A Curry-Howard Correspondence for Linear, Reversible Computation

Kostia Chardonnet
Université Paris-Saclay, CNRS, ENS Paris-Saclay, LMF, 91190, Gif-sur-Yvette, France.
Équipe Quacs, Inria

Alexis Saurin
Université Paris Cité, CNRS, IRIF, 75013, Paris, France.
Équipe π², Inria

Benoît Valiron
Université Paris-Saclay, CNRS, CentraleSupélec, ENS Paris-Saclay, LMF, 91190, Gif-sur-Yvette, France
Équipe Quacs, Inria

Abstract

In this paper, we present a linear and reversible programming language with inductives types and recursion. The semantics of the languages 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 give a Curry-Howard correspondence with the logic $\mu$MALL: linear logic extended with least fixed points allowing inductive statements. The critical part of our work is to show how primitive recursion yields circular proofs that satisfy $\mu$MALL validity criterion and how the language simulates the cut-elimination procedure of $\mu$MALL.

2012 ACM Subject Classification Theory of computation → Linear logic; Theory of computation → Equational logic and rewriting

Keywords and phrases Reversible Computation, Linear Logic, Curry-Howard

1 Introduction

Computation and logic are two faces of the same coin. For instance, consider a proof $s$ of $A \rightarrow B$ 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 $A \rightarrow B$ are symbolized with vertical ellipses. The ellipsis annotated with $s$ indicates that $s$ is a complete proof of $A \rightarrow B$ while $t$ stands for a complete proof of $A$.

This connection is known as the Curry-Howard correspondence [12, 16]. 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 $A \rightarrow B$ 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 an argument (i.e. $t$) of type $A$ to a function (i.e. $s$) of type $A \rightarrow B$. 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 $A$. 

© Kostia Chardonnet and Alexis Saurin and Benoît Valiron; licensed under Creative Commons License CC-BY 4.0
42nd Conference on Very Important Topics (CVIT 2016). Editors: John Q. Open and Joan R. Access; Article No. ; pp. :1–:29
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany
This yields a direct proof of $B$ without any invocation of the “lemma” $A \rightarrow B$.

Paving the way toward the verification of critical softwares, the Curry-Howard correspondence provides a versatile framework. It has been used to mirror first and second-order logics with dependent-type systems [11, 18], separation logics with memory-aware type systems [21, 17], resource-sensitive logics with differential privacy [14], logics with monads with reasoning on side-effects [25, 19], etc.

Reversible computation is a paradigm of computation which emerged as an energy-preserving model of computation in which data is never erased [1] that makes sure that, given some process $f$, there always exists an inverse process $f^{-1}$ such that $f \circ f^{-1} = 1_l = f^{-1} \circ f$. Many aspects of reversible computation have been considered, such as the development of reversible Turing Machines [3], reversible programming languages [6] and their semantics [4, 5]. However, the formal relationship between a logical system and a computational model have not been developed yet.

This paper aims at proposing a type system featuring inductive types for a purely linear and reversible language. We base our study on the approach presented in [23]. In this model, reversible computation is restricted to two main types: 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 $1 \oplus 1$, where $1$ is a type with only one element. The language in [23] offers the possibility to code isos —reversible maps— with pattern matching. An iso is for instance the swap operation, typed with $A \otimes B \leftrightarrow B \otimes A$. However, if [23] hints at an extension towards pure quantum computation, the type system is not formally connected to any logical system.

The problem of reversibility between finite type of same cardinality simply requires to check that the function is injective. That is no longer the case when we work with types of infinite cardinality such as natural numbers.

The main contribution of this work is a Curry-Howard correspondence for a purely reversible typed language in the style of [23], with added generalised inductive types and terminating recursion, enforced by the fact that recursive functions must be structurally recursive: each recursive call must be applied to a decreasing argument. We show how ensuring exhaustivity and non-overlapping of the clauses of the pattern-matching are enough to ensure reversibility and that the obtained language can encode any Primitive Recursive function [28]. For the Curry-Howard part, we capitalize on the logic $\mu$MALL [10, 8]: an extension of the additive and multiplicative fragment of linear logic 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 a reversible type system. In the literature, multiple proofs systems have been considered for $\mu$MALL, some finitary proof system with explicit induction inferences à la Park [10] as well as non-well-founded proof systems which allow to build infinite derivation [8, 9]. The present paper focuses on the latter. In general, an infinite derivation is called a pre-proof and is not necessarily consistent. To solve this problem $\mu$MALL comes equipped with a validity criterion, telling us when an infinite derivation can be considered as a logical proof. We show how the syntactical constraints of being structurally recursive imply the validation of pre-proofs.

**Organisation of the paper** The paper is organised as follows: in Section 2 we present the language, its syntax, typing rules and semantics and show that any function that can be encoded in our language represents an isomorphism. In Section 3 we show that our language can encode any Primitive Recursive Function [28], this is shown by encoding the set of Recursive Primitive Permutations [27] functions. Then in Section 4, we develop on the Curry-Howard Correspondence part: we show, given a well-typed term from our language, how to translate it into a circular
derivation of the logic $\mu$MALL and show that the given derivation respects the validity condition and how our evaluation strategy simulates the cut-elimination procedure of the logic. More details on proofs can be found in the appendix.

## 2 First-order Isos

Our language is based on the one introduced by Sabry et al [23] which define isomorphisms between various types, included the type of lists. We build on the reversible part of the paper by extending the language to support both a more general rewriting system and more general inductive types. The language is defined by layers. Terms and types are presented in Table 1, while typing derivations, inspired from $\mu$MALL, can be found in Tables 2 and 3. The language consists of the following pieces.

**Basic type.** They allow us to construct first-order terms. The constructors $\mathrm{inj}_l$ and $\mathrm{inj}_r$ represent the choice between either the left or right-hand side of a type of the form $A \otimes B$; the constructor $\langle, \rangle$ builds pairs of elements (with the corresponding type constructor $\otimes$); $\mathrm{fold}$ represents inductive structure of the types $\mu X.A$. 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 $\langle x_1, \ldots, x_n \rangle$ or $\overrightarrow{a}$ when $n$ is non-ambiguous and $\langle A \otimes \cdots \otimes A_n \rangle$ for $A \otimes (\cdots \otimes A_n)$ and $A^n$ for $A \otimes \cdots \otimes A$.

**First-order isos.** An iso of type $A \leftrightarrow B$ acts on terms of base types. An iso is a function of type $A \leftrightarrow B$, defined as a set of clauses of the form $\{ v_1 \leftrightarrow e_1 \mid \ldots \mid v_n \leftrightarrow e_n \}$. In the clauses, the tokens $v_i$ are open values and $e_i$ are expressions. In order to apply an iso to a term, the iso must be of type $A \leftrightarrow B$ and the term of type $A$. In the typing rules of isos, the $\mathrm{OD}_A(\{ v_1, \ldots, v_n \})$ predicate (adapted from [23]) syntactically enforces the exhaustivity and non-overlapping conditions on a set of well-typed values $v_1, \ldots, v_n$ of type $A$. The typing conditions make sure that both the left-hand-side and right-hand-side of clauses satisfy this condition. Its formal definition can be found in Table 4 where $\mathrm{Val}(e)$ is defined as $\mathrm{Val}(\mathrm{let} \ p = \omega' \ in \ e) = \mathrm{Val}(e)$, and $\mathrm{Val}(v) = v$ otherwise. These checks are crucial to make sure that our isos are indeed reversible. In the rule $\mathrm{OD}_{A \otimes B}$, we define $S_1^A$ and $S_2^A$ respectively as $\{ w \mid \langle v, w \rangle \in S \}$ and $\{ w \mid \langle w, v \rangle \in S \}$ Exhaustivity for an iso $\{ v_1 \leftrightarrow e_1 \mid \ldots \mid v_n \leftrightarrow e_n \}$ of type $A \leftrightarrow B$ 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 $\mathrm{inj}_l.v$ and $\mathrm{inj}_r.v'$ are non-overlapping while a variable $x$ is always exhaustive. The construction $\mathrm{fix} g.\omega$ represents the creation of a recursive function, rewritten as $\omega[g := \mathrm{fix} \ g.\omega]$ by the operational semantics. Each recursive function needs to satisfy a structural recursion criteria: making sure that one of the input arguments strictly decreases on each recursive call. Indeed, since isos can be non-terminating (due to recursion), we need a criterion that implies termination to ensure that we work with total functions. If $\omega$ is of type $A \leftrightarrow B$, we can build its inverse $\omega^\dagger : B \leftrightarrow A$ and show that their composition is the identity. In order to avoid conflicts between variables we will always work up to $\alpha$-conversion and use Barendregt’s convention [7, p.26] which consists in keeping all bound and free variables names distinct, even when this remains implicit.

The type system is split in two parts: one for terms (noted $\Delta; \Psi \vdash t : A$) and one for isos (noted $\Psi \vdash \omega : A \leftrightarrow B$). In the typing rules, the contexts $\Delta$ are sets of pairs that consist of a term-variable and a base type, where each variable can only occur once and $\Psi$ is a singleton set of a pair of an iso-variable and an iso-type association.

**Definition 1 (Structurally Recursive).** Given an iso $\mathrm{fix} f.\{ v_1 \leftrightarrow e_1 \mid \ldots \mid v_n \leftrightarrow e_n \} : A_1 \otimes \cdots \otimes A_n \leftrightarrow C$, it is structurally recursive if there is $1 \leq j \leq m$ such that $A_j = \mu X.B$ and for all $i \in \{1, \ldots, m\}$ we have that $v_i$ is of the form $\langle v_1^i, \ldots, v_m^i \rangle$ such that $v_i^j$ is either:

CVIT 2016
(Base types) \( A, B \ ::= \mathbb{I} \mid A \oplus B \mid A \otimes B \mid \mu X.A \)

(Isos, first-order) \( \alpha ::= A \leftrightarrow B \)

(Values) \( v ::= () \mid x \mid \text{inj}_1 v \mid \text{inj}_r v \mid \langle v_1, v_2 \rangle \mid \text{fold} v \)

(Pattern) \( p ::= x \mid \{ p_1, p_2 \} \)

(Expressions) \( e ::= v \mid \text{let} p_1 = p_2 \text{ in } e \)

(Isos) \( \omega ::= \{ v_1 \leftrightarrow e_1 \mid \ldots \mid v_n \leftrightarrow e_n \} \mid \text{fix } f. \omega \mid f \)

(Terms) \( t ::= () \mid x \mid \text{inj}_1 t \mid \text{inj}_r t \mid \langle t_1, t_2 \rangle \mid \text{fold } t \mid \omega t \mid \text{let } p = t_1 \text{ in } t_2 \)

\[\begin{array}{ll}
\emptyset; \Psi \vdash e \mathbf{c}t & A; \Psi \vdash e \mathbf{c}t : A \\
\Delta; \Psi \vdash e \mathbf{c}t : A & \Delta; \Psi \vdash e \mathbf{c}t : B \\
\Delta_1; \Psi \vdash t_1 : A & \Delta_2; \Psi \vdash t_2 : B \\
\Delta_1, \Delta_2; \Psi \vdash \{ t_1, t_2 \} : A \oplus B & \Delta_1, \Delta_2; \Psi \vdash \{ X \leftarrow \mu X.A \} \\
\Delta; \Psi \vdash \text{inj}_1 t : A \oplus B & \Delta; \Psi \vdash \text{inj}_2 t : A \oplus B \\
\Delta_1, \Delta_2; \Psi \vdash \text{fold } t : \mu X.A \\
\Delta; \Psi \vdash \text{fold } t : A & \Delta; \Psi \vdash \omega t : \omega t \\
\Delta_1; \Psi \vdash t_1 : A_1 \otimes \cdots \otimes A_n & \Delta_2, x_1 : A_1, \ldots, x_n : A_n; \Psi \vdash t_2 : B \\
\Delta_1, \Delta_2; \Psi \vdash \text{let } (x_1, \ldots, x_n) = t_1 \text{ in } t_2 : B
\end{array}\]

**Table 1** Terms and types

**Table 2** Typing of terms and expressions

A closed value, in which case \( e_i \) does not contain the subterm \( f p \)

Open, in which case for all subterm of the form \( f p \) in \( e_i \) we have \( p = (x_1, \ldots, x_m) \) and \( x_j : \mu X.B \) is a strict subterm of \( v_i^j \).

Given a clause \( v \leftrightarrow e_i \), we call the value \( v_i^j \) (resp. the variable \( x_j \)) the decreasing argument (resp. the focus) of the structurally recursive criterion.

\[\begin{array}{l}
\text{Remark 2. As we are focused on a very basic notion of structurally recursive function, the typing rules of isos allow to have at most one iso-variable in the context, meaning that we cannot have intertwined recursive call.}
\end{array}\]

Finally, our language is equipped with a rewriting system \( \rightarrow \) on terms, defined in Definition 5, that follows a deterministic call-by-value strategy: each argument of a function is fully evaluated before applying the substitution. 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 form. The evaluation of an iso applied to a value relies on with pattern-matching: the argument is matched against the left-hand-side of each clause until one of them matches (written \( \sigma[v] = v' \)), in which case the pattern-matching, as defined in Definition 5, returns a substitution \( \sigma \) that sends variables to values. Because we ensure exhaustivity and non-overlapping (Lemma 3), the pattern-matching can always occur on well-typed terms. The **support** of a substitution \( \sigma \) is defined as \( \text{supp}(\sigma) = \{ x \mid (x \mapsto v) \in \sigma \} \).
\[ \Delta_1 \vdash_\varepsilon v_1 : A \quad \ldots \quad \Delta_n \vdash_\varepsilon v_n : A \quad \text{OD}_{A}(\{v_1, \ldots, v_n\}) \]
\[ \Delta_i : \forall v \in B \quad \Delta_n : \forall v \in B \quad \text{OD}_{B}(\{\text{Val}(e_1), \ldots, \text{Val}(e_n)\}) \]
\[ \Psi \vdash_\omega \{v_1 \leftrightarrow e_1 \mid \ldots \mid v_n \leftrightarrow e_n\} : A \leftrightarrow B. \]

\[ f : \alpha \vdash_\omega f : \alpha \]

\[ f : \alpha \vdash_\omega \omega : \alpha \quad \text{fix } f.\omega \text{ is structurally recursive} \]

\[ \Psi \vdash_\omega \text{fix } f.\omega : \alpha \]

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|}
\hline
\text{Table 3 Typing of isos} & \text{OD}_{A}(\{x\}) & \text{OD}_{A}(\{()\}) \\
\hline
& \text{OD}_{A}(\{x\}) & \text{OD}_{A}(\{()\}) \\
\hline
\hline
\text{OD}_{A}(X \leftrightarrow_{\mu X} \mu A)(S) & \text{OD}_{A}(\{\text{fold} v \mid v \in S\}) \\
\hline
\text{OD}_{\mu X}(\{\text{fold} v \mid v \in S\}) & \text{OD}_{B}(\{\text{fold} v \mid v \in T\}) \\
\hline
\end{tabular}
\end{table}

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|}
\hline
\text{Table 4 Exhaustivity and Non-Overlapping} & \text{OD}_{A}(S) & \text{OD}_{B}(T) \\
\hline
\text{let } X = \{v_1, \ldots, v_n\} \text{ st } \text{OD}_{A}(X), \text{forall } v \in X, \text{OD}_{B}(S_v) \\
\hline
\text{or let } Y = \{v'_1, \ldots, v'_n\} \text{ st } \text{OD}_{B}(Y), \text{forall } v \in Y, \text{OD}_{A}(S_v) \\
\hline
\end{tabular}
\end{table}

\begin{itemize}
\item \textbf{Lemma 3} (\text{OD}_{A}(A) ensures exhaustivity and non-overlapping.). Let \text{OD}_{A}(S) and \vdash_\varepsilon v : A, then there exists a unique \(v' \in S\) such that \(\sigma[v'] = v\).
\item \textbf{Definition 4} (Substitution). Applying substitution \(\sigma\) on an expression \(t\), written \(\sigma(t)\) is defined, as:
\[ \sigma(()) = (), \sigma(x) = v \text{ if } \{x \mapsto v\} \subseteq \sigma, \sigma(\text{inj}_r v) = \text{inj}_r, \sigma(t) = \text{inj}_r, \sigma(\inj_l t) = \inj_l, \sigma((t, t')) = (\sigma(t), \sigma(t')), \sigma(\omega t) = \omega(\sigma(t)) \text{ and } \sigma(\text{let } p = t_1 \text{ in } t_2) = (\text{let } p = \sigma(t_1) \text{ in } \sigma(t_2)) \]
\item \textbf{Definition 5} (Evaluation relation \(\rightarrow\)). We define \(\rightarrow\) the rewriting system of our language as follows:
\[ \frac{t_1 \rightarrow t_2}{C[t_1] \rightarrow C[t_2]} \quad \text{Cong} \quad \frac{\sigma[p] = v}{\text{let } p = v \text{ in } t \rightarrow \sigma(t)} \quad \text{LetE} \quad \frac{(\text{fix } f.\omega) \rightarrow \omega[f := (\text{fix } f.\omega)]}{\text{IsoRec}} \quad \sigma[v_i] = v' \]
\[ \frac{\{v_1 \leftrightarrow e_1 \mid \ldots \mid v_n \leftrightarrow e_n\} v' \rightarrow \sigma(e_i) }{\text{IsoApp}} \]
\end{itemize}

\begin{itemize}
\item As usual we note \(\rightarrow^*\) for the reflexive transitive closure of \(\rightarrow\).
\end{itemize}

As mentioned above, from any iso \(\omega : A \leftrightarrow B\) we can build its inverse \(\omega^\perp : B \leftrightarrow A\), the inverse operation is defined inductively on \(\omega\) and is given in Definition 6.

\begin{itemize}
\item \textbf{Definition 6} (Inversion). Given an iso \(\omega\), we define its dual \(\omega^\perp\) as:
\[ f^\perp = f, (\text{fix } f.\omega^\perp) = \text{fix } f.\omega^\perp, \{(v_i \leftrightarrow e_i)_{i \in I}\}^\perp = \{(v_i \leftrightarrow e_i^\perp)_{i \in I}\} \]
\end{itemize}

And the inverse of a clause as:
\[ \left( \begin{array}{c}
\{v_1 \leftrightarrow \text{let } p_1 = \omega_1 p'_1 \text{ in } \}^\perp \\
\{ \text{let } p_n = \omega_n p'_n \text{ in } v'_n \}
\end{array} \right) = \left( \begin{array}{c}
\{v'_1 \leftrightarrow \text{let } p'_1 = \omega_1 p_1 \text{ in } \}^\perp \\
\{ \text{let } p'_n = \omega_n p_1 \text{ in } v'_1 \}
\end{array} \right).
\]

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

\begin{itemize}
\item \textbf{Lemma 7} (Inversion is well-typed). Given \(\Psi \vdash_\omega \omega : A \leftrightarrow B\), then \(\Psi \vdash_\omega \omega^\perp : B \leftrightarrow A\).
\end{itemize}
In this section, we study the computational content of our language. We show that we can encode Recursive Primitive Permutations [27] (RPP), which shows us that we can encode at least all Primitive Recursive Functions [28].

We give a few reminders on the language RPP and its main results, then show how to encode it.
\[ \begin{align*} x \cdot [S] x + 1 & \quad x \cdot [P] x - 1 & \quad x \cdot [\text{Sign}] - x & \quad x \cdot [\text{Id}] x & \quad x \cdot \left[ X \right] y \cdot x \\

\begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} y_1 \\ \vdots \\ y_n \end{bmatrix} & \quad \begin{bmatrix} x_1 \\ \vdots \\ x_k \end{bmatrix} = \begin{bmatrix} y_1 \\ \vdots \\ y_n \end{bmatrix} & \quad \begin{bmatrix} f \| g \| h \end{bmatrix} = \begin{bmatrix} f \| g \| y_n \end{bmatrix} \\

\begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} f(x_1, \ldots, x_n) \text{ if } x > 0 \\ g(x_1, \ldots, x_n) \text{ if } x = 0 \\ h(x_1, \ldots, x_n) \text{ if } x < 0 \end{bmatrix} & \quad \begin{bmatrix} y_1 \\ \vdots \\ y_n \end{bmatrix} = \begin{bmatrix} f; \ldots; f(x_1, \ldots, x_n) \end{bmatrix} \\

\begin{bmatrix} f \| g \| h \end{bmatrix} = \begin{bmatrix} f \| g \| y_n \end{bmatrix} & \quad \begin{bmatrix} f \| g \| h \end{bmatrix} = \begin{bmatrix} f \| g \| h \end{bmatrix} \\

\end{align*} \]

**Figure 2** Generators of RPP

### 3.1 Reminder on RPP

RPP is a set of integer-valued functions of variable arity, we define it by arity as follows: we note \( RPP^k \) for the set of functions in RPP from \( \mathbb{Z}^k \) to \( \mathbb{Z} \), it is built inductively on \( k \in \mathbb{N} \) by: the successor \( (S) \), the predecessor \( (P) \), the identity \( (\text{Id}) \) and the sign-change that are part of \( RPP^1 \). The swap function \( (X) \) and the binary permutation \( \text{Id} \) are part of \( RPP^2 \) and then, for any function \( f, g, h \in RPP^k \) and \( j \in RPP^l \), we can build (i) the sequential composition \( f; g \in RPP^k \), (ii) the parallel composition \( f \| j \in RPP^{k+l} \) (iii) the iterator \( \text{If}[f] \in RPP^{k+1} \) and (iv) the selection \( \text{If}[f, g, h] \in RPP^{k+1} \).

Finally, the set of all functions that form RPP is taken as the union for all \( k \) all of the \( RPP^k \):

\[ RPP = \bigcup_{k \in \mathbb{N}} RPP^k \]

We present the semantics of each constructor of RPP under a graphical form, as in [27], where the left-hand-side variables of the diagram represent the input of the function and the right-hand-side is the output of the function. The semantics of all those operators are given in Figure 2.

- **Remark 14.** In their paper [27], the authors make use of two other constructors: generalised permutations over \( \mathbb{Z}^k \) and weakenings of functions, but those can actually be defined from the other constructors so that in the following section we do not give their encoding.

Then, if \( f \in RPP^k \) we can define an inverse \( f^{-1} \):

**Definition 15** (Inversion). The inversion is defined as follow:

\[
\begin{align*}
\text{Id}^{-1} &= \text{Id} & \quad \text{S}^{-1} &= P \\
\text{Sign}^{-1} &= \text{Sign} & \quad \text{X}^{-1} &= X \\
(g; f)^{-1} &= f^{-1} \| g^{-1} & \quad (\text{If}[f])^{-1} &= \text{If}[f^{-1}] \\
(f; g; h)^{-1} &= \text{If}[f^{-1}, g^{-1}, h^{-1}] \\
\end{align*}
\]

**Proposition 16** (Inversion defines an inverse [27]). Given \( f \in RPP^k \) then \( f^{-1} = \text{Id} = f^{-1}; f \)
Theorem 17 (Soundness & Completeness [27]). RPP is PRF-Complete and PRF-Sound: it can represent any Primitive Recursive Function and every function in RPP can be represented in PRF.

3.2 From RPP to Isos

We start by defining the type of strictly positive natural numbers, npos, as npos = μX.1 ⊕ X. We define \( n \), the encoding of a positive natural number into a value of type npos as \( 1 = \text{fold inj}_1() \) and \( n + 1 = \text{fold inj}_n(n) \). Finally, we define the type of integers as \( \mathbb{Z} = 1 \oplus (\text{npos \oplus \text{npos}}) \) along with \( \overline{z} \) the encoding of any \( z \in \mathbb{Z} \) into a value of type \( \mathbb{Z} \) defined as: \( \overline{0} = \text{inj}_1() \), \( \overline{z} = \text{inj}_n \text{inj}_1 \overline{z} \) for \( z \) positive, and \( \overline{z} = \text{inj}_n \text{inj}_r \overline{-z} \) for \( z \) negative. Given some function \( f \in \text{RPP}^k \), we will build an iso \( \text{isos}(f) : Z^k \leftrightarrow Z^k \) which simulates \( f \). \( \text{isos}(f) \) is defined by the size of the proof that \( f \) is in \( \text{RPP}^k \).

Definition 18 (Encoding of the primitives).

- The Sign-change is
  \[
  \begin{align*}
  \text{inj}_n \text{inj}_j x & \leftrightarrow \text{inj}_n \text{inj}_j x \\
  \text{inj}_n \text{inj}_j x & \leftrightarrow \text{inj}_n \text{inj}_j x \\
  \text{inj}_n \text{inj}_j() & \leftrightarrow \text{inj}_n() \\
  \text{inj}_n \text{inj}_j \text{fold inj}_j() & \leftrightarrow \text{inj}_n() \\
  \text{inj}_n \text{inj}_j \text{fold inj}_j x & \leftrightarrow \text{inj}_n() \\
  \end{align*}
  \]

- The Successor is
  \[
  \begin{align*}
  \text{inj}_n() & \leftrightarrow \text{inj}_n \text{inj}_j \text{fold inj}_j() \\
  \text{inj}_n \text{inj}_j x & \leftrightarrow \text{inj}_n \text{inj}_j \text{fold inj}_j x \\
  \end{align*}
  \]

- The Predecessor is the inverse of the Successor

Definition 19 (Encoding of Composition). Let \( f, g \in \text{RPP}^j \), \( \omega_f = \text{isos}(f) \) and \( \omega_g = \text{isos}(g) \) the iso encoding \( f \) and \( g \), we build \( \text{isos}(f; g) \) of type \( Z^j \leftrightarrow Z^j \) as:

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

Definition 20 (Encoding of Parallel Composition). Let \( f \in \text{RPP}^j \) and \( g \in \text{RPP}^k \), and \( \omega_f = \text{isos}(f) \) and \( \omega_g = \text{isos}(g) \), we define \( \text{isos}(f \parallel g) \) of type \( Z^{j+k} \leftrightarrow Z^{j+k} \) as:

\[
\text{isos}(f \parallel g) = \left\{ \begin{array}{ccc}
(x_1, \ldots, x_j, y_1, \ldots, y_k) & \leftrightarrow & \text{let } (x'_1, \ldots, x'_j) = \omega_f(x_1, \ldots, x_j) \text{ in } \\
(x'_{1'}, \ldots, x'_{j'}, y'_1, \ldots, y'_k) & \leftrightarrow & \text{let } (y'_1, \ldots, y'_k) = \omega_g(y_1, \ldots, y_k) \text{ in } \\
\end{array} \right\}
\]

Definition 21 (Encoding of Finite Iteration). Let \( f \in \text{RPP}^k \), and \( \omega_f = \text{isos}(f) \), we encode the finite iteration \( \text{It} \{ f \} \in \text{RPP}^{k+1} \) with the help of an auxiliary iso, \( \omega_{aux} \), of type \( Z^k \otimes \text{npos} \leftrightarrow Z^k \otimes \text{npos} \) doing the finite iteration using \( \text{npos} \), defined as:

\[
\omega_{aux} = \text{fixg}. \begin{cases} 
\overline{\omega} & \leftrightarrow \text{let } \overline{\text{f}} = \omega_f \overline{\omega} \text{ in } \\
(\overline{\text{f}}, \text{fold inj}_1()) & \leftrightarrow \text{let } \overline{\text{f}} = \omega_f \overline{\text{f}} \text{ in } \\
(\overline{\text{f}}, \text{fold inj}_n) & \leftrightarrow \text{let } (\overline{\text{f}}, n) = \omega_f (\overline{\text{f}}, n) \text{ in } \\
(\overline{\text{f}}, \text{fold inj}_n, n) & \leftrightarrow \text{let } (\overline{\text{f}}, n') = \omega_f (\overline{\text{f}}, n) \text{ in }
\end{cases}
\]
We can now properly define $\text{isos}(I[f])$ of type $Z^{k+1} \leftrightarrow Z^{k+1}$ as:

$$\text{isos}(I[f]) = \begin{cases} 
(\overline{x}, \text{inj}_j()) & \leftrightarrow (\overline{x}, \text{inj}_j()) \\
(\overline{x}, \text{inj}_i, \text{inj}_j()) & \leftrightarrow \text{let } (\overline{y}, z) = \omega_{aux}(\overline{x}, z) \text{ in } (\overline{y}, \text{inj}_i, \text{inj}_j(), z') \\
(\overline{x}, \text{inj}_i, \text{inj}_j, z) & \leftrightarrow \text{let } (\overline{y}, z) = \omega_{aux}(\overline{x}, z) \text{ in } (\overline{y}, \text{inj}_i, \text{inj}_j, z') 
\end{cases}$$

**Definition 22 (Encoding of Selection).** Let $f, g, h \in \text{RPP}_k$ and $\omega_f = \text{isos}(f), \omega_g = \text{isos}(g), \omega_h = \text{isos}(h)$. We define $\text{isos}(I[f, g, h])$ of type $Z^{k+1} \leftrightarrow Z^{k+1}$ as:

$$\text{isos}(I[f, g, h]) = \begin{cases} 
(\overline{x}, \text{inj}_j()) & \leftrightarrow \text{let } \overline{x} = \omega_f(\overline{x}) \text{ in } (\overline{x}, \text{inj}_j(), z) \\
(\overline{x}, \text{inj}_i()) & \leftrightarrow \text{let } \overline{x} = \omega_g(\overline{x}) \text{ in } (\overline{x}, \text{inj}_i(), z) \\
(\overline{x}, \text{inj}_i, \text{inj}_j, z) & \leftrightarrow \text{let } \overline{x} = \omega_h(\overline{x}) \text{ in } (\overline{x}, \text{inj}_i, \text{inj}_j, z) 
\end{cases}$$

**Theorem 23 (The encoding is well-typed).** Let $f \in \text{RPP}_k$, then $\vdash_\omega \text{isos}(f) : Z^k \leftrightarrow Z^k$.

**Theorem 24 (Simulation).** Let $f \in \text{RPP}_k$ and $n_1, \ldots, n_k$ elements of $\mathbb{Z}$ such that $f(n_1, \ldots, n_k) = (m_1, \ldots, m_k)$ then $\text{isos}(f)(m_1, \ldots, m_k) \rightarrow^* (m_1, \ldots, m_k)$

**Remark 25.** Notice that $\text{isos}(f)^\perp \neq \text{isos}(f^{-1})$, due to the fact that $\text{isos}(f)^\perp$ will inverse the order of the let constructions, which will not be the case for $\text{isos}(f^{-1})$. They can nonetheless be considered equivalent up to a permutation of let constructions and renaming of variable.

### 4 Proof Theoretical Content

We want to relate our language of isos to proofs in a suitable logic. As mentioned earlier, an iso $\vdash_\omega A \leftrightarrow B$ 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 $A$. Therefore we want to be able to translate an iso into a proof isomorphism: two proofs $\pi$ and $\pi^\perp$ of respectively $A \vdash B$ and $B \vdash A$ such that their composition reduces through the cut-elimination to the identity either on $A$ or on $B$ depending on the way we make the cut between those proofs.

Since we are working in a linear system with inductive types we will use an extension of Linear Logic called $\mu\text{MALL}$: linear logic with least and greatest fixed points, which allows us to reason about inductive and coinductive statements. $\mu\text{MALL}$ also allows us to consider infinite derivation trees, which is required as our isos can contain recursive variables. We need to be careful though: infinite derivations cannot always be considered as proofs, hence $\mu\text{MALL}$ comes with a validity criterion on infinite derivations trees (called pre-proofs) that tells us whether such derivations are indeed proofs. We recall briefly the basic notions of $\mu\text{MALL}$, while more details can be found in [9].

### 4.1 Background on $\mu\text{MALL}$

Given an infinite set of variables $V = \{X, Y, \ldots\}$, we call the set of *formula* of $\mu\text{MALL}$ the objects generated by $A, B ::= X \mid 0 \mid T \mid \bot \mid A \land B \mid A \lor B \mid A \otimes B \mid A \vee B \mid A \& B \mid \mu X.A \mid \nu X.A$ where $\mu$ and $\nu$ bind the variable $X$ in $A$. The negation on formula is defined in the usual way: $X^\perp = X, 0^\perp = T, 1^\perp = \bot, (A \land B)^\perp = A^\perp \otimes B^\perp, (A \lor B)^\perp = A^\perp \lor B^\perp, (\mu X.A)^\perp = \mu X.A^\perp$ having $X^\perp = X$ is harmless since we only deal with close formulas.

We call *occurrences*, a word of the form $A \cdot w$ where $A \in \mathfrak{A}_{\text{resh}}$ an infinite set of *atomic addresses* and its dual $\mathfrak{A}_{\text{resh}}^\perp = \{A^\perp \mid A \in \mathfrak{A}_{\text{resh}}\}$ and $w$ a word over $\{l, r, i\}^*$ (for *left, right and inside*) and *formulas occurrences* $F, G, H, \ldots$ as a pair of a formula and an occurrence, written $\langle A, \alpha \rangle$. Finally we write $\Sigma, \Phi$ for *formula contexts*: sets of formulas occurrences. We write $A, \alpha \equiv B, \beta$ when $A = B$. [9]
Negation is lifted to formulas with \((A_\alpha)\perp = A_\perp^\perp\) where \((\alpha \cdot w)^\perp = \alpha^\perp \cdot w\) and \((\alpha^\perp \cdot w)^\perp = \alpha \cdot w\). In general, we write \(\alpha,\beta\) for occurrences.

The connectives need then to be lifted to occurrences as well:

\begin{itemize}
  \item Given \(\# \in \{\otimes, \oplus, \\&\},\) if \(F = A_{\alpha l}\) and \(G = B_{\alpha r}\) then \((F \# G) = (A \# B)_\alpha\)
  \item Given \(\# \in \{\mu, \nu\}\) if \(F = A_{\alpha i}\) then \(#X.F = (\#X.A)_\alpha\)
\end{itemize}

Occurrences allow us to follow a subformula uniquely inside a derivation. Since in \(\mu\)MALL we only works with formula occurrences, we simply use the term formula.

The (possibly infinite) derivation trees of \(\mu\)MALL, called pre-proofs are coinductively generated by the rules given in Figure 3. We say that a formula is principal when it is the formula that the rule is being applied to.

Among the infinite derivations that \(\mu\)MALL offer we can look at the circular ones: an infinite derivation is circular if it has /finitely many different subtrees. The circular derivation can therefore be represented in a more compact way with the help of back-edges: arrows in the derivation that represent a repetition of the derivation. Derivations with back-edge are represented with the addition of sequents marked by a back-edge label, noted \(\vdash \Sigma^f\), which represent a back-edge pointing to the sequent \(\vdash \Sigma\). We take the convention that from the root of the derivation from to rule \(\Sigma^f\) there must be exactly one sequent annotated by \(\Sigma\).

▶ Example 26. An infinite derivation and two different circular representations with back-edges.

\begin{itemize}
  \item While a circular proof has multiple /finitely representations (depending on where the back-edge is placed), they can all be mapped back to the same infinite derivation via an infinite unfolding of the back-edge and forgetting the back-edge labels:

▶ Definition 27 (Unfolding). We define the unfolding of a circular derivation \(P\) with a valuation \(v\) from back-edge labels to derivations by:

\[
\mathcal{U}(P, v) = \mathcal{U}(P_1, v), \ldots, \mathcal{U}(P_n, v)
\]
\[ = \mathcal{U}(\text{be}(f), v) = v(f) \]
\[ = \mathcal{U}\left( P : \vdash f \sum \ r \ \right) = \left( \pi = \mathcal{U}(P_1, v'), \ldots, \mathcal{U}(P_n, v') \ \right) \text{ with } v'(g) = \pi \text{ if } g = f \]

\(\mu\text{MALL}\) comes with a validity criterion on pre-proofs that determines when a pre-proof can be considered as a proof: mainly, whether or not each infinite branch can be justified by a form of coinductive reasoning. The criterion also ensures that the cut-elimination procedure holds. For that, we can define a notion of thread [8, 9]: an infinite sequence of tuples of formulas, sequents and directions (either up or down). Intuitively, these threads follow some formula starting from the root of the derivation and start by going up. If the thread encounters an axiom rule, it will bounce back and start going down in the dual formula of the axiom rule. It may bounce back again, when going down on a cut rule, if it follows the cut-formula. 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 \(\nu\) formula. For the multiplicative fragment, we say that a pre-proof is valid if for all infinite branches, there exists a valid thread, while for the additive part, we require a notion of additive slices and persistent slices which we do not details here. More details can be found in [9].

### 4.2 Translating isos into \(\mu\text{MALL}\)

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 programs and proofs. We then show that our translation entails a dynamic correspondence between the evaluation procedure of our language and the cut-elimination procedure of \(\mu\text{MALL}\). This will imply that the proofs we obtain are indeed isomorphisms, meaning that if we cut the aforementioned proofs \(\pi\) and \(\pi'\), performing the cut-elimination procedure would give either the identity on \(A\) or the identity on \(B\).

The derivation we obtain are circular and we therefore translate the isos directly into finite derivations with back-edge, written \(\text{circ}(\omega)\). We can define another translation into infinite derivations as the composition of \(\text{circ}\) with the unfolding: \([\omega] = \mathcal{U}(\text{circ}(\omega))\).

Because we need to keep track of which formula is associated to which variable from the typing context, the translation uses a slightly modified version of \(\mu\text{MALL}\) in which contexts are split in two parts, written \(\Upsilon; \Theta\), where \(\Upsilon\) is a list of formulas and \(\Theta\) is a set of formulas associated with a term-variable (written \(x : F\)). When starting the translation of an iso of type \(A \leftrightarrow B\), we start in the context \([A_\alpha]; \emptyset\) (for some address \(\alpha\)) and end in the context \(\emptyset; \Theta\). The additional information of the variable in \(\Theta\) is here to make sure we know how to split the contexts accordingly when needed later during the translation, with respect to the way they are split in the typing derivation. We write \(\overline{\Upsilon} = \{ F \mid x : F \in \Theta \} \) and \(\overline{\Theta} = \{ x : A \mid x : A_\alpha \in \Theta \} \). We also use another rule which allow us to send the first formula from \(\Upsilon\) to \(\Theta\) and affect it a variable: \(\Upsilon; \Theta \vdash G \\text{ex}(x)\). Given a derivation \(\iota\) in this system, we write \([\iota]\) for the function that sends \(\iota\) into a derivation of \(\mu\text{MALL}\) where (i) we remove all occurrence of the exchange rule (ii) the contexts \([\_]; \Theta\) becomes \(\emptyset\).

Given an iso \(\omega : A \leftrightarrow B\) and initial addresses \(\alpha, \beta\), its translation into a derivation of \(\mu\text{MALL}\) of \(A_\alpha \vdash B_\beta\) is described with three separate phases:

#### Iso Phase
The first phase consists in travelling through the syntactical definition of an iso, keeping as information the last encountered iso-variable bounded by a \(\text{fix } f.\omega\) and calling the negative phase when encountering an iso of the form \(\{ e_1 \leftrightarrow v_1 | \ldots | e_n \leftrightarrow v_n \}\) and attaching to the formulas \(A\) and \(B\) two distinct addresses \(\alpha\) and \(\beta\) and to the sequent as a label of name of the last encountered iso-variable. Later on during the translation, this phase will be recalled when
encountering another iso in one of the \( \epsilon_i \), and, if said iso correspond to an iso-variable, we will create a back-edge pointing towards the corresponding sequents.

**Negative Phase.** Starting from some context \([A_\alpha], \Theta\), the negative phase consists into decomposing the formula \( A \) according to the in which way the values of type \( A \) on the left-hand-side of \( \omega \) are decomposed. The negative phase works as follows: we consider a set of (list of values, typing judgement), written \((l, \xi)\) where each element of the set corresponds to one clause \( v \leftrightarrow e \) of the given iso and \( \xi \) is the typing judgment 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). Each element of the list \( \Upsilon \) will correspond to exactly one value in the list \( l \). If the term that needs to be decomposed is a variable \( x \), then we will apply the \( ex(x) \) rule, sending the formula to the context \( \Theta \). The negative phase ends when the list is empty and hence when \( \Upsilon = \emptyset \). When it is the case, we can start decomposing \( \xi \) and the *positive phase* start. The negative phase is defined inductively on the first element of the list of every sets, which are known by typing to have the same prefix, and is given in Table 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. Keeping the variable-formula pair in the derivation is here to help us know how to split accordingly the context \( \Theta \) when needed, while \( \Upsilon \) is always empty and is therefore omitted. While the positive phase carry over the information of the last-seen iso-variable, it is not noted explicitly as it is only needed when calling the Iso Phase. The positive phase is given in Table 5.

\[\text{Remark 28. While } \mu MALL \text{ is presented in a one-sided way, we write } \Sigma \vdash \Phi \text{ for } \vdash \Sigma^\dagger, \Phi \text{ in order to stay closer to the formalism of the type system of isos.} \]

\[\text{Definition 29. } \text{circ}(\omega, S, \alpha, \beta) = \pi \text{ takes a well-typed iso, a singleton set } S \text{ of an iso-variable corresponding to the last iso-variable seen in the induction definition of } \omega \text{ and two fresh addresses } \alpha, \beta \text{ and produces a circular derivation of the variant of } \mu MALL \text{ described above with back-edges. } \text{circ}(\omega, S, \alpha, \beta) \text{ is defined inductively on } \omega:\]

\[= \text{circ(fix } f.\omega, S, \alpha, \beta) = \text{circ}(\omega, \{f\}, \alpha, \beta)\]

\[= \text{circ}(f, \{f\}, \alpha, \beta) = A_\alpha \vdash B_\beta \text{ be}(f)\]

\[= \text{circ} \{\langle v_1 \leftrightarrow e'_i \rangle_{i \in I} : A \vdash B, \{f\}, \alpha, \beta\} = \Pi \text{ Neg}((\{v_1\}, \xi_i)_{i \in I}) A_\alpha \vdash B_\beta \]

\[\text{where } \xi_i \text{ is the typing derivation of } \epsilon_i.\]

\[\text{Example 30. The translation } \|\text{circ}(\omega, \emptyset, \alpha, \beta)\| \text{ of the iso } \omega \text{ from Example 9 is, with } F = A_{\text{arr}}, G = B_{\text{arr}}, H = C_{\text{arr}} \text{ and } F' = A'_{\text{arr}}, G' = B'_{\text{arr}}, H' = C'_{\text{arr}}:\]

\[
\begin{array}{llll}
F \vdash id & G \vdash G & H \vdash H & \text{id} \\
F \vdash F \oplus G & G \vdash F \otimes G & H \vdash F \otimes G & \text{id} \\
F \vdash H \oplus (F \otimes G) & G \vdash H \oplus (F \otimes G) & H \vdash H \oplus (F \oplus G) & \text{id} \\
F \vdash (G \oplus H) \oplus H \oplus (F \oplus G) & \text{id} \\
\end{array}
\]

\[\text{Example 31. Considering the iso swap of type } A \otimes B \leftrightarrow B \otimes A \text{ and its } \mu MALL \text{ proof } \pi_S = A \otimes B \vdash A \otimes B \text{ and } \]

\[\pi_m(S) \text{, following Example 10 we give its corresponding proof } \pi_m(S) \text{ where } F = (A \otimes B)_{\text{arr}} \text{ and } G = (B \otimes A)_{\text{arr}}, \text{ then } [F] \text{ and } [G] \text{ are respectively of address } \alpha \text{ and } \beta:\]
We painted in blue the pre-thread that follows the \textit{focus} of the structurally recursive criterion. During the \textit{negative phase} which consists of the $\mu$, $\&$, $\otimes$, $\perp$ rules the pre-thread is going up, at each time going into the subformula corresponding to the \textit{focus}. Then, during the \textit{positive phase} the pre-thread is not active during the multiple \textit{cut} rules until it reaches the $\text{id}$ rule, where the pre-thread bounces and starts going down before bouncing back up again in the \textit{cut} rule, into the infinite branch, where the behavior of the pre-thread will repeat itself.

\[
\begin{array}{c}
\text{Neg}\{\{\text{inj}_1 v_j:: l_j, \xi_j\}_{j \in J}\} \cup \{\{\text{inj}_r v_k:: l_k, \xi_k\}_{k \in K}\} = \\
\text{Neg}\{\{v_j:: l_j, \xi_j\}_{j \in J}\} & \text{Neg}\{\{v_k:: l_k, \xi_k\}_{k \in K}\} & \text{Neg}(\{[]\}) = []:: \Theta \vdash F \\
F_1 :: T; \Theta \vdash G & F_2 :: T; \Theta \vdash G & \& \\
F_1 \otimes F_2 :: T; \Theta \vdash G \\
\text{Neg}\{\{v^1_i:: l_i, \xi_i\}_{i \in I}\} & \text{Neg}\{\{v^2_i:: l_i, \xi_i\}_{i \in I}\} & \text{Neg}(\{l, \xi\}) = T :: T; \Theta \vdash F \\
F_1, F_2 :: T; \Theta \vdash G & \text{Neg}(\{l, \xi\}) = T :: T; \Theta \vdash F \\
\text{Neg}(\{\text{fold} v_i:: l_i, \xi_i\}_{i \in I}) & \text{Neg}(\{\text{fold} v_i:: l_i, \xi_i\}_{i \in I}) & \text{Neg}(\{l, \xi\}) = F :: T; \Theta \vdash G
\end{array}
\]

\textbf{Figure 4 Negative Phase}

\textbf{Lemma 32.} Given $\pi = \text{circ}(\omega)$, for each infinite branch of $\pi$, only a single iso-variable is visited infinitely often.

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. ▶
NN:14  A Curry-Howard Correspondence for Linear, Reversible Computation

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

In order to show the validity of our derivation we need, for each infinite branch, to build a valid thread. From the previous lemmas and the syntactical constraints of the language, we have:

Definition 33 (Purely Positive Proof). A Purely Positive Proof is a finite, cut-free proof whose rules are only $\oplus$, $\otimes$, $\mu$, $\top$, $\bot$, if $i \in \{1, 2\}$.

Lemma 34 (Values are Purely Positive Proofs). Given $x_1 : A^1, \ldots, x_n : A^n \vdash v : A$ then $\vdash x_1 : A^1, \ldots, x_n : A^n \vdash x_i : A_i$ is a purely positive proof.

We can then define the notion of bouncing-cut and their origin:

Definition 35 (Bouncing-Cut). A Bouncing-Cut is a cut of the form:

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

Property 36 (Origin of Bouncing-Cut). Given a well-typed iso, every occurrence of a rule $be(f)$ in $\|\text{circ}(\omega)\|$ is a premise of a bouncing-cut.

In particular, when following a thread going up into a bouncing-cut, it will always start from the left-hand-side of the sequent, before going back down on the right-hand-side of the sequent. It will also always bounce back up on the bouncing-cut to reach the back-edge.

Theorem 37 (Validity of proofs). If $\vdash \omega : A \leftrightarrow B$ and $\pi = \|\text{circ}(\omega, \emptyset, \alpha, \beta)\|$ then $\pi$ satisfies $\muMALL$ validity criterion from [9].

Proof Sketch. In order to show the validity of our derivation we need, for each infinite branch, to build a valid thread. From the previous lemmas and the syntactical constraints of the language, we have:

\[
\begin{align*}
\text{Figure 5 Positive Phase}
\end{align*}
\]
We showed how ensuring non-overlapping and exhaustivity is enough to ensure the reversibility we will build a pre-thread that follows the focus of the primitive recursive criterion. We know that (ii) the thread has for minimal recurring formula that is visited infinitely often a non-termination. µ is terminating. The logic generalized quantum loops: in [23], lists are the only recursive type which is captured and recursion language with linear combinations of terms. We plan to study purely quantum recursive types and a truly general reversible language. This is a focus of our forthcoming research. Pre-proof validity. Along with this, allowing for coinductive statements and terms would allow for encoding of more functions and to see how a more relaxed criterion would be captured in terms of validity. This leads to the following corollary:

\begin{proof}

\begin{itemize}
  \item \textbf{Theorem 38 (Simulation).} Provided an iso $\omega : A \leftrightarrow B$ and values $v : A$ and $v' : B$, let $\pi = \text{Pos}(\omega v)$ and $\pi' = \text{Pos}(v)$, if $\omega v \rightarrow^* v'$ then $\pi \rightarrow^* \pi'$.
  
  \textbf{Proof sketch.} The proof relies on the definition of a novel explicit substitution rewriting system for the language, called $\rightarrow_{\text{e}$.$\beta}$. Explicit substitution are represented as a series of let constructs where the base case of the rewriting system is $\text{let } x = v \text{ in } x \rightarrow_{\text{e}$.$\beta} v$. Each rewriting step of this system represents exactly one step of the cut-elimination procedure of $\mu\text{MALL}$. Then we only need to show that if $\sigma = \{ \overrightarrow{v} \rightarrow \overrightarrow{v'} \}$ then $\text{let } \overleftarrow{x} \rightarrow_{\text{e}$.$\beta} \sigma(\overrightarrow{e})$.
  
  This leads to the following corollary:
  
  \begin{itemize}
    \item \textbf{Corollary 39 (Isomorphism of proofs.)} If $\vdash \omega : A \leftrightarrow B$ then, given $F_1 = A_{\alpha_1}, F_2 = A_{\alpha_2}, G_1 = B_{\beta_1}, G_2 = B_{\beta_2}$ and the corresponding proofs $\pi : F_1 \vdash G_1$ and the proof $\pi^\perp : G_2 \vdash F_2$ of $\omega^\perp$ are isomorphic:
      \[
      \begin{array}{cccccc}
        & F_1 \vdash G_1 & \pi \downarrow & F_1 \vdash G_1 & \pi \downarrow \quad & F_1 \vdash G_1 \\
        A_\alpha \vdash A_\alpha & \text{id} & \text{cut} & \text{cut} & \text{id}
      \end{array}
      \]
  \end{itemize}
\end{proof}

5 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 $\mu\text{MALL}$ proofs. The latter result rests on the fact that our isos are \textit{structurally recursive}.

Future works. A first extension to our work would be to relax this condition in order for the encoding of more functions and to see how a more relaxed criterion would be captured in terms of pre-proof validity. Along with this, allowing for coinductive statements and terms would allow for a truly general reversible language. 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 [23], lists are the only recursive type which is captured and recursion is terminating. The logic $\mu\text{MALL}$ would help in providing a finer understanding of termination and non-termination.
References


9 Baelde, David and Doumane, Amina and Kuperberg, Denis and Saurin, Alexis.: Bouncing threads for circular and non-wellfounded proofs. To appear in LICS (2022)


12 Curry, H.B.: Functionality in combinatory logic. Proceedings of the National Academy of Sciences of the United States of America 20(11), 584 (1934)


27 Luca Paolini and Mauro Piccolo and Luca Roversi: A class of Recursive Permutations which is Primitive Recursive complete https://doi.org/10.1016/j.tcs.2019.11.029

\section{Proof of Section 2}

Proof of Lemma 7. w.l.o.g consider \( \omega = \{ v_1 \leftrightarrow e_1 \mid \ldots \mid v_n \leftrightarrow e_n \} \), we look at one clause in particular and its dual:

\[
\begin{array}{l}
\left( \begin{array}{l}
\alpha_1 \leftrightarrow \text{let } p_1 = \omega_1 p'_1 \text{ in } v_1 \\
\vdots \\
\alpha_n \leftrightarrow \text{let } p_n = \omega_n p'_n \text{ in } v_n \\
\end{array} \right) \\
\end{array}
\]

By typing we know that \( \Delta \vdash \alpha_i : A \) and \( \Delta ; \Psi \vdash e \text{ let } p_i = \omega_i p'_i \text{ in } v_i : B \)

By substitution we get that the typing judgment of the expression \( \text{let } p_i = \omega_i p'_i \text{ in } \ldots \) generates the new typing judgement \( \Gamma^{i+1}_i \vdash \ldots \). For all \( i \) we get that \( \Delta, \bigcup\Gamma^{i+1}_j \vdash \omega_i p'_i \text{, finally } v'_i \) is typed by \( \Delta^{n+1}_i \bigcup\Gamma^{n+1}_i \).

When typing the dual clause, we start with the context \( \Delta^{n+1}_i \bigcup\Gamma^{n+1}_i \). We have from hypothesis that:

- Each \( \omega_i p_i \) is typed by \( \bigcup\Gamma^{i+1}_i \), which is possible by our typing hypothesis.
- Each \( p'_i \) generates the context \( \Delta_i, \bigcup\Gamma^{i+1}_j \).

At the end we end up with \( \Delta_1, \ldots, \Delta_n, \Delta^{n+1}_i \vdash \alpha_i, v_i \) which is typable by our hypothesis.

\[\land\]

\begin{lemma}[Substitution Lemma Of Variables]. \textbf{Let}

\[\forall \Delta \vdash \alpha_i : A, \ldots, \Delta, \Delta_n \vdash \alpha_n : A_n\]

\[\forall \Gamma, \alpha : A, \ldots, x_n : A_n \vdash t : B\]

\[\text{Let } \sigma = \{ x_1 \mapsto v_1, \ldots, x_n \mapsto v_n \} \text{ then}\]

\[\Gamma, \Delta_1, \ldots, \Delta_n \vdash \sigma(t) : B\]

\end{lemma}

\textbf{Proof.} By induction on \( t \).

- Case \( x \), then \( \Gamma = \emptyset \) and we have \( \sigma = \{ x \mapsto v \} \) for some \( v \) of type \( B \) under some context \( \Delta \), then we get \( \Delta \vdash \sigma(x) : B \) which leads to \( \Delta \vdash v : B \) which is typable by our hypothesis.
- Case \( () \), nothing to do.
- Case \( \text{in}_j t' \), by substitution we have \( \sigma(\text{in}_j t') = \text{in}_j \sigma(t') \) and by typing we get

\[\Gamma, \Delta_1, \ldots, \Delta \vdash \sigma(t')\]

\[\Gamma, \Delta_1, \ldots, \Delta \vdash \text{in}_j \sigma(t') \text{ which is typable by induction hypothesis on } t'.\]

- Case \( \text{case}_{\text{in}_j t', \text{fold} t', \omega t', \omega t'} \) are similar.
- Case \( (t_1, t_2) \), by typing we get that we can split \( \Gamma \) into \( \Gamma_1, \Gamma_2 \) and the variables \( x_1, \ldots, x_n \) are split into two parts for typing both \( t_1 \) or \( t_2 \) depending on whenever or not a variable occurs freely in \( t_1 \) or \( t_2 \), w.l.o.g say that \( x_1, \ldots, x_i \) are free in \( t_1 \) and \( x_{i+1}, \ldots, x_n \) are free in \( t_2 \) then we get:

\[\Gamma_1, \Delta_1, \ldots, \Delta_1 \vdash t_1 : B_1\]

\[\Gamma_2, \Delta_{i+1}, \ldots, \Delta_n \vdash t_2 : B_2\]

\[\Gamma_1, \Gamma_2, \Delta_1, \ldots, \Delta_n \vdash (t_1, t_2) : B_1 \otimes B_2\]

By substitution we get that \( \sigma((t_1, t_2)) = \{ \sigma(t_1), \sigma(t_2) \} \) so we get the following typing derivation which is completed by induction hypothesis on the subterms:

\[\Gamma_1, \Delta_1, \ldots, \Delta_i \vdash t_1 : B_1\]

\[\Gamma_2, \Delta_{i+1}, \ldots, \Delta_n \vdash t_2 : B_2\]

\[\Gamma_1, \Gamma_2, \Delta_1, \ldots, \Delta_n \vdash (t_1, t_2) : B_1 \otimes B_2\]

- Case \( \text{let } p = t_1 \text{ in } t_2 \) Similar to the case of the tensor.

\[\land\]
Lemma 41 (Substitution Lemma Of Isos). If:

\[ \Delta; f : \alpha \vdash t : A \]
\[ g : \beta \vdash \omega : \alpha \]

Then \( \Delta; g : \beta \vdash t[f := \omega] : A \)

And if:

\[ f : \alpha \vdash \omega_1 : \beta \]
\[ h : \gamma \vdash \omega_2 : \alpha \]

Then \( h : \gamma \vdash \omega_1[f := \omega_2] : \beta \)

Proof. We prove those two propositions by mutual induction on \( t \) and \( \omega_1 \).

**Terms.** by induction on \( t \).

If \( t = x \) or \( t = () \) then there is nothing to do.

If \( t = \text{inj}_{\beta}[t'] \) or \( \text{inj}_p[t_1, t_2] \), then similarly to the proof of Lemma 40 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: \( (\omega[t := \omega]) \) \( (t'[f := \omega]) \) and by induction hypothesis on \( t' \) and by the mutually recursive proof.

**Isos.** by induction on \( \omega_1 \).

If \( \omega_1 = f \), then we get \( h : \gamma \vdash f[f := \omega_2] : \beta \) which is typable by hypothesis.

If \( \omega_1 = g \neq f \) is impossible by our typing hypothesis.

If \( \omega_1 = \text{fix } g.\omega \), then by typing \( f \) does not occur in \( \omega_1 \) so nothing happens.

If \( \omega_1 = \{ e_1 \leftrightarrow v_1 | ... | e_n \leftrightarrow v_n \} \), then, by definition of the substitution we have that

\( \{ e_1 \leftrightarrow v_1 | ... | e_n \leftrightarrow v_n \}[f := \omega_2] = \{ e_1 \leftrightarrow v_1[f := \omega_2] | ... | e_n[f := \omega_2] \} \)

in which case we apply the substitution lemma of isos on terms.

Proof of Lemma 12. By induction on \( t \rightarrow t' \) and direct by Lemma 40 and Lemma 41.

Proof of Lemma 13. Direct by induction on \( \vdash e \, t : A \). The two possible reduction cases, \( \omega \) and \( \mu \) in \( t \) always reduces by typing, pattern-matching and by Lemma 3.

Lemma 42 (Non-Overlapping). Let \( OD_A(S) \) then \( \forall \vdash e \, v : A \), if there exists \( v_1, v_2 \in S \) such that \( \sigma_1[v_1] = v \) and \( \sigma_2[v_2] = v \) then \( v_1 = v_2 \)

Proof. By induction on \( OD_A(S) \).

Case \( \{ () \} \) or \( \{ x \} \) is direct.

Case \( OD_{A \beta B} \{ \text{fix } j[v] \mid v \in S_1 \} \cup \{ \text{inj}_j[v] \mid v \in S_2 \} \), with \( OD_A(S_1) \) and \( OD_B(S_2) \)

By pattern-matching, both \( v_1 \) and \( v_2 \) are either in \( S_1 \) or \( S_2 \).

Take \( v = \text{inj}_j[v'] \), then for all \( \text{fix } j \, v_1, \text{inj}_j[v_2] \in S_1 \), by pattern matching we know that

\( \sigma_1[v_1] = v' \) and \( \sigma_2[v_2] = v' \). By Induction hypothesis \( v_1 = v_2 \) hence \( \text{inj}_j v_1 = \text{inj}_j v_2 \).

The same goes for \( S_2 \).

The case is similar for \( OD_{\mu X,A}(S) \).
Case $OD_{A \otimes B}(S)$: Assuming we are in the first case of the disjunction in the premise of $OD_{A \otimes B}$, the other case being similar:

Take $(v_1, v_2), (v'_1, v'_2) \in S$ and $(\overline{\sigma}, \overline{\nu})$ such that $\sigma_1[\langle v_1, v_2 \rangle] = \langle \overline{\sigma}, \overline{\nu} \rangle$ and $\sigma_2[\langle v'_1, v'_2 \rangle] = \langle \overline{\sigma}, \overline{\nu} \rangle$.

By definition of the pattern-matching we have: $\sigma_1[v_1] = \overline{\sigma}$ and $\sigma_2[v'_1] = \overline{\sigma}$. By induction hypothesis on $X$ we get that $v_1 = v'_1$. We also get that $v_2, v'_2 \in S_{v_1}^1 = S_{v_2}^1$ and so by induction hypothesis we get that $v_2 = v'_2$ and hence $\langle v_1, v_2 \rangle = \langle v'_1, v'_2 \rangle$.

Lemma 43 (Exhaustivity). Let $OD_A(S)$ and $\vdash v : A$ then there exists $v_1 \in S$ st $\sigma[v_1] = v$

Proof. By induction on $OD_A(S)$

- $OD(\{x\})$ then the pattern-matching matches for any $v$.
- $OD(\{\}())$ then the only possible value for $v$ is () and $\sigma[()] = ()$
- $OD_{A \otimes B}(\{inj_1, v \mid v \in S_A \cup \{inj_1, v \mid v \in S_B\})$ let $v = inj_1, \overline{\nu}$ then by induction hypothesis on $S_A$ there exists $v \in S_A$ st $\sigma[v] = \overline{\nu}$, hence $\sigma[inj_1 v] = inj_1, \overline{\nu}$
- Other cases are similar.

Assuming we are in the first case of the disjunction in the premise of $OD_{A \otimes B}$, the other case being similar:

Take some $\langle v, v' \rangle : A \otimes B$, by induction hypothesis we know that there exists some $v_i \in \{v_1, \ldots, v_n\}$ such that $\sigma[v_i] = v$ and therefore that there exists some $v'_i \in S_{v_i}^1$ such that $\sigma'[v'_i] = v'$, therefore we get $(\sigma \cup \sigma')[\langle v_1, v'_1 \rangle] = \langle v, v' \rangle$.

Proof of Lemma 3. Direct implication of Lemma 42 and Lemma 43.

Lemma 44 (Commutativity of substitution). Let $\sigma_1, \sigma_2$ and $v$, such that $\sigma_1 \cup \sigma_2$ closes $v$ and $\text{supp}(\sigma_1) \cap \text{supp}(\sigma_2) = \emptyset$

Then $\sigma_1(\sigma_2(v)) = \sigma_2(\sigma_1(v))$

Proof. Direct as $\sigma_1$ and $\sigma_2$ have disjoint support.

Proof of Theorem 8. By induction hypothesis on the size of $\omega$:

- Case where $\omega = \{v_1 \leftrightarrow v_1 \mid \ldots \mid v_n \leftrightarrow v'_n\}$ then $\omega^{-1}(\omega v_0)$, by non-overlapping and exhaustivity there exists a $v_i$ such that $\sigma[v_i] = v_0$ and hence the terms reduces to $\omega^{-1} \sigma(v'_i)$. It is clear that $\sigma'[v'_i] = \sigma(v_i)$ and hence the terms reduces to $\sigma(v_i)$, but by the first pattern-matching we know that $\sigma(v_i) = v_0$, which concludes the case.

- Case where $\omega = \{v_1 \leftrightarrow e_1 \mid \ldots \mid v_n \leftrightarrow e_n\}$, for simplicity of writing we write a single clause:

$$
\begin{pmatrix}
  v_1 \leftrightarrow p_1 = \omega_1 p'_1 \text{ in } \\
  \ldots \\
  \ldots \\
  \ldots \\
  \text{let } p_n = \omega_n p'_n \text{ in } v'_1
\end{pmatrix}
\begin{pmatrix}
  v'_1 \leftrightarrow p'_n = \omega_n^{-1} p_n \text{ in } \\
  \ldots \\
  \ldots \\
  \ldots \\
  \text{let } p'_1 = \omega_1^{-1} p_1 \text{ in } v_1
\end{pmatrix}

$$

Take some closed value $\vdash v_0 : A$ such that $\sigma[v_1] = v_0$.

By linearity, we can decompose $\sigma$ into $\sigma_1, \ldots, \sigma_n, \sigma_{n+1}$ such that, after substitution we obtain
let \( p_1 = \omega_1 \sigma_1(p'_1) \) in \\
\( \ldots \)

let \( p_n = \omega_n \sigma_n(p'_n) \) in \\
\( \sigma_{n+1}(v'_1) \)

By Lemma 13, each let construction will reduce, and by the rewriting strategy we get:

\[
\begin{align*}
\text{let } p_1 &= \omega_1 \sigma_1(p'_1) \text{ in} \\
\text{let } p_2 &= \omega_2 \gamma_1^1(\sigma_1(p'_1)) \text{ in} \\
\text{let } p_3 &= \omega_3 \gamma_1^1(\sigma_1(p'_1)) \text{ in} \\
\text{let } p_4 &= \omega_4 \gamma_1^1(\sigma_1(p'_1)) \text{ in} \\
\text{let } p_5 &= \omega_5 \gamma_1^1(\sigma_1(p'_1)) \text{ in} \\
\text{let } p_6 &= \omega_6 \gamma_1^1(\sigma_1(p'_1)) \text{ in} \\
\ldots
\end{align*}
\]

\[ let \ p_n = \omega_n \sigma_n(p'_n) \text{ in } \sigma_{n+1}(v'_1) \]

The final term reduces to \( \gamma_n^n(\ldots(\gamma_1^n(\sigma_{n+1}(v'_1)))\ldots) \)
and creates a new substitution \( \gamma, \) the term will hence reduce to \( \gamma_n(\ldots(\gamma_1(\sigma_{n+1}(v'_1)))\ldots). \) Let 
\( \delta = \bigcup_i \gamma_i \cup \sigma_{n+1} \)

We now want to evaluates

\[
\left( \begin{array}{c}
   v'_1 \\
   \vdots \\
   v'_1
\end{array} \right) \leftrightarrow \\
\text{let } p_n = \omega^{-1}_n p_n \text{ in } v_1
\]

We get \( \delta[v'_1] = \delta(v'_1) \).

We know that each \( \gamma \) closes only \( p_i \), we can therefore substitute the term as:

\[
\begin{align*}
\text{let } p'_n &= \omega_1 \gamma_n(p_n) \text{ in} \\
\text{let } p'_1 &= \omega_1 \gamma_1(p_1) \text{ in} \\
\text{let } p'_1 &= \omega_1 \gamma_1(p_1) \text{ in} \\
\text{let } p'_1 &= \omega_1 \gamma_1(p_1) \text{ in} \\
\text{let } p'_1 &= \omega_1 \gamma_1(p_1) \text{ in} \\
\text{let } p'_1 &= \omega_1 \gamma_1(p_1) \text{ in} \\
\text{let } p'_1 &= \omega_1 \gamma_1(p_1) \text{ in} \\
\text{let } p'_1 &= \omega_1 \gamma_1(p_1) \text{ in} \\
\text{let } p'_1 &= \omega_1 \gamma_1(p_1) \text{ in} \\
\text{let } p'_1 &= \omega_1 \gamma_1(p_1) \text{ in} \\
\text{let } p'_1 &= \omega_1 \gamma_1(p_1) \text{ in} \\
\text{let } p'_1 &= \omega_1 \gamma_1(p_1) \text{ in} \\
\text{let } p'_1 &= \omega_1 \gamma_1(p_1) \text{ in} \\
\text{let } p'_1 &= \omega_1 \gamma_1(p_1) \text{ in} \\
\text{let } p'_1 &= \omega_1 \gamma_1(p_1) \text{ in} \\
\text{let } p'_1 &= \omega_1 \gamma_1(p_1) \text{ in} \\
\text{let } p'_1 &= \omega_1 \gamma_1(p_1) \text{ in} \\
\text{let } p'_1 &= \omega_1 \gamma_1(p_1) \text{ in} \\
\text{let } p'_1 &= \omega_1 \gamma_1(p_1) \text{ in} \\
\text{let } p'_1 &= \omega_1 \gamma_1(p_1) \text{ in} \\
\text{let } p'_1 &= \omega_1 \gamma_1(p_1) \text{ in} \\
\text{let } p'_1 &= \omega_1 \gamma_1(p_1) \text{ in} \\
\text{let } p'_1 &= \omega_1 \gamma_1(p_1) \text{ in} \\
\text{let } p'_1 &= \omega_1 \gamma_1(p_1) \text{ in} \\
\text{let } p'_1 &= \omega_1 \gamma_1(p_1) \text{ in} \\
\text{let } p'_1 &= \omega_1 \gamma_1(p_1) \text{ in} \\
\end{align*}
\]

By induction hypothesis, Each let clause will re-create the substitution \( \sigma_i \), we know this as the fact that the initial let construction, let \( p_1 = \omega_1 \gamma_i(p_i) \) in \( \ldots \)

reduces to let \( p_i = v_i \) in \( \ldots \)

While the new one, let \( p'_i = \omega^+_i \gamma_i(p_i) \) in \( \ldots \), is, by definition of the substitution the same as

let \( p'_i = \omega^+_i v_i \) in \( \ldots \)

Then, since we know that \( v_i \) is the result of \( \omega \gamma_i(p'_i) \), we get by induction hypothesis \( \sigma(p'_i) \) as the result of the evaluation.

Therefore, after rewriting we obtain:

\( \sigma_n(\ldots(\sigma_1(\sigma_{n+1}(v'_1)))\ldots) \), by commutativity its the same as \( \sigma_1(\ldots(\sigma_n(\sigma_{n+1}(v'_1)))\ldots) \) which is equal to \( v \).

\[ \blacktriangleright \]

\textbf{Theorem 45 (Termination).} Every well-typed terms \( s \) terminates.

\textbf{Proof.} Given \( \omega' = \text{fix } f.\omega \), the only possible source of non termination is that if when evaluating

\( \omega'(v_1, \ldots, v_n) \) we evaluate infinitely often a term of the form \( s(v'_1, \ldots, v'_n) \), but since we are structurally recursive, we know one of the \( v'_i \) is small than \( v_i \) and we hence have a decreasing argument at each recursive call.

\[ \blacktriangleright \]
B Proof of Section 3

Proof of Theorem 23. By direction induction on \( f \).

Proof of Theorem 24. By induction on \( f \).

- Direction for the identity, swap and sign-change.
- For the Successor:
  \[
  \omega = \begin{cases}
  \inj_i() & \leftrightarrow \inj_r \inj_l \text{fold} \inj_i() \\
  \inj_i, \inj_l x & \leftrightarrow \inj_r \inj_l \text{fold} \inj_r x \\
  \inj_r \inj_l \text{fold} \inj_i() & \leftrightarrow \inj_j() \\
  \inj_r, \inj_r \text{fold} \inj_r x & \leftrightarrow \inj_r \inj_r x
  \end{cases}
  \]
  we do it by case analysis on the sole input \( n \).
  - \( n = 0 \) then \( \bar{0} = \inj_i() \) and \( \omega \inj_i() \rightarrow \inj_r (\inj_l (\text{fold} (\inj_i())))) = \bar{1} \)
  - \( n = -1 \) then \( \bar{-1} = \inj_i (\inj_r (\text{fold} (\inj_i()))), \) so the term reduces to \( \inj_i() = \bar{0} \)
  - Case \( n < -1 \), we have \( \bar{n} = \inj_r (\inj_r (\text{fold} (\inj_r n'))) \) with \( n' = n + 1 \), by pattern matching we get \( \inj_r \inj_r x \) which is \( n' \)
  - Case \( n > 1 \) is similar.
- The Predecessor is the dual of the Successor.
- **Composition & Parallel composition**: Direct by induction hypothesis on \( \omega_f \) and \( \omega_g \).
- **Finite Iteration: \( \text{It}[f] \)**.
  We need the following lemma: \( \omega_{\text{aux}}(\vec{x}_1, \ldots, \vec{x}_n, z) \rightarrow^* (\vec{z}_1, \ldots, \vec{z}_n, \bar{z}) \) where \( z \) is a non-zero integer and \( (\vec{z}_1, \ldots, \vec{z}_n) = f|z|(\vec{x}_1, \ldots, \vec{x}_n) \) which can be shown by induction on \( |z| \): the case \( z = 1 \) and \( \bar{z} = \text{fold} \inj_j() \) is direct by induction hypothesis on \( \omega_f \). Then if \( z = n + 1 \) we get it directly by induction hypothesis on both \( \omega_f \) and our lemma.
  Then, for isos(\( \text{It}[f] \)) 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 stricly positive or stricly negative) then we enter \( \omega_{\text{aux}} \), and apply the previous lemma.
- **Conditional: \( \text{If}[f, g, h] \)**. Direct by case analysis of the last value and by induction hypothesis on \( \omega_f, \omega_g, \omega_h \).

C Proofs of Section 4

C.1 Simulation of the cut-elimination procedure

To make the relation with the logic \( \mu \text{MALL} \) and its cut-elimination procedure simpler, we consider a new rewriting system based on explicit substitution, represented with \textit{let}.

\[
\begin{align*}
\text{let } x &= v \text{ in } x \rightarrow_{e\beta} v \\
\text{let } (x_1, p) &= (t_1, t_2) \text{ in } t \rightarrow_{e\beta} \text{let } x_1 = t_1 \text{ in } \text{let } p = t_2 \text{ in } t \\
\text{let } x &= v \text{ in } (t_1, t_2) \rightarrow_{e\beta} (\text{let } x = v \text{ in } t_1, t_2) & \text{when } x \in \text{FV}(t_1) \\
\text{let } x &= v \text{ in } (t_1, t_2) \rightarrow_{e\beta} (t_1, \text{let } x = v \text{ in } t_2) & \text{when } x \in \text{FV}(t_2) \\
\text{let } x &= v \text{ in } \inj_j t \rightarrow_{e\beta} \inj_j \text{ let } x = v \text{ in } t \\
\text{let } x &= v \text{ in } \inj_r t \rightarrow_{e\beta} \inj_r \text{ let } x = v \text{ in } t \\
\text{let } x &= v \text{ in } \text{fold } t \rightarrow_{e\beta} \text{fold let } x = v \text{ in } t \\
\text{let } x &= v \text{ in } \omega t \rightarrow_{e\beta} \omega \text{ let } x = v \text{ in } t
\end{align*}
\]
Lemma 46 (Specialisation of the substitution on pairs). Let \( \sigma \) be a substitution that closes \( \Delta \vdash \langle t_1, t_2 \rangle \), then there exists \( \sigma_1, \sigma_2 \), 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 \( \sigma \) into \( \sigma_1, \sigma_2 \) defined as \( \sigma_i = \{ (x_i \mapsto v_i) \mid x_i \in FV(t_i) \} \) for \( i \in \{1, 2\} \).

Lemma 47 (Explicit substitution and substitution coincide). Let \( \sigma = \{ x_i \mapsto v_i \} \) be a substitution that closes \( t \), then let \( x_1 = v_1 \) in \( \ldots \), let \( x_n = v_n \) in \( t \rightarrow e_\beta \sigma(t) \).

Proof. By induction on \( t \).

- \( x \), then \( \sigma(x) = v \) and let \( x = v \) in \( t \rightarrow e_\beta v = \sigma(x) \).
- \( \lambda \) then \( \sigma \) is empty and no substitution apply.
- \( \langle t_1, t_2 \rangle \), then by Lemma 46 \( \sigma(\langle t_1, t_2 \rangle) = \langle \sigma_1(t_1), \sigma_2(t_2) \rangle \). By \( \rightarrow e_\beta \), each let construction will enter either \( t_1 \) or \( t_2 \), then by induction hypothesis.
- let \( p = t_1 \) in \( t_2 \) is similar to the product case.
- inj \( j \), inj \( j \), fold \( \omega \). All cases are treated the same: by definition of \( \rightarrow e_\beta \), each let will enter into the subterm \( t \), as with the substitution \( \sigma \), then by induction hypothesis.

Theorem 48 (Simulation). Let \( \Theta \vdash t \) be a well-typed closed term:

If \( t \rightarrow e_\beta t' \) then \( \| \text{Pos}(t) \| \sim \| \text{Pos}(t') \| \).

Proof. By induction on \( \rightarrow e_\beta \).

- let \( x = v \) in \( t \rightarrow e_\beta v \).

\[
\frac{\| \text{Pos}(v) \|}{\Theta \vdash G} \quad \frac{G \vdash F}{G \vdash F} \quad \frac{\text{id}}{\Theta \vdash G} \quad \frac{\| \text{Pos}(v) \|}{\Theta \vdash G} \\
\frac{\Theta_1 \vdash G}{\Theta_1 \vdash G} \quad \frac{\Theta_2 \vdash F}{\Theta_2 \vdash F} \quad \frac{\Theta_3, G, F \vdash H}{
\Theta_3, G, F \vdash H}
\]

\( \Theta_1, \Theta_2, \Theta_3 \vdash H \)

- let \( (x_1, p) = \langle t_1, t_2 \rangle \) in \( t \rightarrow e_\beta \) let \( x_1 = t_1 \) in let \( p = t_2 \) in \( t \).

\[
\frac{\| \text{Pos}(t_1) \|}{\Theta_1 \vdash G} \quad \frac{\| \text{Pos}(t_2) \|}{\Theta_2 \vdash F} \quad \frac{\| \text{Neg}([p], t) \|}{\Theta_3, G, F \vdash H} \quad \frac{\| \text{Pos}(t_1) \|}{\Theta_1 \vdash G} \\
\frac{\Theta_1 \vdash G}{\Theta_1 \vdash G} \quad \frac{\Theta_2 \vdash F}{\Theta_2 \vdash F} \quad \frac{\Theta_3 \vdash G \vdash F}{\Theta_3 \vdash G \vdash F}
\]

\( \Theta_1, \Theta_2, \Theta_3 \vdash H \)

Then:

\( \Theta_1, \Theta_2, \Theta_3 \vdash H \)

- let \( x = v \) in \( \text{inj}_j \) \( t \rightarrow e_\beta \) \( \text{inj}_j \) let \( x = v \) in \( t \).

\[
\frac{\| \text{Pos}(v) \|}{\Theta_1 \vdash G} \quad \frac{\| \text{Pos}(t) \|}{\Theta_2 \vdash F} \quad \frac{\| \text{Pos}(v) \|}{\Theta_3 \vdash G \vdash F}
\]

\( \Theta_1, \Theta_2, \Theta_3 \vdash H \)

Then:

\( \Theta_1, \Theta_2, \Theta_3 \vdash H \)

- The same goes for let \( x = v \) in \( \text{inj}_r \) \( t \rightarrow e_\beta \) \( \text{inj}_r \) let \( x = v \) in \( t \) and let \( x = v \) in \( \text{fold} \) \( t \rightarrow e_\beta \) fold let \( x = v \) in \( t \)

- let \( x = v \) in \( \langle t_1, t_2 \rangle \rightarrow e_\beta \langle \text{let} x = v \in t_1, t_2 \rangle \) when \( x \in FV(t_1) \)

Then:

\[
\frac{\| \text{Pos}(t_1) \|}{\Theta_1 \vdash F} \quad \frac{\| \text{Pos}(t_2) \|}{\Theta_3 \vdash G}
\]

\( \Theta_1, \Theta_2, \Theta_3 \vdash H \)

\( \Theta_1, \Theta_2, \Theta_3 \vdash H \)

\( \Theta_1, \Theta_2, \Theta_3 \vdash H \)

\( \Theta_1, \Theta_2, \Theta_3 \vdash H \)

\( \Theta_1, \Theta_2, \Theta_3 \vdash H \)
A Curry-Howard Correspondence for Linear, Reversible Computation

Lemma 49. Let Γ ⊢ v : A such and Δ ⊢ v' : A st σ[v_j] = v' and σ = {x_j \mapsto \overline{w_j}} then for any list l = [\overline{v_1}, \ldots, \overline{v_n}] where (Γ_i ⊢ v_i)_{i \in I} and such that \text{OD}_A(l(v_1, \ldots, v_n)) and any set of derivations (Γ_i ⊢ v_i : B)_{i \in I} such that \text{OD}_B(V_1(e_i)) and given Θ = \{ x : A_{\text{st}} | x : A \in Δ \} and G = B_1, we have:

\[ \Theta \vdash H \quad \Theta \vdash G \]
\[ \pi = \text{Neg}((l, \text{let } x = \overline{w_i} \text{ in } e)) \]

Proof. By induction on \text{OD}_A(\cdot)

Case \text{OD}_A(\{x\}) we get σ[x] = v then \pi = \pi'.

Case PE_1(\{|\}) then σ(\{\}) = ()

\[ \pi = \text{Neg}((l, \text{let } x = \overline{w_i} \text{ in } e)) \]

Which reduces to

Then \pi =  \Theta \vdash G

Then by induction hypothesis.

Case OD_{A\oplus B}(\{\text{inj}_j, v_i \} \cup \{\text{inj}_j, v_k \}) with σ[\text{inj}_j v_i] = \text{inj}_j v'

\[ \Theta \vdash H \quad \Theta \vdash G \]

Which reduces to

Then \pi =  \Theta \vdash G

Then by induction hypothesis.
K. Chardonnet and A. Saurin and B. Valiron XX:25

Case $\sigma[\text{inj}_r, v] = \text{inj}_r, v'$. Similar to the previous case.

Case $\text{OD}_{A \otimes B}^{\top} \{(v_1^1, v_2^2)\}$ with $\sigma[(v_1^1, v_2^2)] = (v_1', v_2')$

\[\begin{array}{c}
\text{Pos}(v_1') \\
\text{Neg}(\{(v_1, v_2, l, e)\}) \\
\hline
\Theta_1 \vdash H \\
\Theta_2 \vdash B \\
\hline
\Theta_1, \Theta_2 \vdash H \otimes F \\
H, F \vdash G \\
\hline
\Theta_1, \Theta_2 \vdash H \otimes G \\
\hline
\text{cut}
\end{array}\]

Then $\pi = \Theta_1, \Theta_2 \vdash G$

Which reduces to

\[\begin{array}{c}
\text{Pos}(v_1') \\
\text{Pos}(v_2') \\
\text{Neg}(\{(v_1, v_2, l, e)\}) \\
\hline
\Theta_1 \vdash H \\
\Theta_2 \vdash F \\
\hline
\Theta_2, H \vdash G \\
\hline
\Theta_1, \Theta_2 \vdash H \otimes G \\
\text{cut}
\end{array}\]

Because the negative phase on $[v_1, v_2]$ only produces & $\exists, \top, \nu$ rules, we get that $\text{Neg}(\{(v_1 :: v_2 :: l, e)\}) = \text{Neg}(\{(v_2 :: v_1 :: l, e)\})$ by the commutation of rules of Linear Logic. Therefore we can get

\[\begin{array}{c}
\text{Pos}(v_1') \\
\text{Pos}(v_2') \\
\text{Neg}(\{(v_1, v_2, l, e)\}) \\
\hline
\Theta_1 \vdash H \\
\Theta_2 \vdash F \\
\hline
\Theta_2, H \vdash G \\
\hline
\Theta_1, \Theta_2 \vdash H \otimes G \\
\text{cut}
\end{array}\]

which by induction hypothesis on $v_2$ reduces to

\[\begin{array}{c}
\text{Pos}(v_1') \\
\text{Neg}(\{(v_1, v_2, l, e)\}) \\
\hline
\Theta_1 \vdash H \\
\hline
\Theta_1, \Theta_2 \vdash H \otimes G \\
\text{cut}
\end{array}\]

And then we can imply our second induction hypothesis on $v_1$.

\[\Box\]

**Theorem 50** (Iso-substitution cut-elim). Let $\{v_1 \leftrightarrow e_1 \mid \ldots \mid v_n \leftrightarrow e_n\} v \rightarrow \sigma(e_i)$ when $\sigma[v_i] = v$ then $\{\text{Pos}(\{v_1 \leftrightarrow e_1 \mid \ldots \mid v_n \leftrightarrow e_n\} v)\} \rightarrow^* \{\text{Pos}(\text{let } x_j = v_j \text{ in } e_i)\} \rightarrow^*

\{\text{Pos}(\sigma(e_i))\}$ when $\sigma = \{x_j \mapsto v_j\}$


\[\Box\]

### C.2 Proof validity

**Proof of Lemma 34.** By induction on $\Delta \vdash v : A$

- $x : A \vdash x : A$ then the derivation is $\|; F \vdash F \|^{id}$, which is a purely primitive proofs
- $\vdash () : \emptyset$ then the derivation is $\|; \emptyset \vdash \emptyset \|^{id}$, which is a purely primitive proofs
- $\Delta_1, \Delta_2 \vdash (v_1, v_2) : A \otimes B$ then we get $\Delta_1, \Delta_2 \vdash A \otimes B \otimes$ and then by induction hypothesis on $\pi_1$ and $\pi_2$
- $\Delta \vdash \text{inj}_r, v : A \oplus B$ then the derivation is $\Delta \vdash A \oplus B \oplus^1$ then by induction hypothesis on $\pi$
- Similar for $\text{inj}_r, v$ and fold $v$.

\[\Box\]
We define a notion of term occurrence and show that it matches the addresses obtained from the negative phases:

**Definition 51 (Term Occurrence).** We note by \( \text{Occ}(v) \) the set of Occurrence in the value \( v \) defined inductively on \( v \) by:

- \( \text{Occ}(v) = \{ e \} \) if \( v = x \) or \( v = () \)
- \( \text{Occ}(v_1, v_2) = \{ e \} \cup \{ l \cdot \text{Occ}(v_1) \cup r \cdot \text{Occ}(v_2) \} \)
- \( \text{Occ}(\text{inj}_v) = \{ e \} \cup \{ i \cdot \text{Occ}(v) \} \)
- \( \text{Occ}(\text{fold} \ v) = \{ e \} \cup \{ i \cdot \text{Occ}(v) \} \)

Where \( x : S = \{ x_\alpha \mid \alpha \in S \} \) for \( x \in \{ l, r, i \} \)

Given \( \alpha \in \text{Occ}(v) \) we write \( v @ \alpha \) for the subterm of \( v \) at position \( \alpha \)

We write \( \xi(v) = \{ \alpha \in \text{Occ}(v) \mid v @ \alpha = x \} \) for the set of position of variables in \( v \) and \( \xi(x, v) \) for the position of \( x \) in \( v \).

**Theorem 52.** Given a sequence of sequents \( S_0, \ldots, S_n \), with \( S_0 = A_\alpha \vdash B_\beta \) and \( S_n = \Sigma \vdash B_\beta \)
and the only rules applied are \( \top, \& \), \( \& \), \( \nu \).

There exists a unique variable \( v \) and context \( \Delta \) such that \( \Delta \vdash_e v : A \) and such that for all expression \( e \)
such that \( \Delta \vdash_e e : B \), for all iso \( \omega : A \leftrightarrow B \) such that \( v \leftrightarrow e \) is a clause of \( \omega \), consider \( \pi = \{ \text{circ}(\omega) \} \)
then \( S_0, \ldots, S_n \) is a branch of \( \pi \) and for all formula \( A_\alpha \in \Sigma \), there exists a unique variable \( x \) such
that \( \xi(x, v) \) is a suffix of \( \alpha \).

**Proof.** By induction on \( n \).

- Case 0, then take \( \Delta = x : A \) and \( v = x \), obviously \( \Delta \vdash_e x : A \). We also get that \( \omega = \{ x \leftrightarrow e \} \)

and \( \{ \text{circ}(\omega) \} = A_\alpha \vdash B_\beta \) so the empty sequence is a branch and \( \xi(x, v) = e \) which is a suffix of \( \alpha \).

- Case \( n + 1 \). By induction hypothesis, the sequence \( S_0, \ldots, S_n \) with \( S_n \) sequent of \( \Sigma \vdash B_\beta \)
gives us \( \Delta^n \vdash v^n : A \). Define the values contexts as \( V = [] \mid \langle V, v \rangle \mid \langle v, V \rangle \mid \text{inj}_l \ V \mid \text{fold} \ V \).

Then, by case analysis on the rule of \( S_{n+1} \).

\( \& \): then we can write \( \Sigma^n \) as \( A_{\alpha^1} \ldots, (C_1 \oplus C_2)_{\alpha^2} \ldots, A_{\alpha^n} \vdash B_\beta \) then we know that

\( \Delta^n = x_1 : A_1, \ldots, x_k : C_1 \oplus C_2, \ldots, x_n : A^n \) then \( v \) can be written as \( V[x_k] \).

Build \( v^{n+1} = V[y, z] \) and \( \Delta^{n+1} = \Delta \setminus \{ x_k : C_1 \oplus C_2 \} \cup \{ y : C_1, z : C_2 \} \).

We get that \( \Delta^{n+1} \vdash_e v^{n+1} \), then for any iso \( \omega \) such that \( V[x_k] \leftrightarrow e \) is a clause, we can build
replace the clause by \( V[y, z] \leftrightarrow e[x \leftarrow (y, z)] \) in order to build \( \omega' \), and if \( S_0, \ldots, S_n \) was a branch in \( \{ \text{circ}(\omega) \} \) then so is \( S_0, \ldots, S_n, S_{n+1} \) in \( \omega' \).

We know that \( \xi(x, v) = \gamma \) is a suffix of \( \alpha_k \), then after applying the \( \& \) rule we have that \( C_1 \)
have address \( \alpha l_k \) and \( C_2 \) have address \( \alpha r_k \). Therefore \( \xi(y, v^{n+1}) = \gamma l \) and \( \xi(z, v^{n+1}) = \gamma r \)
which are respectively suffixes of \( \alpha l_k \) and \( \alpha r_k \).

\( \nu \): Assuming that \( S_{n+1} \) goes to the left branch of the \( \& \) rule.

We then have \( \Sigma^n = A_{\alpha^1} \ldots, (C_1 \oplus C_2)_{\alpha^2} \ldots, A_{\alpha^n} \vdash B_\beta \) and \( \Delta^n = x_1 : A_1, \ldots, x_k : C_1 \oplus C_2, \ldots, x_n : A^n \) with \( v = V[x_k] \).

Consider \( v^{n+1} = V[\text{inj}_l y] \) and \( \Delta^{n+1} \).

For any iso \( \omega \) where \( v^n \leftrightarrow e \) was a clause, we can consider the isos \( \omega' \) where the clause
\( v^n \leftrightarrow e \) has been replaced by two clauses \( V[\text{inj}_l y] \) and \( V[\text{inj}_r y] \) with \( e[x \leftarrow y] \) and \( e[x \leftarrow z] \). \( S_0, \ldots, S_n, S_{n+1} \) is obviously a branch in \( \{ \text{circ}(\omega) \} \) by definition if the negative phase.
Also since \( \xi(x, v) = \gamma \) is a suffix of \( \alpha_k \), after applying the \& rule, on the left branch we get \( C_1 \) with address \( \alpha_k l_k \). And \( \xi(y, v^{n+1}) = \gamma I \) is a suffix of \( \alpha_k l_k \).

The other side of the \& is similar and so is the \( \nu \) rule.

\( \top \). In which case we have \( S^n = I_k, A^1_{\alpha_1}, \ldots, A^n_{\alpha_n} \), with \( \Delta^n = x : I_1, \ldots, x_n : A^n \) with \( v^n = V[x] \), build \( v^{n+1} \) as \( V[I] \) and \( \Delta^{n+1} = x_1 : A^1, \ldots, x_n : A^n \). Then after the \( \top \) rule we get \( \Gamma^{n+1} = A^1_{\alpha_1}, \ldots, A^n_{\alpha_n} \) so the property holds by our induction hypothesis.

Given an iso \( \omega = \text{fix } f \{v_1 \leftrightarrow e_1 \mid \ldots \mid v_n \leftrightarrow e_n\} \), we want to show that for any infinite branch there exists a valid thread that inhabit it. As given by Lemma 32, an infinite branch is uniquely defined by a single iso-variable.

Given the value \( v_i^l \) that is the decreasing argument structurally recursive criterion, we want to build a pre-thread that follow the variable \( x_j : \mu X.B \) in \( v_i^l \) that is the focus of the criterion.

**Definition 53 (Pre-Thread of the negative phase).** Given a well typed iso

\( \omega = \text{fix } f \{v_1 \leftrightarrow e_1 \mid \ldots \mid v_n \leftrightarrow e_n\} \) and a clause \( v_1 \leftrightarrow e_1 \) such that \( f p \subseteq e_i \) and the variable \( x^p \) that is the focus of the primitive recursive criterion, and considering \( \pi = \text{circ}(\omega) \) we build the finite pre-thread \( PT_n(x^p) \) that follows the formula \( \mu X.A \) corresponding to \( x^p \).

This is done by induction on \( \text{Neg}((\{v_l, e_i\}_{i \in I})) \).

- Case \( \text{Neg}((\{\}, e)) \) is impossible as we follow a variable.
- Case \( \text{Neg}((\{t : l, e\})) \) then we built \( PT_n(\text{Neg}((\{t, e\})) \)
- Case \( \text{Neg}((\{y : l, e\})) \) = \( \{\}

\begin{align*}
\text{if } y = x^p \\
PT_n(\text{Neg}((\{l, e\})))
\end{align*}

otherwise

- Case \( \text{Neg}(\{\text{inj}_l, v_1 \vdash e_l, v_i \vdash e_i\}_{i \in I}) \cup \{\text{inj}_l, v_j \vdash e_j, v_i \vdash e_i\}_{j \in J}) \)

\begin{align*}
(A \cup C) ; A \cup C, \Delta + B ; (A ; A, \Delta + B) \cdot PT_n(\text{Neg}(\{v_1 \vdash l, e_i \}_{i \in I})))
\end{align*}

\( \text{if } x^p \subseteq v_i \)

\begin{align*}
(C \cup A) ; C \cup A, \Delta + B ; (A ; A, \Delta + B) \cdot PT_n(\text{Neg}(\{v_j \vdash l, e_j \}_{j \in J}))
\end{align*}

\( \text{if } x^p \subseteq v_j \)

\begin{align*}
(\mu X.D ; A_1 \cup A_2, \Delta + B ; (\mu X.D ; A_1, \Delta + B) \cdot PT_n(\text{Neg}(\{v_l \vdash l, e_i \}_{i \in I})))
\end{align*}

\( \text{if } L \in \{I, J\}, k \subseteq \{1, 2\} \) and \( x^p \subseteq l \)

- Case \( \text{Neg}(\{v_1^l, v_i^2 \vdash l, e_i \}_{i \in I}) \)

\begin{align*}
(A_1 \cup A_2, A_1 \cup A_2, \Delta + B ; (A_k ; A_1, A_2, \Delta + B) \cdot PT_n(\text{Neg}(\{v_1^l, v_i^2 \vdash l, e_i \}_{i \in I})))
\end{align*}

\( \text{for } k \in \{1, 2\} \) if \( x^p \subseteq v_k \)

\begin{align*}
(\mu X.D ; A_1 \cup A_2, \Delta + B ; (\mu X.D ; A_1, A_2, \Delta + B) \cdot PT_n(\text{Neg}(\{v_1^l, v_i^2 \vdash l, e_i \}_{i \in I})))
\end{align*}

\( \text{if } x^p \subseteq l \)

- Case \( \text{Neg}(\{\text{fold } v_1 \vdash l, e_i \}_{i \in I}) \)

\begin{align*}
(\mu X.A ; \mu X.A, \Delta + B ; (A[X \leftarrow \mu X.A]; A[X \leftarrow \mu X.A], \Delta + B) \quad \text{if } x^p \subseteq v_i
\end{align*}

\begin{align*}
(\mu X.D ; \mu X.A, \Delta + B ; (\mu X.D ; A[X \leftarrow \mu X.A], \Delta + B) \quad \text{if } x^p \subseteq l_i
\end{align*}

**Lemma 54.** Weight of the Pre-thread for the negative phase \( w(PT(\text{Neg}(\{l, e\}))) \) is a word over \( \{l, r, i, W\} \).

**Proof.** By case analysis of Definition 53:

- If the variable \( x^p \) is not a subterm of the first value from the list \( l \) then the thread has the form \( (A ; C, \Delta + B, \uparrow) \), \( (A ; C', \Delta + B, \uparrow) \) and the weight is \( W \).

- If the variable \( x^p \) is a subterm of the first value of the list \( l \) then by direct case analysis on the first value.
A similar analysis can be done for the positive phase:

\[\text{Definition 55 (Pre-thread for Pos(e)).} \quad \text{Given the formula } x : \nu X. F, \text{ we describe the pre-thread following this formula, } \text{PT}_p \text{ as:} \]

\[= (\nu X.F; s; \uparrow) \cdot (\nu X.F; s'; \uparrow) \quad \text{when } x : \nu X.F \text{ is on the left-hand-side of the sequent.} \]

\[= (\nu X.A_n; (\nu X.A)_1 \vdash (\nu X.A)_2; \uparrow) \cdot ((\nu X.A)_3; (\nu X.A)_4) \quad \text{when going down on a right-rule where } F \text{ is principal and } s \text{ is a premise of } s'. \]

\[= (F; s; \downarrow) \cdot (F'; s'; \uparrow) \quad \text{when going down on a bouncing-cut where } s \text{ is the left-premise of the bouncing cut and } s' \text{ is the conclusion of the back-edge.} \]

When bouncing back on an axiome, } F \text{ necessarily comes from the left-hand-side of the sequent, and therefore when going down on a purely positive proofs, we follow the only formula on the right-hand-side of the sequent.

\[\text{Lemma 56 (Form of Pre-Thread for Pos(e)).} \quad \text{We have } \text{PT}_p(\text{Pos(e)}) = \text{of the form } \mathcal{W}^+, \mathcal{A}\{\mathbb{I}, \tau, \mathcal{W}\}^* C \]

\[\text{Proof.} \quad \text{Direction by case analysis on } \text{Pos(e).} \quad \text{As the thread only goes up by encountering cut-rules or right-rules, we get } \mathcal{W}^+, \text{ and the thread goes up all the way to an axiome rule, corresponding to the formula } x : \nu X.F, \text{ which add the } \mathcal{A}. \quad \text{Finally the thread goes down on the purely positive proof, generating } \mathcal{I}, \tau, \mathcal{W} \text{ until reaching the cut-rule from the bouncing cut.} \]

\[\text{We can then consider the infinite pre-thread as the concatenation of both } \text{PT}_n \text{ and } \text{PT}_p. \]

\[\text{Lemma 57. Form of the Pre-Thread} \]

\[\text{Given the pre-thread } t \text{ following } x : \text{ we have that } t(p) = p_0((\Sigma_{i \leq n} p_i) \mathcal{W}^+ A_i C)^* \]

\[\text{With} \]

\[= p_0 \text{ is any prefix.} \]

\[= p_i \in \{l, r, i, \mathcal{W}\}^* \]

\[= q_i \in \{l, \tau, \mathcal{W}\}^* \]

\[\text{With } \forall i \leq n, \mathcal{F} \subseteq \mathcal{F} \text{ and } \mid p_i \mid \mid q_i \mid \text{ without counting the } \mathcal{W}. \text{ Where } p \subset q \text{ is a prefix of } p. \]

\[\text{With } \mathcal{F} = \mathcal{F} \text{ if } x \in \{l, r, i\}, \mathcal{F} = \mathcal{F} \text{ if } x \in \{l, r, i\}, \text{ and } \mathcal{F} = \mathcal{F} \]

\[\text{Proof.} \quad p_i \text{ is generated from Definition 53 while the rest up to the } C \text{ (included) is generated from Definition 55.} \]

\[\text{First, we show that } \mid p_i \mid \mid q_i \mid \text{ modulo the } \mathcal{W}. \]

\[\text{Since } p_i \text{ is generated by the negative phase, we have that, modulo } \mathcal{W}, \text{ modulo } \mathcal{W}, p_i = \{r\}^* l^+ \{l, r, i\}^*, \text{ this is due by definition of being primitive recursive and because we are looking for the right variable.} \]

\[\text{By definition of being primitive recursive the input type of the iso is } A_1 \otimes \cdots \otimes A_n \text{, hence } \{r\}^* l^+ \text{ is for searching for the good } A_i. \]

\[\text{Then } \{l, r, i\} \text{ is the decomposition of the primitive recursive value, as described in Theorem 52.} \]

\[\text{As } q_i \text{ correspond to the Purely Positive Proof, we know that the Purely Positive Proof is the encoding of a pattern } p = \langle x_1, \ldots, x_n \rangle. \text{ Hence } q_i \text{ can be decomposed as } \{l^+, r^+ \}

\[\text{By the fact that we the iso is primitive recursive we know that the variable in } p \text{ is a strict subterm of the primitive recursive value, hence } \mid p_i \mid \mid q_i \mid \]

\[\text{The fact that } \mathcal{F} \subseteq \mathcal{F} \text{ is direct as the Nice Proof reconstruct the type } A_1 \otimes \cdots \otimes A_n \text{ without modifying the } A_i \text{ while } \mathcal{F} \text{ start by seraching for the corresponding type } A_i, \text{ so its only composed of } \{l, r\}^*, \text{ which will be the same as } \mathcal{F}. \]
Theorem 58. The Pre-thread generated is a thread. We want to find a decomposition of the pre-thread such that it can uniquely decomposed into $\bigodot (H_i \odot V_i)$ with

- $w(V_i) \in \{l, r, i, W\}^\infty$ and non empty if $i \neq \lambda$
- $w(H_i) \in \mathcal{H}$

We recall that, with $x \in \{l, r, i\}$:

$$B ::= C | BW^* AW^* B | \exists W^* BW^* x$$

$$\mathcal{H} ::= \epsilon | AW^* B$$

Proof. We set $H_0$ as the empty pre-thread. (so $w(H_0) = \epsilon$) We set $V_0$ as the maximal possible sequence such that $w(V_0) \in \{l, r, i, W\}^*$, i.e the sequence that end with $(A; A \vdash A; \uparrow)$. Then, for all $i \geq 1$ we set

- $H_i$ start at $(A; A \vdash A; \uparrow)$ just before the axiome rule so that the first element of $w(H_i)$ is $A$. Then $H_i$ is compose of
  - All of the pre-thread going down on the Purely Positive Proof after the axiome, accumulating a word over $\{\bar{I}, \bar{T}, \bar{I}, \bar{W}\}^*$.
  - Going back up into the cut-rule of the bouncing cut, making a $C$
  - Going up to compensate every $\exists$ seen in the Purely Positive Proof while going down. This is possible as shown in Lemma 57
- $V_i$ is the maximal possible sequence such that $w(V_i) \in \{l, r, i, W\}^*$, i.e the sequence that end with $(A; A \vdash A; \uparrow)$.

Theorem 59. Thread Validity The generated thread is valid.

Proof. By Theorem 58 we know that we have a thread.

We also know by Theorem 58 that the visible part is not stationary.

Finally, by Lemma 57 and Theorem 58 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 different 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 $\mu$ formula when going down on a purely positive proof.

By constraints on the syntax of our iso's, all the possible slices are necessarily persistent.

Therefore the smallest formula we will encounter is $\mu X.B$ which is a $\mu$ formula so and the thread is on the left so the thread is valid.