle v_i^1, v_i^2\rangle$ is a well-typed value, there exists Θ_i^1, Θ_i^2 such that $\Theta^0 = \Theta_i^1, \Theta_i^2$ and $\Theta_i^1 \vdash v_i^1 : F_1$ and $\Theta_i^2 \vdash v_i^2 : F_2$. As a consequence, $\Gamma, \Theta_i^1, \ldots, \Theta_i^n$ is a reordering or Θ_i and $\Theta_i^j \vdash v_i^j : F_j$ for $1 \leq j \leq k$ as expected.
  - Case of $\&$. Wlog we assume that the premise we are considering is the left premise of the $\&$. Then, π is of the form $\texttt{Neg}(\{([\texttt{inj}_l\ v_i^1, v_i^2, \ldots, v_i^k], \xi_i), i \in K\} \cup \{(\texttt{inj}_r\ v_j :: l_j, \xi_j), j \in J\}, [F_1 \oplus F_1', F_2, \ldots F_k], \Gamma, B, \vdash^{\Psi'})$. By induction hypothesis, there exists $\Theta_i^j, 1 \leq j \leq k$ such that (i) $\Theta_i^1 \vdash \texttt{inj}_l\ v_i^1 : F_1 \oplus F_1'$, (ii) $\Theta_i^j \vdash v_i^j : F_j$ for $2 \leq j \leq k$ and $\Gamma, \Theta_i^1, \Theta_i^2, \ldots \Theta_i^n$ is a reordering of Θ_i.
    Since $\texttt{inj}_l\ v_i^1$ is a well-typed value, $\Theta_i^1 \vdash v_i^1 : F_1$ and $\Gamma, \Theta_i^1, \ldots, \Theta_i^n$ is a reordering of Θ_i and $\Theta_i^j \vdash v_i^j : F_j$ for $1 \leq j \leq k$ as expected. The case of the right premise of a $\&$ is similar.
  - The ν rule is similar to the above case, just adapted to the fold constructor. Then, π is of the form $\texttt{Neg}(\{([\texttt{fold}\ v_i^1, v_i^2, \ldots, v_i^k], \xi_i), i \in I\}, [\mu X.F, F_2, \ldots F_k], \Gamma, B, \vdash^{\Psi'})$, with $F_1 = F[X \leftarrow \mu X.F]$. By induction hypothesis, there exists $\Theta_i^j, 1 \leq j \leq k$ such that (i) $\Theta_i^1 \vdash \texttt{fold}\ v_i^1 : \mu X.F$, (ii) $\Theta_i^j \vdash v_i^j : F_j$ for $2 \leq j \leq k$ and $\Gamma, \Theta_i^1, \ldots, \Theta_i^n$ is a reordering of Θ_i.

Since `fold` v_i^1 is a well-typed value, $\Theta_i^1 \vdash v_i^1 : F[X \leftarrow \mu X.F]$ and finally $\Gamma, \Theta_i^1, \dots, \Theta_i^n$ is a reordering of Θ_i and $\Theta_i^j \vdash v_i^j : F_j$ for $1 \leq j \leq k$ as expected.

– Case of $\top$. Then, π is of the form $\texttt{Neg}(\{([(), v_i^1, \dots v_i^k], \xi_i), i \in I\}, [\mathbb{1}, F_1, F_2, \dots F_k], \Gamma, B, \vdash^{\Psi'})$ and induction hypothesis ensures that there exists $\Theta_i^j, 0 \leq j \leq k$ such that (i) $\Theta_i^0 \vdash () : \mathbb{1}$, (ii) $\Theta_i^j \vdash v_i^j : F_j$ for $1 \leq j \leq k$ and $\Gamma, \Theta_i^0, \dots, \Theta_i^n$ is a reordering of Θ_i.
By the typing rule for value $()$ we have that Θ_i^0 is empty from which we can conclude, as expected, that $\Gamma, \Theta_i^1, \dots, \Theta_i^n$ is a reordering of Θ_i.

– Case of ex. In this case, π is of the form

$$\texttt{Neg}(\{([x, v_i^1, \dots, v_i^k], \xi_i), i \in I\}, [F, F_1, F_2, \dots, F_k], \Gamma', B, \vdash^{\Psi'})$$

and induction hypothesis ensuring that there exists $\Theta_i^j, 0 \leq j \leq k$ such that (i) $\Theta_i^0 \vdash x : F$, (ii) $\Theta_i^j \vdash v_i^j : F_j$ for $1 \leq j \leq k$ and $\Gamma', \Theta_i^0, \dots, \Theta_i^n$ is a reordering of Θ_i.
By the typing rule for value x we have that $\Theta_i^0 = (x : F)$ from which we can conclude, as expected, that $\Gamma, \Theta_i^1, \dots, \Theta_i^n$ is a reordering of Θ_i. since $F, \Gamma' = \Gamma$.

□

We can now show that our translation is well-defined:

Proposition 5.8. *Given a closed iso* $\vdash_i \ \omega$ *then* $\texttt{Circ}(\omega, \emptyset)$ *is well-defined.*

Proof. By induction on $\vdash_i$. The iso is of the form $\textbf{fix}\ f.\{v_1 \leftrightarrow e_1 \mid \dots \mid v_n \leftrightarrow e_n\}$ or $\{v_1 \leftrightarrow e_1 \mid \dots \mid v_n \leftrightarrow e_n\}$ and we end up in the case $\texttt{Circ}(\{(v_i \leftrightarrow e_i')_{i \in I}\} : A \leftrightarrow B, \vdash^f)$ – where the root of the derivation takes place, annotated with f – or with $\texttt{Circ}(\{(v_i \leftrightarrow e_i')_{i \in I}\} : A \leftrightarrow B, \vdash)$ where the root of the derivation is not annotated with any label.

The negative phase $\texttt{Neg}(([v_i], \xi_i')_{i \in I}, [A], \emptyset, B, \vdash^{\Psi})$ is well-defined due to the predicate $\texttt{OD}_A$. Indeed, whenever $\texttt{OD}_A(S)$ holds on a well-typed set of values S of the same type, all the values in S necessarily have the same constructors at the root position.

The positive phase is well-defined as it consists in recreating the typing judgment of each expression e_i. By Proposition 5.7 we know that the typing context of e_i and the one obtained from the negative phase contain the same variable associated to the same formula. □

5.1.2. *Validity of the Translation.* We now turn to the validity of the μMALL proof corresponding to a well-typed iso. Given an iso $\omega = \textbf{fix}\ f.\{v_1 \leftrightarrow e_1 \mid \dots \mid v_n \leftrightarrow e_n\}$, we want to show that any infinite branch is inhabited by a valid thread.

Before going further, it is useful to notice that the bouncing thread we are encountering in translations of isos have a specific shape, the reason for which we introduce the notion of *bouncing-cut* and their origin:

Definition 5.9 (Bouncing-Cut)**.** A Bouncing-cut is a cut of the form: $\dfrac{\dfrac{\pi}{\Gamma \vdash G} \qquad \dfrac{}{G \vdash F}\ be(f)}{\Gamma \vdash F}\ \texttt{cut}$

Due to the syntactical restrictions of the language we get straightforwardly the following fact:

Proposition 5.10 (Origin of Bouncing-Cut)**.** *Given a well-typed iso, every occurrence of a rule* $be(f)$ *in* $\texttt{Circ}(\omega)$ *is a premise of a bouncing-cut.*

In particular, when following a thread going up into a *bouncing-cut*, it will always start from some negative formula (represented graphically as a positive formula in the left-hand-side of the

sequent with our writing convention), before going back down following a positive formula (represented as *the only* right-hand-side formula of the sequent) and bouncing back up on the bouncing-cut to reach immediately the back-edge.

As given by Lemma 5.4, an infinite branch is uniquely defined by a single iso-variable. Given the value v_i^j that is the decreasing argument for the structurally recursive criterion, we want to build a pre-thread that follows the variable $x_j : \mu X.B$ in $v_i^j : \mu X.B$ that is the focus of the criterion.

Since we want to show that $\texttt{Circ}(\omega)$ is a valid derivation, we need to show that for every infinite branch, there exists a valid thread. Among the multiple threads that exist inside an infinite branch, we look at the one which follows the formula corresponding to the decreasing argument of the structurally recursive criterion. We describe in the following definitions how this thread behaves inside $\texttt{Circ}(\omega)$. But first, notice that (i) the base case of $\texttt{Neg}(-)$ is necessarily a call to $\texttt{Pos}(-)$ and (ii) when we have a set of clauses $\{v_1 \leftrightarrow e_1 \mid \ldots \mid v_n \leftrightarrow e_n\}$ there will be exactly n calls to $\texttt{Pos}(-)$ for each typing derivation of each e_i. This is due to the fact that branching occurs for each occurrence of a left or right injection in the v_i.

Definition 5.11 (Pre-Thread of the negative phase)**.** Given a well-typed iso $\omega = \textbf{fix}\ f.\{v_1 \leftrightarrow e_1 \mid \ldots \mid v_n \leftrightarrow e_n\} : A \leftrightarrow B$ and a clause $v_i \leftrightarrow e_i$ such that $f\ p$ is a subterm of e_i and the variable x of type $\mu X.C$ which is the focus of the primitive recursive criterion. Given that x occurs free only in the clause $v_i \leftrightarrow e_i$, we define $PT_n(x, (v_i \leftrightarrow e_i), \texttt{Circ}(\omega))$ as the pre-thread that follows the subformula $\mu X.C$ until $\texttt{Pos}(\xi_i)$ is reached, where ξ_i is the typing derivation of e_i

Now that we have defined the pre-thread which follows the decreasing argument during the negative phase, we can look at its weight.

Lemma 5.12 (Weight of the pre-thread for the negative phase)**.** *Given a well-typed iso $\omega = \textbf{fix}\ f.\omega$ with a clause $v \leftrightarrow e$ where x is the focus, then $w(PT_n(x, (v \leftrightarrow e), \texttt{Circ}(\omega)))$ is a word over $\{l, r, i, \mathcal{W}\}$.*

Proof. By case analysis on the rule of $\texttt{Neg}(-)$, reasoning on whether the focus x is a subterm of the values being decomposed or not:

- If the variable x is not a subterm of the first value from the list l then the thread has the form:$(A; C, \Delta \vdash B, \uparrow) \cdot (A; C', \Delta \vdash B, \uparrow)$ where $A \in \Delta$ and the weight is $\mathcal{W}$.
- If the variable x is a subterm of the first value of the list l then by direct case analysis on the first value: if the value is of form $\langle v_1, v_2 \rangle$ then depending on whether x is in v_1 or v_2 the weight will be l or r, similarly for the $\texttt{inj}_l$ and $\texttt{inj}_r$, while the $\texttt{fold}$ will create weight i. □
- Finally in the case where the first value of the list is a variable y (whether $x = y$ or not), then as we apply the ex rule the weight is $\mathcal{W}$.

The same reasoning can now be done for the positive phase.

Definition 5.13 (Pre-thread of the positive phase)**.** Given a well-typo iso $\omega = \textbf{fix}\ f.\{v_1 \leftrightarrow e_1 \mid \ldots \mid v_n \leftrightarrow e_n\} : A \leftrightarrow B$ and a clause $(v_i \leftrightarrow e_i)$ where $\xi : \Theta \vdash_e e_i : B$, such that $f\ p$ is a subterm of e_i and x is the focus of the primitive recursive criterion of type $\mu X.C$. Considering $\pi = \texttt{Circ}(\omega)$, we define $PT_p(x, \texttt{Pos}(\xi, \Theta))$ as the pre-thread that starts from the conclusion of $\texttt{Pos}(\xi, \Theta)$ and follows the type $\mu X.C$ until the sequent $A \vdash^f B$ is reached.

We are guaranteed that such a pre-thread exists in an iso $\textbf{fix}\ f.\omega$ with clause $v \leftrightarrow e$ where $f\ p$ occurs in e since (i) the recursive the focus x is used linearly in e, (ii) x is contained in the variables

of p and therefore, the pre-thread will start by going up on the left-hand-side of the sequent until it bounces back on an axiom, and then go down and bounces back up on the `cut` generated by the subterm $f\ p$. This is formalized by the following lemma, where we look at the weight of the positive phase:

Lemma 5.14 (Weight of the pre-Thread for the positive phase)**.** *Given a well-typed iso* $\mathbf{fix}\ f.\omega$ *such that the variable* x *is the focus of the primitive recursive criterion in a clause* $v \leftrightarrow e$ *where* $\xi : \Theta \vdash_e e : A$ *and where* $f\ p$ *is a subterm of* e*, the weight of the pre-thread on the positive phase* $PT_p(x, \mathtt{Pos}(\xi, \Theta))$ *is of the shape* $\mathcal{W}^*\mathcal{A}\{\overline{l}, \overline{r}, \mathcal{W}\}^*\mathcal{C}$

Proof. By case analysis on $\mathtt{Pos}(e)$. As the thread starts on the left-hand-side (since the formula is in Θ), it only goes up by encountering cut-rules or right-rules, therefore we get $\mathcal{W}^*$, and the thread goes up all the way to an axiom rule. This axiom necessarily exists and is unique since the type system is linear corresponding to the variable x. This axiom rule therefore add the $\mathcal{A}$, and bounce back. Finally, the thread goes down on the purely positive proof, which by syntactical definition of an iso is a pattern p, and hence generating $\{\overline{l}, \overline{r}, \mathcal{W}\}^*$. The thread keeps going down as such until bouncing again on the `cut` rule generated by the subterm $f\ p$. □

We can then consider the infinite pre-thread as the concatenation of both the pre-thread of the negative phase, and the one of the positive phase.

Lemma 5.15 (Form of the Pre-Thread)**.** *Given the pre-thread* t *following the focus is the primitive recursive criterion* x *(meaning the composition of* $PT_n(-)$ *and* $PT_p(-)$*) we have that* $w(t) = p_0(\Sigma_{i\leq n} p_i \mathcal{W}_i^* \mathcal{A} q_i \mathcal{C})^\omega$ *with* p_0 *any prefix,* $p_i \in \{l, r, i, \mathcal{W}\}^*$ *and* $q_i \in \{\overline{l}, \overline{r}, \mathcal{W}\}^*$ *and with,* $\forall i \leq n, \overline{q_i} \sqsubset \overline{\overline{p_i}}$ *and* $\mid p_i \mid > \mid q_i \mid$ *without counting the* $\mathcal{W}$*, where* $p \sqsubset q$ *is* q *is a prefix of* p *and with* $\overline{a \cdot p} = \overline{a} \cdot \overline{p}$ *if* $a \in \{l, r, i\}$*,* $\overline{\overline{a}p} = a\overline{p}$ *if* $a \in \{l, r, i\}$ *and* $\overline{\mathcal{W}p} = \overline{p}$*.*

Proof. p_i is generated from Definition 5.11 while the rest up to the C (included) is generated from Definition 5.13.

First, we show that $\mid p_i \mid > \mid q_i \mid$ modulo the $\mathcal{W}$.

Since p_i is generated by the negative phase, we have that, modulo $\mathcal{W}$, $p_i = r^*l?\{l, r, i\}^*$, this is due to the definition of being primitive recursive and because we are following the focus of the primitive recursion criterion. By definition of being primitive recursive the input type of the iso is $A_1 \otimes (\cdots \otimes A_n)$, hence $r^*l?$ correspond to the search for the correct A_i of type $\mu X.B$ in the input type, while $\{l, r, i\}^*$ is the decomposition of the primitive recursive value, as described in Proposition 5.7.

As q_i corresponds to the purely positive proof, and by syntactical definition of the iso, we know that the purely positive proof is the encoding of a pattern $p = \langle x_1, \langle \ldots, x_n \rangle \rangle$. Hence, q_i can be decomposed as $\overline{l}?\ \overline{r}^*$. By the fact that the iso is primitive recursive we know that the variable in p is a strict subterm of the primitive recursive value, hence $\mid p_i \mid > \mid q_i \mid$.

The fact that $\overline{q_i} \sqsubset \overline{\overline{p_i}}$ is direct as the purely positive proof reconstructs the type $A_1 \otimes \cdots \otimes A_n$ without modifying the A_i while $\overline{\overline{p_i}}$ start by searching for the corresponding type A_i, so it is only composed of $\{l, r\}^*$, which will be the same as $\overline{q_i}$. □

Theorem 5.16 (The Pre-thread generated is a thread)**.** *Given a well-typed iso* $\mathbf{fix}\ f.\omega : A \leftrightarrow B$*, and its corresponding pre-proof* π*, the pre-thread following the focus of the recursive criteria of formula* $\mu X.C$ *is a thread, i.e. it can be uniquely decomposed into* $\bigodot(H_i \odot V_i)$ *with:*

- $w(V_i) \in \{l, r, i, \mathcal{W}\}^\infty$ *and non-empty if* $i \neq \lambda$
- $w(H_i) \in \mathfrak{P}$*, and it is non-empty if* $i \neq 0$

Proof. We set H_0 as $(\mu X.C; A \vdash B; \uparrow)$ (then $w(H_0) = \epsilon \in \mathfrak{P}$). V_0 consists of the whole negative phase followed by the positive phase until the axiom rule is reached. Therefore, V_0 starts by $(\mu X.C; A \vdash B; \uparrow)$ hence the first elements of H_0 and V_0 coincide. From Lemma 5.12, we know that $w(V_0)$ is a word over $\{l, r, i, \mathcal{W}\}$ followed by the $\mathcal{W}^*$ that comes from the positive phase, as described from Lemma 5.14. Then, for all $i \geq 1$, we set:

- H_i starts at $(\mu X.C; \mu X.C \vdash \mu X.C; \uparrow)$ just before the axiom rule so that the first element of $w(H_i)$ is $\mathcal{A}$. Then H_i is composed of:
  - All of the pre-thread going down on the Purely Positive Proof after the axiom, accumulating a word over $\{\overline{l}, \overline{r}, \overline{i}, \mathcal{W}\}^*$;
  - Going back up into the cut-rule of the bouncing cut, making a $\mathcal{C}$;
  - Going up to compensate every $\overline{x}$ seen in the Purely Positive Proof while going down. This is possible as shown in Lemma 5.15.
- V_i is the maximal possible sequence such that $w(V_i) \in \{l, r, i, \mathcal{W}\}^*$, i.e. the sequence that ends with $(A; A \vdash A; \uparrow)$. □

Theorem 5.17 (*Validity of proofs*)**.** *If* $\vdash_i \ \omega : A \leftrightarrow B$*, then* $\mathtt{Circ}(\omega)$ *satisfies* μ*MALL bouncing validity criterion.*

Proof. By Theorem 5.16 we know that we have a thread of which we also know by Theorem 5.16 that the visible part is not stationary.

Finally, by Lemma 5.15 and Theorem 5.16 we know that the visible part will see infinitely often the subformulas of the formula $\mu X.B$ that is the focus of the primitive recursive criterion. This is due to the difference in size in the part of the thread from the negative and from the positive phase and the fact that the positive phase does not encounter a μ formula when going down on a purely positive proof.

Therefore, the smallest formula we will encounter is a ν formula, validating the thread. □

Remark 5.18. In the last proof, we used a notion of validity different from [BDKS22]. While the one from [BDKS22] is more general, both criteria coincide on derivations that are images of isos.

5.2. **Proof Simulation.** We have shown that any iso from type $A \leftrightarrow B$ give rise to a valid proof of conclusion $A \vdash B$. In order to show the relationship between the cut-elimination procedure of μMALL and the rewriting system of our language we first introduce a slightly modified version of the iso rewriting system, which we call $\rightarrow_{e\beta}$, using explicit substitution in order to make the iso-evaluation closer to the cut-elimination procedure.

Remark 5.19. In the remaining of this section, we fix the following convention: given a substitution $\sigma = \{x_1 \mapsto v_1, \ldots, x_n \mapsto v_n\}$ (and assuming a linear order on variables) we use the shorthand $\mathtt{let}\,\sigma\,\mathtt{in}\,t$ for $\mathtt{let}\,x_1 = v_1\,\mathtt{in}\ldots\mathtt{let}\,x_n = v_n\,\mathtt{in}\,t$.

Definition 5.20 (Explicit Substitution Rewriting System)**.** $\rightarrow_{e\beta}$ is defined by the following rules :

$$\mathtt{let}\,x = v\,\mathtt{in}\,x \rightarrow_{\mathtt{elet}} v$$

$$\mathtt{let}\,\langle p_1, p_2\rangle = \langle t_1, t_2\rangle\,\mathtt{in}\,t \rightarrow_{\mathtt{elet}} \mathtt{let}\,p_1 = t_1\,\mathtt{in}\,\mathtt{let}\,p = t_2\,\mathtt{in}\,t$$

$$\mathtt{let}\,x = v\,\mathtt{in}\,\langle t_1, t_2\rangle \rightarrow_{\mathtt{elet}} \langle \mathtt{let}\,x = v\,\mathtt{in}\,t_1, t_2\rangle \qquad \text{when } x \in FV(t_1)$$

$$\mathtt{let}\,x = v\,\mathtt{in}\,\langle t_1, t_2\rangle \rightarrow_{\mathtt{elet}} \langle t_1, \mathtt{let}\,x = v\,\mathtt{in}\,t_2\rangle \qquad \text{when } x \in FV(t_2)$$

$$\mathtt{let}\,x = v\,\mathtt{in}\,\mathtt{inj}_l\,t \rightarrow_{\mathtt{elet}} \mathtt{inj}_l\,(\mathtt{let}\,x = v\,\mathtt{in}\,t)$$

$$\mathtt{let}\,x = v\,\mathtt{in}\,\mathtt{inj}_r\,t \rightarrow_{\mathtt{elet}} \mathtt{inj}_r\,(\mathtt{let}\,x = v\,\mathtt{in}\,t)$$

$$\texttt{let}\, x = v\, \texttt{in}\, \texttt{fold}\, t \to_{\texttt{elet}} \texttt{fold}\, (\texttt{let}\, x = v\, \texttt{in}\, t)$$
$$\texttt{let}\, x = v\, \texttt{in}\, \omega\, t \to_{\texttt{elet}} \omega\, (\texttt{let}\, x = v\, \texttt{in}\, t)$$

and the following rules (with C an arbitrary context):

$$\frac{t \to_{e\beta} \cup \to_{\texttt{elet}} t'}{C[t] \to_{e\beta} C[t']}\ \beta\text{−Cong}$$

$$\frac{}{(\textbf{fix}\ f.\omega)v \to_{e\beta} (\omega[f \leftarrow (\textbf{fix}\ f.\omega)])v}\ \beta\text{−IsoRec}$$

$$\frac{\sigma[v_i] = v'}{\{v_1 \leftrightarrow e_1 \mid \ldots \mid v_n \leftrightarrow e_n\}\, v' \to_{e\beta} \texttt{let}\, \sigma\, \texttt{in}\, e_i}\ \beta\text{−IsoApp}$$

As always, we use $\to^*_{e\beta}$ for the reflexive-transitive closure of $\to_{e\beta}$.

Proposition 5.21. *The following rule*

$$\frac{\sigma[p] = v}{\texttt{let}\, p = v\, \texttt{in}\, t \to_{e\beta} \texttt{let}\, \sigma\, \texttt{in}\, t}\ \beta - \text{LetE}$$

is admissible.

Proof. We have two cases depending on the shape of p. If it is a simple variable then it is direct, otherwise $p = \langle p_1, p_2\rangle$, then, by typing, t reduces to $\langle t_1, t_2\rangle$ in which case one can apply the second rule of $\to_{\texttt{elet}}$ and reason inductively on the two subterms until we obtain $\texttt{let}\, \sigma\, \texttt{in}\, t$. □

Each step of this rewriting system will correspond to exactly one step of cut-elimination, while in the previous system, the rewriting using a substitution σ corresponds to multiple steps of cut-elimination. $\to_{e\beta}$ makes this explicit, clarifying the relation between logic and reversible computation. Before establishing this result, we need to first ensure that both rewriting systems have the same operational semantics:

Lemma 5.22 (Specialization of the substitution on pairs)**.** *Let σ be a substitution that closes $\Delta \vdash_e \langle t_1, t_2\rangle$, then there exists σ_1, σ_2, such that $\sigma(\langle t_1, t_2\rangle) = \langle \sigma_1(t_1), \sigma_2(t_2)\rangle$ Where $\sigma = \sigma_1 \cup \sigma_2$.*

Proof. By the linearity of the typing system we know that $FV(t_1) \cup FV(t_2) = \emptyset$, so there always exists a decomposition of σ into σ_1, σ_2 defined as $\sigma_i = \{(x_i \mapsto v_i) \mid x_i \in FV(t_i)\}$ for $i \in \{1, 2\}$. □

Lemma 5.23 (Explicit substitution and substitution coincide)**.** *Let $\sigma = \{x_i \mapsto v_i\}$ be a substitution that closes t, then $\texttt{let}\, \sigma\, \texttt{in}\, t \to^*_{\texttt{elet}} \sigma(t)$.*

Proof. By induction on t.

- If $t = x$, then $\sigma(x) = v$ and $\texttt{let}\, x = v\, \texttt{in}\, x \to_{\texttt{elet}} v = \sigma(x)$.
- If $t = ()$ then σ is empty and no substitution applies.
- If $t = \langle t_1, t_2\rangle$, then by Lemma 5.22 $\sigma(\langle t_1, t_2\rangle) = \langle \sigma_1(t_1), \sigma_2(t_2)\rangle$. By $\to_{\texttt{elet}}$, each $\texttt{let}$ construction will enter either t_1 or t_2, we can then conclude by using the induction hypothesis.
- If $t = \texttt{let}\, p = t_1\, \texttt{in}\, t_2$ is similar to the product case.
- If t is $\texttt{inj}_l\, t', \texttt{inj}_r\, t', \texttt{fold}\, t'$ or $\omega\ t'$. All cases are treated in the same way: by definition of $\to_{\texttt{elet}}$, each $\texttt{let}$ will enter into the subterm t', as with the substitution σ, we then conclude applying the induction hypothesis. □

Corollary 5.24. *If $t \to t'$ then $t \to^*_{e\beta} t'$.*

Proof. By induction on $\to$, the case of IsoRec is the same, while for the other rules, just by applying either $\beta-$IsoApp or $\beta-$LetE and then by Lemma 5.23, $\texttt{let}\,\sigma\,\texttt{in}\,t \to^*_{\texttt{elet}} \sigma(t)$ for any substitution σ that closes t. But $\sigma(t) = t'$, so $t \to^*_{e\beta} t'$. □

It is then possible to show the first step of the simulation procedure, namely that $\to_{e\beta}$ corresponds to one step of cut-elimination:

Lemma 5.25 (Simulation of the let-rules of $\to_{e\beta}$)**.** *Let $\xi : \Theta \vdash_e t : A$ be a well-typed closed term. If $t \to_{\texttt{elet}} t'$, by Lemma 3.24 we have that there exists a typing derivation $\xi' : \Theta \vdash_e t' : A$ and then we get: $\texttt{Pos}(\xi, \Gamma) \rightsquigarrow^* \texttt{Pos}(\xi', \Gamma)$ for any Γ which is a list ordering the formulas occurring in Θ.*

Proof. By case analysis on $\to_{\texttt{elet}}$, for simplicity of reading, we will write directly $\texttt{Pos}(t, \Gamma)$ for $\texttt{Pos}(\xi, \Gamma)$ when ξ is a typing derivation of t and is clear from the context.

- $\texttt{let}\,x = v\,\texttt{in}\,x \to_{\texttt{elet}} v$.

$$\dfrac{\dfrac{\texttt{Pos}(v,\Gamma)}{\Gamma \vdash A} \qquad \dfrac{}{A \vdash A}\,\texttt{id}}{\Gamma \vdash A}\,\texttt{cut} \quad \rightsquigarrow \quad \dfrac{\texttt{Pos}(v,\Gamma)}{\Gamma \vdash A}$$

- $\texttt{let}\,\langle p_1, p_2\rangle = \langle t_1, t_2\rangle\,\texttt{in}\,t \to_{\texttt{elet}} \texttt{let}\,p_1 = t_1\,\texttt{in}\,\texttt{let}\,p_2 = t_2\,\texttt{in}\,t$. Denote by ξ the typing derivation of $\texttt{let}\,p_2 = t_2\,\texttt{in}\,t$ and by ξ' the typing derivation of t, then by definition of the negative phase we know that $\texttt{Neg}(\{([p_1], \xi\}, [A \otimes B], \Gamma_2 \cup \Gamma_3, B, \vdash)$ consists only in a succession of $\parr$ and ex rules. We note Δ_1 the context obtained by decomposing the type A through the negative phase, then:

$$\dfrac{\dfrac{\texttt{Pos}(t_1,\Gamma_1)}{\Gamma_1 \vdash A} \qquad \dfrac{\texttt{Pos}(t_2,\Gamma_2)}{\Gamma_2 \vdash B}\,\otimes}{\Gamma_1, \Gamma_2 \vdash A \otimes B} \qquad \dfrac{\dfrac{\texttt{Neg}(\{([p_1 :: p_2], \xi')\}, [A, B], \Gamma_3, C, \vdash)}{A, B, \Gamma_3 \vdash C}\,\parr}{A \otimes B, \Gamma_3 \vdash C}\,\texttt{cut}}{\Gamma_1, \Gamma_2, \Gamma_3 \vdash C} \quad \rightsquigarrow^*$$

$$\dfrac{\dfrac{\texttt{Pos}(t_1,\Gamma_1)}{\Gamma_1 \vdash A} \qquad \dfrac{\dfrac{\texttt{Pos}(t_2,\Gamma_2)}{\Gamma_2 \vdash B} \qquad \dfrac{\texttt{Neg}(\{([p_2], \xi')\}, B, \Gamma_3 \cup \Delta_1, C, \vdash)}{B, \Gamma_3, \Delta_1 \vdash C}\,\texttt{cut}}{\Gamma_2, \Gamma_3, \Delta_1 \vdash C}\,\parr \,/\, \texttt{ex}}{\Gamma_1, \Gamma_2, \Gamma_3 \vdash C}\,\texttt{cut}$$

In the first proof, we know that $\texttt{Neg}(\{([p_1 :: p_2], \xi')\}, [A, B], \Gamma_3, C, \vdash)$ will start by applying only $\parr$ rules and ex rules until A is fully decomposed into Δ_1, before doing the same with B. Because those are negative rules, we know that we can commute them, hence obtaining the same proofs.

- $\texttt{let}\,x = v\,\texttt{in}\,\texttt{inj}_l\,(t) \to_{\texttt{elet}} \texttt{inj}_l\,(\texttt{let}\,x = v\,\texttt{in}\,t)$.

$$\dfrac{\dfrac{\texttt{Pos}(v,\Gamma_1)}{\Gamma_1 \vdash C} \qquad \dfrac{\dfrac{\texttt{Pos}(t,\Gamma_2 :: C)}{\Gamma_2, C \vdash A}}{\Gamma_2, C \vdash A \oplus B}\,\oplus^1}{C, \Gamma_2 \vdash A \oplus B}\,\texttt{ex}}{\Gamma_1, \Gamma_2 \vdash A \oplus B}\,\texttt{cut} \quad \rightsquigarrow^* \quad \dfrac{\dfrac{\dfrac{\texttt{Pos}(t,\Gamma_1)}{\Gamma_1 \vdash C} \qquad \dfrac{\dfrac{\texttt{Pos}(t,\Gamma_2 :: C)}{\Gamma_2, C \vdash A}}{C, \Gamma_2 \vdash A}\,\texttt{ex}}{\Gamma_1, \Gamma_2 \vdash A}\,\texttt{cut}}{\Gamma_1, \Gamma_2 \vdash A \oplus B}\,\oplus^1$$

- The same goes for $\texttt{let}\, x \,=\, v \;\texttt{in}\; \texttt{inj}_r\; (t) \;\to_{\texttt{elet}}\; \texttt{inj}_r\; (\texttt{let}\, x = v \;\texttt{in}\; t)$ and for $\texttt{let}\, x \,=\, v \;\texttt{in}\; \texttt{fold}\; (t) \to_{\texttt{elet}} \texttt{fold}\; (\texttt{let}\, x = v \;\texttt{in}\; t)$

- $\texttt{let}\, x = v \;\texttt{in}\; \langle t_1, t_2\rangle \to_{\texttt{elet}} \langle \texttt{let}\, x = v \;\texttt{in}\; t_1, t_2\rangle$ when $x \in FV(t_1)$. Then:

$$\dfrac{\dfrac{\texttt{Pos}(v,\Gamma_1)}{\Gamma_1 \vdash A} \qquad \dfrac{\dfrac{\texttt{Pos}(t_1,\Gamma_2 :: A)}{\Gamma_2, A \vdash C} \qquad \dfrac{\texttt{Pos}(t_2,\Gamma_3)}{\Gamma_3 \vdash B}\otimes}{\dfrac{\Gamma_2,\Gamma_3, A \vdash C\otimes B}{A,\Gamma_2,\Gamma_3 \vdash C\otimes B}\,\texttt{ex}}}{\Gamma_1,\Gamma_2,\Gamma_3 \vdash C\otimes B}\,\texttt{cut} \quad \rightsquigarrow$$

$$\dfrac{\dfrac{\texttt{Pos}(v,\Gamma_1)}{\Gamma_1 \vdash A} \qquad \dfrac{\dfrac{\texttt{Pos}(t_1,\Gamma_2 :: A)}{\Gamma_2, A \vdash C}}{A,\Gamma_2 \vdash C}\,\texttt{ex}}{\Gamma_1,\Gamma_2 \vdash C}\,\texttt{cut} \qquad \dfrac{\texttt{Pos}(t_2,\Gamma_3)}{\Gamma_3 \vdash B}}{\Gamma_1,\Gamma_2,\Gamma_3 \vdash C\otimes B}\otimes$$

- Similar for the second rule on the pair.

- $\texttt{let}\, x = v \;\texttt{in}\; \omega\; t \to_{\texttt{elet}} \omega\; (\texttt{let}\, x = v \;\texttt{in}\; t)$. Then:

$$\dfrac{\dfrac{\texttt{Pos}(v,\Gamma_1)}{\Gamma_1 \vdash A} \qquad \dfrac{\dfrac{\texttt{Pos}(t,\Gamma_2 :: A)}{\Gamma_2, A \vdash B}\,\texttt{ex}}{A,\Gamma_2 \vdash B} \qquad \dfrac{\texttt{Circ}(\omega)}{B \vdash C}\,\texttt{cut}}{A,\Gamma_2 \vdash C}\,\texttt{cut} \quad \rightsquigarrow$$

$$\dfrac{\dfrac{\texttt{Pos}(v,\Gamma_1)}{\Gamma_1 \vdash A} \qquad \dfrac{\dfrac{\texttt{Pos}(t,\Gamma_2 :: A)}{\Gamma_2, A \vdash B}\,\texttt{ex}}{A,\Gamma_2 \vdash B}}{\Gamma_1,\Gamma_2 \vdash B}\,\texttt{cut} \qquad \dfrac{\texttt{Circ}(\omega)}{B \vdash C}}{\Gamma_1,\Gamma_2 \vdash C}\,\texttt{cut}$$

□

We can now show how the pattern-matching is captured by the cut-elimination:

Lemma 5.26. *Given a set of well-typed values and expressions $\Theta_i \vdash_e v_i : A$ and $\xi_i : \Theta_i \vdash_e e_i : B$ for $i \in \{1,\dots,n\}$ such that $\texttt{OD}_A(v_1,\dots,v_n)$ and $\texttt{OD}_B(\mathit{Val}(e_1),\dots,\mathit{Val}(e_n))$, then, given a well-typed value $\Sigma \vdash_e v : A$ such that there exists $j \in \{1,\dots,n\}$ and σ such that $\sigma[v_j] = v$ then, given $\Gamma = \overline{\Sigma}$, we have:*

$$\pi = \dfrac{\dfrac{\texttt{Pos}(v,\Gamma)}{\Gamma \vdash A} \qquad \dfrac{\texttt{Neg}(\{([v_i],\xi_i)_{i\in\{1,\dots,n\}}\},[A],\emptyset,B,\vdash)}{A \vdash B}}{\Gamma \vdash B}\,\texttt{cut} \rightsquigarrow^* \dfrac{\texttt{Pos}(\texttt{let}\,\sigma\;\texttt{in}\; e_j,\Gamma)}{\Gamma \vdash B} = \pi'$$

Proof. By induction on the derivation of $\texttt{OD}_A(\{v_1,\dots,v_n\})$:

- Case $\texttt{OD}_A(\{x\})$. We get $\sigma[x] = v$ then $\pi = \pi'$.

- Case $\texttt{OD}_{\mathbb{1}}(\{()\})$. In this case, $\sigma[()] = ()$ and there is a single e with typing derivation ξ.

$$\pi = \dfrac{\dfrac{}{\vdash \mathbb{1}}\,\mathbb{1} \qquad \dfrac{\dfrac{\mathtt{Pos}(\xi, \emptyset)}{\vdash B}}{\mathbb{1} \vdash B}\,\bot}{\vdash G}\,\mathtt{cut} \quad \text{which reduces to} \quad \dfrac{\mathtt{Pos}(e, \emptyset)}{\vdash B} = \pi' \text{ as } \sigma \text{ is empty.}$$

- Case $\mathtt{OD}_{\mu X.A}(\{(\mathtt{fold}\ v_i)_{i\in I}\})$ such that $\sigma[\mathtt{fold}\ v_j] = \mathtt{fold}\ v'$. Then

$$\pi = \dfrac{\dfrac{\dfrac{\mathtt{Pos}(v', \Gamma)}{\Gamma \vdash A[X \leftarrow \mu X.A]}}{\Gamma \vdash \mu X.A}\,\mu \qquad \dfrac{\mathtt{Neg}(\{([v_i], \xi_i)_{i\in I}\}, [A[X \leftarrow \mu X.A]], \emptyset, B, \vdash)}{\dfrac{H[X \leftarrow \mu X.H] \vdash B}{\mu X.H \vdash B}\,\nu}}{\Gamma \vdash B}\,\mathtt{cut}$$

  will reduce to

$$\dfrac{\dfrac{\mathtt{Pos}(v', \Gamma)}{\Gamma \vdash A[X \leftarrow \mu X.A]} \qquad \dfrac{\mathtt{Neg}(\{([v_i], \xi_i)_{i\in I}\}, [A[X \leftarrow \mu X.A]], \emptyset, B, \vdash)}{A[X \leftarrow \mu X.A] \vdash B}\,\mathtt{cut}}{\Gamma \vdash B}$$

  then we can apply our induction hypothesis.

- Case $\mathtt{OD}_{A\oplus B}(\{(\mathtt{inj}_l\ v_i)_{i\in I}\} \cup \{(\mathtt{inj}_r\ v_k)_{k\in K}\})$ such that $\sigma[\mathtt{inj}_l\ v_j] = \mathtt{inj}_l\ v'$. Then the proof

$$\pi = \dfrac{\dfrac{\dfrac{\mathtt{Pos}(v', \Gamma)}{\Gamma \vdash A}}{\Gamma \vdash A \oplus B}\,\oplus^1 \qquad \dfrac{\dfrac{\mathtt{Neg}(\{([v_i], \xi_i)_{i\in I}\}, [A], \emptyset, C, \vdash)}{H \vdash C} \qquad \dfrac{\mathtt{Neg}(\{([v_k], \xi_k)_{k\in K}\}, [B], \emptyset, C, \vdash)}{B \vdash C}\,\&}{A \oplus B \vdash C}\,\mathtt{cut}}{\Gamma \vdash C}$$

  reduces to

$$\dfrac{\dfrac{\mathtt{Pos}(v', \Gamma)}{\Gamma \vdash A} \qquad \dfrac{\mathtt{Neg}(\{([v_i], \xi_i)_{i\in I}\}, [A], \emptyset, C, \vdash)}{A \vdash C}\,\mathtt{cut}}{\Gamma \vdash C}$$

  then we can apply our induction hypothesis.

- The case $\mathtt{OD}_{A\oplus B}(\{\mathtt{inj}_l\ v_i\} \cup \{\mathtt{inj}_r\ v_k\})$ such that $\sigma[\mathtt{inj}_l\ v_j] = \mathtt{inj}_l\ v'$ is similar to the previous case.

- Case $\mathtt{OD}_{A\otimes B}(\{(\langle v_i^1, v_i^2\rangle)_{i\in I}\})$ with $\sigma[\langle v_j^1, v_j^2\rangle] = \langle v_1', v_2'\rangle$ and therefore $\sigma = \sigma_1 \cup \sigma_2$. Then

$$\pi = \dfrac{\dfrac{\dfrac{\mathtt{Pos}(v_1', \Gamma_1)}{\Gamma_1 \vdash A} \qquad \dfrac{\mathtt{Pos}(v_2', \Gamma_2)}{\Gamma_2 \vdash B}}{\Gamma_1, \Gamma_2 \vdash A \otimes B}\,\otimes \qquad \dfrac{\mathtt{Neg}(\{([\langle v_i^1, v_i^2\rangle], \xi_i)_{i\in I}\}, [A, B], \emptyset, C, \vdash)}{\dfrac{A, B \vdash C}{A \otimes B \vdash C}\,\parr}}{\Gamma_1, \Gamma_2 \vdash C}\,\mathtt{cut}$$

  reduces to

$$\dfrac{\dfrac{\mathtt{Pos}(v_1', \Gamma_1)}{\Gamma_1 \vdash A} \qquad \dfrac{\dfrac{\mathtt{Pos}(v_2', \Gamma_2)}{\Gamma_2 \vdash B} \qquad \mathtt{Neg}(\{((v_i^1 :: v_i^2)_i, \xi_i)_{i\in I}\}, [A, B], \emptyset, C, \vdash)}{\Gamma_2, A \vdash C}\,\mathtt{cut}}{\Gamma_1, \Gamma_2 \vdash C}\,\mathtt{cut}$$

Because the negative phase on $[v_1, v_2]$ only produces $\&, \parr, \top, \nu$ rules, we get that $\texttt{Neg}(\{(v_i^1 :: v_i^2, \xi_i)_{i\in I}\}, [A, B], \emptyset, C, \vdash) = \texttt{Neg}(\{(v_i^2 :: v_i^1, \xi_i)_{i\in I}\}, [B, A], \emptyset, C, \vdash)$ up to the commutation of rules of Linear Logic. Therefore, we can get

$$\dfrac{\dfrac{\texttt{Pos}(v_1', \Gamma_1)}{\Gamma_1 \vdash A} \quad \dfrac{\dfrac{\texttt{Pos}(v_2', \Gamma_2)}{\Gamma_2 \vdash B} \quad \dfrac{\texttt{Neg}(\{(v_i^2 :: v_i^1, \xi_i)_{i\in I}\}, [B, A], \emptyset, C, \vdash)}{A, B \vdash C}\ \texttt{ex}}{\Gamma_2, A \vdash C}\ \texttt{cut}}{\Gamma_1, \Gamma_2 \vdash C}\ \texttt{cut}$$

which, by induction hypothesis on v_2, reduces to

$$\dfrac{\dfrac{\texttt{Pos}(v_1', \Gamma_1)}{\Gamma_1 \vdash A} \quad \dfrac{\texttt{Neg}(\{(v_i^1, \texttt{let}\ \sigma_2\ \texttt{in}\ \xi_i)_{i\in I}\}, [A], \emptyset, C, \vdash)}{\Gamma_2, A \vdash C}}{\Gamma_1, \Gamma_2 \vdash C}\ \texttt{cut}$$

and then we can apply our second induction hypothesis on v_1. □

We can then conclude the proof with the global simulation theorem as a direct implication of the two previous lemmas:

Theorem 5.27 (Iso-substitution cut-elimination)**.** *Let* $\vdash \{v_1 \leftrightarrow e_1 \mid \ldots \mid v_n \leftrightarrow e_n\}\ v \to \sigma(e_i)$ *when* $\sigma[v_i] = v$ *then* $\texttt{Pos}(\{v_1 \leftrightarrow e_1 \mid \ldots \mid v_n \leftrightarrow e_n\}\ v, []) \rightsquigarrow^* \texttt{Pos}(\texttt{let}\ \sigma\ \texttt{in}\ e_i, []) \rightsquigarrow^* \texttt{Pos}(\sigma(e_i), [])$.

Proof. By Lemma 5.23, we know that the explicit substitution and the substitution coincide. Lemma 5.25 tells us that one step of $\to_{\texttt{elet}}$ is simulated by exactly one step of cut-elimination and finally Lemma 5.26 tells us that we simulate properly the pattern-matching. □

Corollary 5.28 (Simulation)**.** *Given an iso* $\vdash_i\ \omega : A \leftrightarrow B$ *and values* $\vdash_e v : A$ *and* $\vdash_e v' : B$ *such that* $\omega\ v \to^* v'$*, we have that* $\texttt{Pos}(\omega\ v, []) \rightsquigarrow^* \texttt{Pos}(v', [])$

Proof. Direct application of Theorem 5.27. □

We can wonder about the converse: taking a term $\omega\ v$ and its corresponding proof $\pi = \texttt{Pos}(\omega\ v, [])$, if $\pi \rightsquigarrow^* \pi'$ and $\omega\ v \to^* v'$, does $\texttt{Pos}(v', []) = \pi'$? The answer to that question is yes. Since $\texttt{Pos}(v', [])$ is valid and is the translation of an iso that terminates, we know that it reaches a purely positive proof of $\vdash B$. This cut-free proof is necessarily unique. By the simulation, we know that $\texttt{Pos}(v', [])$ reduces to another purely positive proof of $\vdash B$, but by uniqueness those two proofs must be the same.

This leads to the following corollary:

Corollary 5.29 (Isomorphism of proofs.)**.** *Given a well-typed iso* $\vdash_i\ \omega : F \leftrightarrow G$ *and two well-typed closed values* v_1 *of type* F *and* v_2 *of type* G *and the proofs* $\pi : F \vdash G$, $\pi^\perp : G \vdash F$, $\phi : \vdash F$, $\psi : \vdash G$ *corresponding respectively to the translation of* $\omega, \omega^\perp, v_1, v_2$ *then:*

$$\dfrac{\phi}{\vdash F} \leftrightsquigarrow \dfrac{\dfrac{\dfrac{\phi}{\vdash F} \quad \dfrac{\pi}{F \vdash G}\ \texttt{cut}}{\vdash G} \quad \dfrac{\pi^\perp}{G \vdash F}\ \texttt{cut}}{\vdash F} \qquad \dfrac{\dfrac{\dfrac{\psi}{\vdash G} \quad \dfrac{\pi^\perp}{G \vdash F}\ \texttt{cut}}{\vdash F} \quad \dfrac{\pi}{F \vdash G}\ \texttt{cut}}{\vdash G} \rightsquigarrow \dfrac{\psi}{\vdash G}$$

Proof. As a direct implication of Theorem 3.31 and Corollary 5.28. □

Define the proof π_0 and π_S of the natural number 0 and the Successor as:

$$\pi_0 = \dfrac{\dfrac{\dfrac{}{\vdash \mathbb{1}}\,\mathbb{1}}{\vdash \mathbb{1} \oplus \mathbb{N}}\,\oplus^1}{\vdash \mathbb{N}}\,\mu \quad \text{and} \quad \pi_S = \dfrac{\dfrac{\dfrac{}{\mathbb{N} \vdash \mathbb{N}}\,\texttt{id}}{\mathbb{N} \vdash \mathbb{1} \oplus \mathbb{N}}\,\oplus^2}{\mathbb{N} \vdash \mathbb{N}}\,\mu \ .$$

Then the proof of ω_1, ω_2 and CantorPairing are:

$$\pi_{\omega_1} = \left\{ \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash \mathbb{1}}\,\mathbb{1}}{\vdash (\mathbb{N}\otimes\mathbb{N}) \oplus \mathbb{1}}\,\oplus^2}{\mathbb{1} \vdash (\mathbb{N}\otimes\mathbb{N}) \oplus \mathbb{1}}\,\bot \quad \dfrac{\dfrac{\dfrac{\pi_0 \quad \pi_0}{\vdash \mathbb{N}\otimes\mathbb{N}}\,\otimes}{\vdash (\mathbb{N}\otimes\mathbb{N}) \oplus \mathbb{1}}\,\oplus^1}{\mathbb{1} \vdash (\mathbb{N}\otimes\mathbb{N}) \oplus \mathbb{1}}\,\bot}{\mathbb{1} \oplus \mathbb{N} \vdash (\mathbb{N}\otimes\mathbb{N}) \oplus \mathbb{1}}\,\nu \quad \dfrac{\dfrac{\dfrac{\pi_S \quad \pi_0}{\mathbb{N} \vdash \mathbb{N}\otimes\mathbb{N}}\,\otimes}{\mathbb{N} \vdash (\mathbb{N}\otimes\mathbb{N}) \oplus \mathbb{1}}\,\oplus^1}{\mathbb{N} \vdash (\mathbb{N}\otimes\mathbb{N}) \oplus \mathbb{1}}\,\nu}{\mathbb{1} \oplus \mathbb{N} \vdash (\mathbb{N}\otimes\mathbb{N}) \oplus \mathbb{1}}\,\nu \quad \dfrac{\dfrac{\dfrac{}{\mathbb{N} \vdash \mathbb{N}}\,\texttt{id} \quad \dfrac{\pi_S \quad \dfrac{}{\mathbb{N}, \mathbb{N} \vdash \mathbb{N}\otimes\mathbb{N}}\,\otimes}{\mathbb{N}, \mathbb{N} \vdash (\mathbb{N}\otimes\mathbb{N}) \oplus \mathbb{1}}\,\oplus^1}{\mathbb{1}, \mathbb{N} \vdash (\mathbb{N}\otimes\mathbb{N}) \oplus \mathbb{1}}\,\bot}{\mathbb{1} \oplus \mathbb{N}, \mathbb{N} \vdash (\mathbb{N}\otimes\mathbb{N}) \oplus \mathbb{1}}\,\nu}{\mathbb{N}, \mathbb{N} \vdash (\mathbb{N}\otimes\mathbb{N}) \oplus \mathbb{1}}\,\nu}{\mathbb{N}\otimes\mathbb{N} \vdash (\mathbb{N}\otimes\mathbb{N}) \oplus \mathbb{1}}\,\parr \right.$$

$$\pi_{\omega_2} = \left\{ \dfrac{\dfrac{\dfrac{}{\mathbb{N}\otimes\mathbb{N} \vdash \mathbb{N}\otimes\mathbb{N}}\,\texttt{id} \quad \dfrac{}{\mathbb{N}\otimes\mathbb{N} \vdash \mathbb{N}}\,be(g)}{\mathbb{N}\otimes\mathbb{N} \vdash \mathbb{N}}\,\texttt{cut} \quad \pi_S}{\mathbb{N}\otimes\mathbb{N} \vdash \mathbb{N}}\,\texttt{cut} \quad \dfrac{\pi_0}{\mathbb{1} \vdash \mathbb{N}}\,\bot}{(\mathbb{N}\otimes\mathbb{N}) \oplus \mathbb{1} \vdash \mathbb{N}}\,\& \right.$$

$$\pi = \left\{ \dfrac{\dfrac{\dfrac{}{\mathbb{N}\otimes\mathbb{N} \vdash \mathbb{N}\otimes\mathbb{N}}\,\texttt{id} \quad \dfrac{\pi_{\omega_1}}{\mathbb{N}\otimes\mathbb{N} \vdash (\mathbb{N}\otimes\mathbb{N}) \oplus \mathbb{1}}}{\mathbb{N}\otimes\mathbb{N} \vdash (\mathbb{N}\otimes\mathbb{N}) \oplus \mathbb{1}}\,\texttt{cut} \quad \dfrac{\dfrac{}{(\mathbb{N}\otimes\mathbb{N}) \oplus \mathbb{1} \vdash (\mathbb{N}\otimes\mathbb{N}) \oplus \mathbb{1}}\,\texttt{id} \quad \dfrac{\pi_{\omega_2}}{(\mathbb{N}\otimes\mathbb{N}) \oplus \mathbb{1} \vdash \mathbb{N}}}{(\mathbb{N}\otimes\mathbb{N}) \oplus \mathbb{1} \vdash \mathbb{N}}\,\texttt{cut} \quad \dfrac{}{\mathbb{N} \vdash \mathbb{N}}\,\texttt{id}}{(\mathbb{N}\otimes\mathbb{N}) \oplus \mathbb{1} \vdash \mathbb{N}}\,\texttt{cut}}{\mathbb{N}\otimes\mathbb{N} \vdash^{g} \mathbb{N}}\,\texttt{cut} \right.$$

Figure 6: Cantor Pairing in μMALL

We should notice that this is a specific notion of proof-isomorphisms. Usually, one would expect the proofs corresponding to ω and $\omega^\perp$ reduce to the identity proof. However, it is not clear how to prove this formally, and is left for future work.

6. Conclusion

Summary of the contribution. We presented a linear, reversible language with inductive types. We showed how ensuring non-overlapping and exhaustivity is enough to ensure the reversibility of the isos. The language comes with both an expressivity result that shows that any Primitive Recursive Functions can be encoded in this language as well as an interpretation of programs into μMALL proofs. The latter result rests on the fact that our isos are *structurally recursive*.

Future works. We showed a one-way encoding from isos to proofs of μMALL, it is clear that there exist proof-isomorphisms of μMALL that do not correspond to an iso of our language. Therefore, a first extension to our work would be to relax the different constraints to allow for more functions to be encoded. For instance, the Cantor Pairing [CLV24] $\mathbb{N} \leftrightarrow \mathbb{N}\otimes\mathbb{N}$ can be encoded, by representing $\mathbb{N}$ with `npos`, and then $0 = \texttt{fold}\ (\texttt{inj}_l\ ())$ and $S\ x = \texttt{fold}\ (\texttt{inj}_r\ (x))$, as follows.

Example 6.1 (Cantor Pairing)**.**

$$\omega_1 = \left\{ \begin{array}{ll} \langle S\ i, j\rangle & \leftrightarrow \mathtt{inj}_l\ (\langle i, S\ j\rangle) \\ \langle 0, S\ S\ j\rangle & \leftrightarrow \mathtt{inj}_l\ (\langle S\ j, 0\rangle) \\ \langle 0, S\ 0\rangle & \leftrightarrow \mathtt{inj}_l\ (\langle 0, 0\rangle) \\ \langle 0, 0\rangle & \leftrightarrow \mathtt{inj}_r\ () \end{array} \right\} : \mathbb{N} \otimes \mathbb{N} \leftrightarrow (\mathbb{N} \otimes \mathbb{N}) \oplus \mathbb{1}$$

$$\omega_2 = \left\{ \begin{array}{ll} \mathtt{inj}_l\ (x) & \leftrightarrow \mathtt{let}\ y = g\ x\ \mathtt{in}\ S\ y \\ \mathtt{inj}_r\ (()) & \leftrightarrow 0 \end{array} \right\} : (\mathbb{N} \otimes \mathbb{N}) \oplus \mathbb{1} \leftrightarrow \mathbb{N}$$

$$\mathrm{CantorPairing} = \mathbf{fix}\ g. \left\{ \begin{array}{r} x \leftrightarrow \mathtt{let}\ y = \omega_1\ x\ \mathtt{in} \\ \mathtt{let}\ z = \omega_2\ y\ \mathtt{in}\ z \end{array} \right\} : (\mathbb{N} \otimes \mathbb{N}) \leftrightarrow \mathbb{N}$$

The iso ω_1 is computing the predecessor of a point in the plane, marching in diagonal, as the Cantor isomorphism prescribes. The iso ω_2 increases a counter if the origin has not been met. The iso $\mathrm{CantorPairing}$ is then simply repeatedly iterating the predecessor function of the plane $\mathbb{N} \otimes \mathbb{N}$, increasing the counter as it goes. It stops when the origin is met: the value of the counter is the desired number.

Then, consider its associated proofs $\pi, \pi_{\omega_1}, \pi_{\omega_2}$, given in Figure 6, of (respectively) the isos $\mathrm{CantorPairing}$, ω_1 and ω_2. While the iso has the expected operational semantics, it is not well-typed as it is not structurally recursive. Its associated derivation π is also not valid: the visible part is not contained in the infinite branches. However, whenever π is cut with any proof of $\vdash \mathbb{N} \otimes \mathbb{N}$ it will reduce to a finite proof a $\vdash \mathbb{N}$, computing the desired result. Therefore we need to extend the validity criterion of μMALL. In order to properly type this iso, one would require accepting lexicographical order (or more generally, any well-founded order) on recursive isos that act as a termination proof. And then, see how such a criterion would be captured in terms of pre-proof validity. Along with this, allowing for coinductive statements and non-polarity-preserving types and terms would allow to express more functions and capture a bigger subset of μMALL isomorphisms. This would require relaxing the condition on recursive isos, as termination would be no longer possible to ensure. This is a focus of our forthcoming research.

A second direction for future work is to consider quantum computation, by extending our language with linear combinations of terms. We plan to study purely quantum recursive types and generalized quantum loops: in [SVV18], lists are the only recursive type that is captured and recursion is terminating. The logic μMALL would help in providing a finer understanding of termination and non-termination.

A quantum-oriented version of the language presented in this paper has already been studied from a denotational semantics point of view [Cha23], albeit without inductive types and the proof-theoretical aspects have not been studied yet.

Acknowledgments

This work has been partially funded by the French National Research Agency (ANR) with the projects RECIPROG ANR-21-CE48-0019, PPS ANR-19-CE48-0014, TaQC ANR-22-CE47-0012 and within the framework of "Plan France 2030", under the research projects EPIQ ANR-22-PETQ-0007, OQULUS ANR-23-PETQ-0013, HQI-Acquisition ANR-22-PNCQ-0001 and HQI-R&D ANR-22-PNCQ-0002.

References

[AG11] Holger Bock Axelsen and Robert Glück. A simple and efficient universal reversible turing machine. In *International Conference on Language and Automata Theory and Applications*, pages 117–128. Springer, 2011.

[Bae08] David Baelde. *A Linear Approach to the Proof-Theory of Least and Greatest Fixed Points*. Theses, École Polytechnique, 2008.

[Bae12] David Baelde. Least and greatest fixed points in linear logic. *ACM Transactions on Computational Logic*, 13(1), January 2012. URL: http://www.lsv.fr/Publis/PAPERS/PDF/baelde12tocl.pdf, doi:10.1145/2071368.2071370.

[BAP+12] Antoine Bérut, Artak Arakelyan, Artyom Petrosyan, Sergio Ciliberto, Raoul Dillenschneider, and Eric Lutz. Experimental verification of landauer's principle linking information and thermodynamics. *Nature*, 483(7388):187–189, 2012.

[Bar84] Henk Barendregt. The lambda calculus: its syntax and semantics. *Studies in logic and the foundations of Mathematics*, 1984.

[BC13] Yves Bertot and Pierre Castéran. *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions*. Springer Science & Business Media, 2013.

[BDC99] Vincent Balat and Roberto Di Cosmo. A linear logical view of linear type isomorphisms. In *Computer Science Logic: 13th International Workshop, CSL'99 8th Annual Conference of the EACSL Madrid, Spain, September 20–25, 1999 Proceedings 13*, pages 250–265. Springer, 1999.

[BDCL92] Kim B. Bruce, Roberto Di Cosmo, and Giuseppe Longo. Provable isomorphisms of types. *Mathematical Structures in Computer Science*, 2(2):231–247, 1992.

[BDKS22] David Baelde, Amina Doumane, Denis Kuperberg, and Alexis Saurin. *Bouncing Threads for Circular and Non-Wellfounded Proofs: Towards Compositionality with Circular Proofs*, pages 1–13. Association for Computing Machinery, New York, NY, USA, 2022. URL: https://doi.org/10.1145/3531130.3533375.

[BDS16] David Baelde, Amina Doumane, and Alexis Saurin. Infinitary Proof Theory: the Multiplicative Additive Case. In Jean-Marc Talbot and Laurent Regnier, editors, *25th EACSL Annual Conference on Computer Science Logic (CSL 2016)*, volume 62 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 42:1–42:17, Dagstuhl, Germany, 2016. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. URL: http://drops.dagstuhl.de/opus/volltexte/2016/6582, doi:10.4230/LIPIcs.CSL.2016.42.

[Ben73] Charles H Bennett. Logical reversibility of computation. *IBM journal of Research and Development*, 17(6):525–532, 1973.

[BL85] Kim B Bruce and Giuseppe Longo. Provable isomorphisms and domain equations in models of typed languages. In *Proceedings of the seventeenth annual ACM symposium on Theory of computing*, pages 263–272, 1985.

[BM07] David Baelde and Dale Miller. Least and greatest fixed points in linear logic. In Nachum Dershowitz and Andrei Voronkov, editors, *Proceedings of the 14th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR'07)*, volume 4790 of *Lecture Notes in Artificial Intelligence*, pages 92–106, Yerevan, Armenia, October 2007. Springer. URL: http://www.lsv.fr/Publis/PAPERS/PDF/baelde07lpar.pdf, doi:10.1007/978-3-540-75560-9_9.

[Cha23] Kostia Chardonnet. *Towards a Curry-Howard Correspondence for Quantum Computation*. Theses, Université Paris-Saclay, January 2023. URL: https://theses.hal.science/tel-03959403.

[CLV21] Kostia Chardonnet, Louis Lemonnier, and Benoît Valiron. Categorical semantics of reversible pattern-matching. In Ana Sokolova, editor, Proceedings 37th Conference on *Mathematical Foundations of Programming Semantics,* Hybrid: Salzburg, Austria and Online, 30th August - 2nd September, 2021, volume 351 of *Electronic Proceedings in Theoretical Computer Science*, pages 18–33. Open Publishing Association, 2021. doi:10.4204/EPTCS.351.2.

[CLV24] Kostia Chardonnet, Louis Lemonnier, and Benoît Valiron. Semantics for a turing-complete reversible programming language with inductive types. In *Proceedings of FSCD*, 2024. To appear.

[CSV23] Kostia Chardonnet, Alexis Saurin, and Benoît Valiron. A Curry-Howard correspondence for linear, reversible computation. In Bartek Klin and Elaine Pimentel, editors, *31st EACSL Annual Conference on Computer Science Logic (CSL 2023)*, volume 252 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 13:1–13:18, Dagstuhl, Germany, 2023. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. URL: https://drops.dagstuhl.de/opus/volltexte/2023/17474, doi:10.4230/LIPIcs.CSL.2023.13.

[Cur34] Haskell B Curry. Functionality in combinatory logic. *Proceedings of the National Academy of Sciences*, 20(11):584–590, 1934.

[DGL23] Rémi Di Guardia and Olivier Laurent. Type isomorphisms for multiplicative-additive linear logic. In *8th International Conference on Formal Structures for Computation and Deduction (FSCD 2023)*, volume 260, 2023.

[DL08] Joachim De Lataillade. Second-order type isomorphisms through game semantics. *Annals of Pure and Applied Logic*, 151(2-3):115–150, 2008.

[Dou17] Amina Doumane. *On the infinitary proof theory of logics with fixed points*. Theses, Université Paris 7 - Diderot, June 2017. URL: https://hal.science/tel-01676953.

[DR79] Willem-Paul De Roever. Recursive program schemes: semantics and proof theory. *Journal of Symbolic Logic*, 44(4), 1979.

[FS13] Jérôme Fortier and Luigi Santocanale. Cuts for circular proofs: semantics and cut-elimination. In Simona Ronchi Della Rocca, editor, *Computer Science Logic 2013 (CSL 2013)*, volume 23 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 248–262, Dagstuhl, Germany, 2013. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. URL: http://drops.dagstuhl.de/opus/volltexte/2013/4201.

[GHH+13] Marco Gaboardi, Andreas Haeberlen, Justin Hsu, Arjun Narayan, and Benjamin C Pierce. Linear dependent types for differential privacy. In *Proceedings of the 40th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 357–370, 2013.

[Gir87] Jean-Yves Girard. Linear logic. *Theoretical computer science*, 50(1):1–101, 1987.

[GY23] Robert Glück and Tetsuo Yokoyama. Reversible computing from a programming language perspective. *Theoretical Computer Science*, 953:113429, 2023. URL: https://www.sciencedirect.com/science/article/pii/S0304397522003619, doi:https://doi.org/10.1016/j.tcs.2022.06.010.

[How80] William A Howard. The formulae-as-types notion of construction. *To HB Curry: essays on combinatory logic, lambda calculus and formalism*, 44:479–490, 1980.

[JJKD17] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. Rustbelt: Securing the foundations of the rust programming language. *Proceedings of the ACM on Programming Languages*, 2(POPL):1–34, 2017.

[JKT18] Petur Andrias Højgaard Jacobsen, Robin Kaarsgaard, and Michael Kirkedal Thomsen. CoreFun : A typed functional reversible core language. In Jarkko Kari and Irek Ulidowski, editors, *Reversible Computation - 10th International Conference, RC 2018, Leicester, UK, September 12-14, 2018, Proceedings*, volume 11106 of *Lecture Notes in Computer Science*, pages 304–321. Springer, 2018. doi:10.1007/978-3-319-99498-7_21.

[JSS14] Roshan P James, Amr Sabry, and J Street. Theseus: a high level language for reversible computing. *Work-in-progress report presented at Reversible Computation 2014*, 2014. Draft, available at https://legacy.cs.indiana.edu/~sabry/papers/theseus.pdf.

[Koz83] Dexter Kozen. Results on the propositional μ-calculus. *Theoretical computer science*, 27(3):333–354, 1983.

[KP23] Bartek Klin and Elaine Pimentel. Lipics, volume 252, csl 2023, complete volume. In *31st EACSL Annual Conference on Computer Science Logic (CSL 2023)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2023.

[KR21] Robin Kaarsgaard and Mathys Rennela. Join inverse rig categories for reversible functional programming, and beyond. In Ana Sokolova, editor, *Proceedings 37th Conference on Mathematical Foundations of Programming Semantics, MFPS 2021, Hybrid: Salzburg, Austria and Online, 30th August - 2nd September, 2021*, volume 351 of *EPTCS*, pages 152–167, 2021. doi:10.4204/EPTCS.351.10.

[Lan61] Rolf Landauer. Irreversibility and heat generation in the computing process. *IBM Journal of Research and Development.*, 5(3):183–191, 1961. doi:10.1147/rd.53.0183.

[Ler09] Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.

[MHRVM19] Kenji Maillard, Cătălin Hriţcu, Exequiel Rivas, and Antoine Van Muylder. The next 700 relational program logics. *Proceedings of the ACM on Programming Languages*, 4(POPL):1–33, 2019.

[MY07] Kenichi Morita and Yoshikazu Yamaguchi. A universal reversible turing machine. In *International Conference on Machines, Computations, and Universality*, pages 90–98. Springer, 2007.

[Par69] David Park. Fixpoint induction and proofs of program properties. *Machine intelligence*, 5, 1969.

[PPR20] Luca Paolini, Mauro Piccolo, and Luca Roversi. A class of recursive permutations which is primitive recursive complete. *Theoretical Computer Science*, 813:218–233, 2020. URL: https://www.sciencedirect.com/science/article/pii/S0304397519307558.

[Rey02] John C Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*, pages 55–74. IEEE, 2002.

[Rit93] Mikael Rittri. Retrieving library functions by unifying types modulo linear isomorphism. *RAIRO-Theoretical Informatics and Applications*, 27(6):523–540, 1993.

[RJ87] Hartley Rogers Jr. *Theory of recursive functions and effective computability*. MIT press, 1987.

[SA97] Sergei V Soloviev and Alexander E Andreev. A decision algorithm for linear isomorphism of types with complexity o (nlog2 (n)). In *Category Theory and Computer Science*, volume 1290, 1997.

[San02] Luigi Santocanale. A calculus of circular proofs and its categorical semantics. In *FOSSACS 2002*, volume 2303 of *FOSSACS 2002, proceedings of the 5th International Conference on Foundations of Software Science and Computation Structures.*, pages 357–371, Grenoble, France, April 2002. Springer. URL: https://hal.archives-ouvertes.fr/hal-01261170.

[Sau23] Alexis Saurin. A linear perspective on cut-elimination for non-wellfounded sequent calculi with least and greatest fixed-points. In Revantha Ramanayake and Josef Urban, editors, *Automated Reasoning with Analytic Tableaux and Related Methods - 32nd International Conference, TABLEAUX 2023, Prague, Czech Republic, September 18-21, 2023, Proceedings*, volume 14278 of *Lecture Notes in Computer Science*, pages 203–222. Springer, 2023. doi:10.1007/978-3-031-43513-3_12.

[SHK+16] Nikhil Swamy, Cătălin Hriţcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, et al. Dependent types and multi-monadic effects in f. In *Proceedings of the 43rd annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 256–270, 2016.

[SVV18] Amr Sabry, Benoit Valiron, and Juliana Kaizer Vizzotto. From symmetric pattern-matching to quantum control. In Christel Baier and Ugo Dal Lago, editors, *Proceedings of the 21st International Conference on Foundations of Software Science and Computation Structures (FOSSACS'18)*, volume 10803 of *Lecture Notes in Computer Science*, pages 348–364, Thessaloniki, Greece, 2018. Springer. doi:10.1007/978-3-319-89366-2_19.

[TA15] Michael Kirkedal Thomsen and Holger Bock Axelsen. Interpretation and programming of the reversible functional language RFUN. In Ralf Lämmel, editor, *Proceedings of the 27th Symposium on the Implementation and Application of Functional Programming Languages, IFL 2015, Koblenz, Germany, September 14-16, 2015*, pages 8:1–8:13. ACM, 2015. doi:10.1145/2897336.2897345.

[Vie95] Carlin James Vieri. *Pendulum–a reversible computer architecture*. PhD thesis, Massachusetts Institute of Technology, 1995.

[YAG12] Tetsuo Yokoyama, Holger Bock Axelsen, and Robert Glück. Towards a reversible functional language. In Alexis De Vos and Robert Wille, editors, *Revised Papers of the Third International Workshop on Reversible Computation (RC'11)*, volume 7165 of *Lecture Notes in Computer Science*, pages 14–29, Gent, Belgium, 2012. Springer. doi:10.1007/978-3-642-29517-1_2.