A Curry-Howard Correspondence for Linear, Reversible Computation

Kostia Chardonnet
Université Paris-Saclay, Inria, CNRS, ENS Paris-Saclay, LMF, 91190, Gif-sur-Yvette, France
Équipe Quacs, Inria, Palaiseau, France
Université Paris Cité, CNRS, IRIF, 75013, Paris, France

Alexis Saurin
Université Paris Cité, CNRS, IRIF, 75013, Paris, France
Inria πr^2, Paris, France

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

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 μMALL: 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 μMALL validity criterion and how the language simulates the cut-elimination procedure of μMALL.

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

Digital Object Identifier 10.4230/LIPIcs.CSL.2023.13

Related Version *Full Version*: https://hal.archives-ouvertes.fr/hal-03747425

Funding This work has been partially funded by the French National Research Agency (ANR) under the research projects SoftQPRO ANR17-CE25-0009-02 and RECIPROG ANR-21-CE48-019-01 and within the framework of "Plan France 2030", under the research project ANR-22-PETQ-0007.

1 Introduction

$$\frac{\begin{array}{c} s \\ \vdots \\ A \to B \end{array} \quad \begin{array}{c} t \\ \vdots \\ A \end{array}}{B}$$

Figure 1 Modus Ponens.

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.

31st EACSL Annual Conference on Computer Science Logic (CSL 2023).
Editors: Bartek Klin and Elaine Pimentel; Article No. 13; pp. 13:1–13:18
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

This connection is known as the *Curry-Howard correspondence* [7, 10]. 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 an argument (i.e. t) of type A to a function (i.e. s) of type $A \to B$. 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. This yields a direct proof of B without any invocation of the "lemma" $A \to B$.

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 [5, 14], separation logics with memory-aware type systems [18, 12], resource-sensitive logics with differential privacy [9], logics with monads with reasoning on side-effects [21, 15], etc.

Reversible computation is a paradigm of computation which emerged as an energy-preserving model of computation in which data is never erased [8] that makes sure that, given some process f, there always exists an inverse process f^{-1} such that $f \circ f^{-1} = \mathrm{Id} = f^{-1} \circ f$. Many aspects of reversible computation have been considered, such as the development of reversible Turing Machines [16], reversible programming languages [11] and their semantics [6, 13]. 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 [20]. 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 $\mathbb{1} \oplus \mathbb{1}$, where $\mathbb{1}$ is a type with only one element. The language in [20] 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 [20] 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 [20], 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 [19]. For the Curry-Howard part, we capitalize on the logic μMALL [1, 3]: 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 μMALL, some finitary proof system with explicit induction inferences à la Park [1] as well as non-well-founded proof systems which allow to build infinite derivation [3, 2]. 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 μMALL 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 validity 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 [19], this is shown by encoding the set of Recursive Primitive Permutations [17] 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 μMALL and show that the given derivation respects the validity condition and how our evaluation strategy simulates the cut-elimination procedure of the logic.

2 First-order Isos

Our language is based on the one introduced by Sabry et al [20] 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: while they only allow the inductive type of lists, we consider arbitrary inductive types. The language is defined by layers. Terms and types are presented in Table 1, while typing derivations, inspired from μMALL, 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 $\mathtt{inj}_l$ 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 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 $(x_1, \dots, x_n)$ for $\langle x_1, \langle \dots, x_n \rangle \rangle$ or $\overrightarrow{x}$ when n is non-ambiguous and $A_1 \otimes \cdots \otimes A_n$ for $A_1 \otimes (\cdots \otimes A_n)$ and A^n for $\underbrace{A \otimes \cdots \otimes A}_{n \text{ times}}$.

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 $\{v_1 \leftrightarrow e_1 \mid \dots \mid v_n \leftrightarrow e_n\}$. 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 $\mathtt{OD}_A(\{v_1, \dots, v_n\})$ predicate (corrected from [20], as their definition makes it impossible to type Toffoli) syntactically enforces the exhaustivity and non-overlapping conditions on a set of well-typed values $v_1, \dots, v_n$ 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 $Val(e)$ is defined as $Val(\mathtt{let}\, p = \omega\ p' \ \mathtt{in}\ e) = Val(e)$, and $Val(v) = v$ otherwise. These checks are crucial to make sure that our isos are indeed reversible. In the last rule on Table 4, we define $\pi_1(S)$ and $\pi_2(S)$ as respectively $\{v \mid \langle v, w \rangle \in S\}$ and $\{w \mid \langle v, w \rangle \in S\}$ and S_v^1 and S_v^2 respectively as $\{w \mid \langle v, w \rangle \in S\}$ and $\{w \mid \langle w, v \rangle \in S\}$ Exhaustivity for an iso $\{v_1 \leftrightarrow e_1 \mid \dots \mid v_n \leftrightarrow e_n\}$ 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 $\mathtt{inj}_l\ v$ and $\mathtt{inj}_r\ v'$ are non-overlapping

Table 1 Terms and types.

(Base types)	$A, B ::= \mathbb{1} \mid A \oplus B \mid A \otimes B \mid \mu X.A$
(Isos, first-order)	$\alpha ::= A \leftrightarrow B$
(Values)	$v ::= () \mid x \mid \texttt{inj}_l\, v \mid \texttt{inj}_r\, v \mid \langle v_1, v_2 \rangle \mid \texttt{fold}\, v$
(Pattern)	$p ::= x \mid \langle p_1, p_2 \rangle$
(Expressions)	$e ::= v \mid \texttt{let}\, p_1 = \omega\; p_2\; \texttt{in}\; e$
(Isos)	$\omega ::= \{v_1 \leftrightarrow e_1 \mid \ldots \mid v_n \leftrightarrow e_n\} \mid \textbf{fix}\, f.\omega \mid f$
(Terms)	$t ::= () \mid x \mid \texttt{inj}_l\, t \mid \texttt{inj}_r\, t \mid \langle t_1, t_2 \rangle \mid$ $\texttt{fold}\, t \mid \omega\; t \mid \texttt{let}\, p = t_1\; \texttt{in}\; t_2$

while a variable x is always exhaustive. The construction $\textbf{fix}\; g.\omega$ represents the creation of a recursive function, rewritten as $\omega[g := \textbf{fix}\; g.\omega]$ 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 ω is of type $A \leftrightarrow B$, we can build its inverse $\omega^{\perp} : B \leftrightarrow A$ and show that their composition is the identity. In order to avoid conflicts between variables we will always work up to α-conversion and use Barendregt's convention [4, 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 $\Delta; \Psi \vdash_e t : A$) and one for isos (noted $\Psi \vdash_\omega \omega : A \leftrightarrow B$). In the typing rules, the contexts Δ are sets of pairs that consist of a term-variable and a base type, where each variable can only occur once and Ψ is a singleton set of a pair of an iso-variable and an iso-type association.

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

- *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 $v \leftrightarrow e$*, we call the value* v_i^j *(resp. the variable* x_j*) the* decreasing argument *(resp. the* focus*) of the structurally recursive criterion.*

▶ 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.

Finally, our language is equipped with a rewriting system $\to$ on terms, defined in Definition 4, 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 pattern-matching : the argument is matched against the left-hand-side of each clause until

Table 2 Typing of terms and expressions.

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

$$\frac{\Delta_1; \Psi \vdash_e t_1 : A \quad \Delta_2; \Psi \vdash_e t_2 : B}{\Delta_1, \Delta_2; \Psi \vdash_e \langle t_1, t_2 \rangle : A \otimes B} \quad \frac{\Delta; \Psi \vdash_e t : A[X \leftarrow \mu X.A]}{\Delta; \Psi \vdash_e \mathtt{fold}\, t : \mu X.A}$$

$$\frac{\Psi \vdash_\omega f : A \leftrightarrow B \quad \Delta; \Psi \vdash_e t : A}{\Delta; \Psi \vdash_e f\ t : B} \quad \frac{\vdash_\omega \omega : A \leftrightarrow B \quad \Delta; \Psi \vdash_e t : A}{\Delta; \Psi \vdash_e \omega\ t : B}$$

$$\frac{\Delta_1; \Psi \vdash_e t_1 : A_1 \otimes \cdots \otimes A_n \qquad \Delta_2, x_1 : A_1, \ldots, x_n : A_n; \Psi \vdash_e t_2 : B}{\Delta_1, \Delta_2; \Psi \vdash_e \mathtt{let}\, (x_1, \ldots, x_n) = t_1\ \mathtt{in}\ t_2 : B}$$

Table 3 Typing of isos.

$$\frac{\begin{array}{lll} \Delta_1 \vdash_e v_1 : A & \ldots\ \Delta_n \vdash_e v_n : A & \mathtt{OD}_A(\{v_1, \ldots, v_n\}) \\ \Delta_1; \Psi \vdash_e e_1 : B & \ldots\ \Delta_n; \Psi \vdash_e e_n : B & \mathtt{OD}_B(\{Val(e_1), \ldots, Val(e_n)\}) \end{array}}{\Psi \vdash_\omega \{v_1\ \leftrightarrow\ e_1 \mid \ldots \mid v_n\ \leftrightarrow\ e_n\} : A \leftrightarrow B.}$$

$$\frac{}{f : \alpha \vdash_\omega f : \alpha} \quad \frac{f : \alpha \vdash_\omega \omega : \alpha \qquad \textbf{fix}\ f.\omega \text{ is structurally recursive}}{\Psi \vdash_\omega \textbf{fix}\ f.\omega : \alpha}$$

one of them matches (written $\sigma[v] = v'$), in which case the pattern-matching, as defined in Table 5, returns a substitution σ that sends variables to values. Because we ensure exhaustivity and non-overlapping (Lemma 5), the pattern-matching can always occur on well-typed terms. The *support* of a substitution σ is defined as $\mathrm{supp}(\sigma) = \{x \mid (x \mapsto v) \in \sigma\}$.

▶ **Definition 3** (Substitution). *Applying substitution σ on an expression t, written $\sigma(t)$, is defined as : $\sigma(()) = (), \sigma(x) = v$ if $\{x \mapsto v\} \subseteq \sigma, \sigma(\mathtt{inj}_r\ t) = \mathtt{inj}_r\ \sigma(t), \sigma(\mathtt{inj}_l\ t) = \mathtt{inj}_l\ \sigma(t), \sigma(\langle t, t' \rangle) = \langle \sigma(t), \sigma(t') \rangle, \sigma(\omega\ t) = \omega\ \sigma(t)$ and $\sigma(\mathtt{let}\, p = t_1\ \mathtt{in}\ t_2) = (\mathtt{let}\, p = \sigma(t_1)\ \mathtt{in}\ \sigma(t_2))$.*

▶ **Definition 4** (Evaluation relation $\to$). *We define $\to$ the rewriting system of our language as follows:*

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

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

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

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

▶ **Lemma 5** ($\mathtt{OD}_A(A)$ ensures exhaustivity and non-overlapping.). *Let $\mathtt{OD}_A(S)$ and $\vdash_e v : A$, then there exists a unique $v' \in S$ such that v' matches v under substitution σ, i.e. $\sigma[v'] = v$.*

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 ω and is given in Definition 6.

Table 4 Exhaustivity and Non-Overlapping.

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

$$\frac{\mathtt{OD}_{A[X\leftarrow \mu X.A]}(S)}{\mathtt{OD}_{\mu X.A}(\{\mathtt{fold}\ v \mid v \in S\})} \qquad \frac{\begin{array}{c}\mathtt{OD}_A(\pi_1(S)), \forall v \in \pi_1(S), \mathtt{OD}_B(S_v^1) \\ \text{or } \mathtt{OD}_B(\pi_2(S)), \forall v \in \pi_2(S), \mathtt{OD}_A(S_v^2)\end{array}}{\mathtt{OD}_{A\otimes B}(S = \{\langle v_1, v_1'\rangle, \dots, \langle v_n, v_n'\rangle\})}$$

Table 5 Pattern-matching.

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

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

▶ **Definition 6** (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:*

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

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

▶ **Lemma 7** (Inversion is well-typed). *Given $\Psi \vdash_\omega \omega : A \leftrightarrow B$, then $\Psi \vdash_\omega \omega^\perp : B \leftrightarrow A$.*

▶ **Theorem 8** (Isos are isomorphisms). *For all well-typed isos $\vdash_\omega \omega : A \leftrightarrow B$, and for all well-typed values $\vdash_e v : A$, if $(\omega\ (\omega^\perp\ v)) \to^* v'$ then $v = v'$.*

▶ **Example 9.** We can define the iso of type : $A \oplus (B \oplus C) \leftrightarrow C \oplus (A \oplus B)$ as

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

▶ **Example 10.** We give the encoding of the isomorphism $map(\omega)$ and its inverse: for any given iso $\vdash \omega : A \leftrightarrow B$ in our language, we can define $map(\omega) : [A] \leftrightarrow [B]$ where $[A] = \mu X.\mathbb{1} \oplus (A \otimes X)$ is the type of lists of type A and $[\,]$ is the empty list ($\mathtt{fold}\ (\mathtt{inj}_l\ ())$) and $h :: t$ is the list construction ($\mathtt{fold}\ (\mathtt{inj}_r\ \langle h, t\rangle)$). We also give its dual $map(\omega)^\perp$ below, as given by Definition 6.

$$\begin{array}{l} map(\omega) : [A] \leftrightarrow [B] \\ = \mathbf{fix}\ f. \left\{ \begin{array}{lcl} [\,] & \leftrightarrow & [\,] \\ h :: t & \leftrightarrow & \mathtt{let}\ h' = \omega\ h\ \mathtt{in} \\ & & \mathtt{let}\ t' = f\ t\ \mathtt{in} \\ & & h' :: t' \end{array} \right\} \end{array} \qquad \begin{array}{l} map(\omega)^\perp : [B] \leftrightarrow [A] \\ = \mathbf{fix}\ f. \left\{ \begin{array}{lcl} [\,] & \leftrightarrow & [\,] \\ h' :: t' & \leftrightarrow & \mathtt{let}\ t = f\ t'\ \mathtt{in} \\ & & \mathtt{let}\ h = \omega^\perp h'\ \mathtt{in} \\ & & h :: t \end{array} \right\} \end{array}$$

▶ Remark 11. In our two examples, the left and right-hand side of the $\leftrightarrow$ on each function 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. Both isos are therefore bijections.

The language enjoys the standard properties of typed languages of progress and subject reduction:

▶ **Lemma 12** (Subject Reduction). *If $\Delta; \Psi \vdash_e t : A$ and $t \to t'$ then $\Delta; \Psi \vdash_e t' : A$.*

▶ **Lemma 13** (Progress). *If $\vdash_e t : A$ then, either t is a value, or $t \to t'$.*

3 Computational Content

In this section, we study the computational content of our language. In the case of only finite types made up of the tensor, plus and unit, one can represent any bijection by case analysis. Homewer, with infinite types, the expressivity becomes less clear. We show that we can encode Recursive Primitive Permutations [17] (RPP), which shows us that we can encode at least all Primitive Recursive Functions [19].

We give a few reminders on the language RPP and its main results, then show how to encode it.

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 $\mathbb{Z}^k$, it is built inductively on $k \in \mathbb{N}$ by: the successor (S), the predecessor (P), the identity (ID) and the sign-change that are part of RPP^1. The swap function $(\mathcal{X})$ and the binary permutation $\mathcal{X}$ which sends the pair (x, y) to (y, x) are part of RPP^2 and then, for any function $f, g, h \in \mathrm{RPP}^k$ and $j \in \mathrm{RPP}^l$, we can build (i) the sequential composition $f; g \in \mathrm{RPP}^k$, (ii) the parallel composition $f \mid\mid j \in \mathrm{RPP}^{k+l}$ (iii) the iterator $\mathit{It}[f] \in \mathrm{RPP}^{k+1}$ and (iv) the selection $\mathit{If}[f, g, h] \in \mathrm{RPP}^{k+1}$.

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

$$\mathrm{RPP} = \cup_{k \in \mathbb{N}} \mathrm{RPP}^k$$

We present the semantics of each constructors of RPP under a graphical form, as in [17], 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 [17], 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 $f \in \mathrm{RPP}^k$ we can define an inverse f^{-1}:

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

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

▶ **Proposition 16** (Inversion defines an inverse [17]). *Given $f \in \mathrm{RPP}^k$ then $f; f^{-1} = \mathrm{Id} = f^{-1}; f$*

$$x \;\boxed{S}\; x+1 \qquad x \;\boxed{P}\; x-1 \qquad x \;\boxed{\mathrm{Sign}}\; -x \qquad x \;\boxed{\mathrm{Id}}\; x \qquad \begin{array}{c} x \\ y \end{array} \boxed{\mathcal{X}} \begin{array}{c} y \\ x \end{array}$$

$$\begin{array}{c} x_1 \\ \vdots \\ x_n \end{array} \boxed{f;g} \begin{array}{c} y_1 \\ \vdots \\ y_n \end{array} = \begin{array}{c} x_1 \\ \vdots \\ x_n \end{array} \boxed{f}\;\boxed{g} \begin{array}{c} y_1 \\ \vdots \\ y_n \end{array} \qquad \begin{array}{c} x_1 \\ \vdots \\ x_k \\ x'_1 \\ \vdots \\ x'_l \end{array} \boxed{f \mid\mid g} \begin{array}{c} y_1 \\ \vdots \\ y_k \\ y'_1 \\ \vdots \\ y'_l \end{array} = \begin{array}{c} \begin{array}{c} x_1 \\ \vdots \\ x_k \end{array} \boxed{f} \begin{array}{c} y_1 \\ \vdots \\ y_n \end{array} \\ \begin{array}{c} x'_1 \\ \vdots \\ x_l \end{array} \boxed{g} \begin{array}{c} y'_1 \\ \vdots \\ y'_l \end{array} \end{array}$$

$$\left.\begin{array}{c} x_1 \\ \vdots \\ x_n \\ x \end{array} \boxed{If[f,g,h]} \begin{array}{c} y_1 \\ \vdots \\ y_n \\ x \end{array}\right\} = \begin{cases} f\ (x_1,\dots,x_n) & \text{if } x>0 \\ g\ (x_1,\dots,x_n) & \text{if } x=0 \\ h\ (x_1,\dots,x_n) & \text{if } x<0 \end{cases} \qquad \left.\begin{array}{c} x_1 \\ \vdots \\ x_n \\ x \end{array} \boxed{It[f]} \begin{array}{c} y_1 \\ \vdots \\ y_n \\ x \end{array}\right\} = \underbrace{(f;\dots;f)}_{\mid x \mid}(x_1,\dots,x_n)$$

Figure 2 Generators of RPP.

▶ **Theorem 17** (Soudness & Completeness [17]). *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 stricly positive natural numbers, npos, as npos $= \mu X.\mathbb{1} \oplus X$. We define $\underline{n}$, the encoding of a positive natural number into a value of type npos as $\underline{1} = \texttt{fold}\,\texttt{inj}_l\,()$ and $\underline{n+1} = \texttt{fold}\,\texttt{inj}_r\,\underline{n}$. Finally, we define the type of integers as $Z = \mathbb{1} \oplus (\text{npos} \oplus \text{npos})$ along with $\overline{z}$ the encoding of any $z \in \mathbb{Z}$ into a value of type Z defined as: $\overline{0} = \texttt{inj}_l\,()$, $\overline{z} = \texttt{inj}_r\,\texttt{inj}_l\,\underline{z}$ for z positive, and $\overline{z} = \texttt{inj}_r\,\texttt{inj}_r\,\underline{-z}$ for z negative. Given some function $f \in \mathrm{RPP}^k$, we will build an iso $\mathrm{isos}(f) : Z^k \leftrightarrow Z^k$ which simulates f. $\mathrm{isos}(f)$ is defined by the size of the proof that f is in RPP^k.

▶ **Definition 18** (Encoding of the primitives).

- *The Sign-change of type $Z \leftrightarrow Z$ is* $\left\{\begin{array}{ccc} \texttt{inj}_r\,(\texttt{inj}_l\,x) & \leftrightarrow & \texttt{inj}_r\,(\texttt{inj}_r\,x) \\ \texttt{inj}_r\,(\texttt{inj}_r\,(x)) & \leftrightarrow & \texttt{inj}_r\,(\texttt{inj}_l\,x) \\ \texttt{inj}_l\,() & \leftrightarrow & \texttt{inj}_l\,() \end{array}\right\}$
- *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*
- The Successor is

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

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

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

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

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

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

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

We can now properly define $\mathrm{isos}(\mathit{It}[f])$ of type $Z^{k+1} \leftrightarrow Z^{k+1}$ as:

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

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

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

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

▶ **Theorem 24** (Simulation). *Let $f \in \mathrm{RPP}^k$ and $n_1, \dots, n_k$ elements of $\mathbb{Z}$ such that $f(n_1, \dots, n_k) = (m_1, \dots, m_k)$ then $\mathrm{isos}(f)(\overline{n_1}, \dots, \overline{n_k}) \to^* (\overline{m_1}, \dots, \overline{m_k})$*

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

4 Proof Theorical Content

We want to relate our langage of isos to proofs in a suitable logic. As mentioned earlier, an iso $\vdash_\omega \omega : A \leftrightarrow B$ 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 π 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 μMALL : linear logic with least and greatest fixed points, which allows us to reason about inductive and coinductive statements. μMALL 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 μMALL 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 μMALL, while more details can be found in [2].

4.1 Background on μMALL

Given an infinite set of variables $\mathcal{V} = \{X, Y, \dots\}$, we call *formulas* of μMALL the objects generated by $A, B ::= X \mid \mathbb{1} \mid \mathbb{0} \mid \top \mid \bot \mid A \otimes B \mid A \parr B \mid A \oplus B \mid A \,\&\, B \mid \mu X.A \mid \nu X.A$ where μ and ν bind the variable X in A. The negation on formula is defined in the usual way: $X^\perp = X, \mathbb{0}^\perp = \top, \mathbb{1}^\perp = \bot, (A \parr B)^\perp = A^\perp \otimes B^\perp, (A \oplus B)^\perp = A^\perp \,\&\, B^\perp, (\nu X.A)^\perp = \mu X.A^\perp$ having $X^\perp = X$ is harmless since we only deal with closed formulas.

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

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

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

Occurrences allow us to follow a subformula uniquely inside a derivation. Since in μMALL we only work with formula occurrences, we simply use the term *formula*.

The (possibly infinite) derivation trees of μMALL, 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 μMALL 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 $\vdash^f$, and an additional rule, $\dfrac{}{\vdash \Sigma}\, be(f)$, which represent a back-edge pointing to the sequent $\vdash^f$. We take the convention that from the root of the derivation from to rule $be(f)$ there must be exactly one sequent annotated by f.

$$
\frac{F \equiv G}{\vdash F^{\perp}, G}\ id
\qquad
\frac{\vdash \Sigma, F \quad \vdash \Phi, F^{\perp}}{\vdash \Sigma, \Phi}\ cut
\qquad
\frac{}{\vdash \top, \Sigma}\ \top
$$

$$
\frac{}{\vdash \mathbb{1}}\ \mathbb{1}
\qquad
\frac{\vdash F, G, \Sigma}{\vdash F \parr G, \Sigma}\ \parr
\qquad
\frac{\vdash F, \Sigma \quad \vdash G, \Phi}{\vdash F \otimes G, \Sigma, \Phi}\ \otimes
$$

$$
\frac{\vdash F, \Sigma \quad \vdash G, \Sigma}{\vdash F \,\&\, G, \Sigma}\ \&
\qquad
\frac{\vdash F_i, \Sigma}{\vdash F_1 \oplus F_2, \Sigma}\ \oplus^i\ i \in \{1,2\}
\qquad
\frac{\Sigma}{\vdash \Sigma, \perp}\ \perp
$$

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

Figure 3 Rules for μMALL.

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

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

While a circular proof has multiple finite 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:*

- $\mathcal{U}\left(P : \dfrac{P_1, \ldots, P_n}{\vdash \Sigma}\ r\ , v \right) = \dfrac{\mathcal{U}(P_1, v), \ldots, \mathcal{U}(P_n, v)}{\vdash \Sigma}\ r$
- $\mathcal{U}(be(f), v) = v(f)$
- $\mathcal{U}\left(P : \dfrac{P_1, \ldots, P_n}{\vdash^{f}\ \Sigma}\ r\ , v \right) = \left(\pi = \dfrac{\mathcal{U}(P_1, v'), \ldots, \mathcal{U}(P_n, v')}{\vdash \Sigma}\ r \right)$ *with $v'(g) = \pi$ if $g = f$ else $v(g)$.*

μMALL 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* [3, 2]: 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 ν 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 detail here. Example 31 features an example of a valid proof together with its thread. More details can be found in [2].

4.2 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 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 μMALL. This will imply that the proofs we obtain are indeed isomorphisms, meaning that if we cut the aforementioned proofs π and $\pi^\perp$, 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 $circ(\omega)$. We can define another translation into infinite derivations as the composition of $circ()$ with the unfolding: $[\![\omega]\!] = \mathcal{U}(circ(\omega))$.

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 μMALL in which contexts are split in two parts, written $\Upsilon; \Theta$, where Υ is a list of formulas and Θ 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 α) and end in the context $[]; \Theta$. The additional information of the variable in Θ 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 $\overline{\Theta} = \{F \mid x : F \in \Theta\}$ and $\underline{\Theta} = \{x : A \mid x : A_\alpha \in \Theta\}$. We also use another rule which allow to send the first formula from Υ to Θ and associate it a variable to the formula:

$$\frac{\Upsilon; x : F, \Theta \vdash G}{F :: \Upsilon; \Theta \vdash G}\ ex(x)$$

Given a derivation ι in this system, we write $[\![\iota]\!]$ for the function that sends ι into a derivation of μMALL where (i) we remove all occurrence of the exchange rule (ii) the contexts $[]; \Theta$ becomes $\overline{\Theta}$.

Given an iso $\omega : A \leftrightarrow B$ and initial addresses α, β, its translation into a derivation of μMALL 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 **fix** $f.\omega$ and calling the negative phase when encountering an iso of the form $\{v_1 \leftrightarrow e_1 \mid \dots \mid v_n \leftrightarrow e_n\}$ and attaching to the formulas A and B two distinct addresses α and β 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 e_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 way in which the values of type A on the left-hand-side of ω are decomposed. The negative phase works as follows: we consider a set of (list of values, typing judgement), 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 judgement 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 Υ 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 Θ. The negative phase ends when the list is empty and hence when $\Upsilon = []$. When it is the case, we can start decomposing 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. Keeping the variable-formula pair in the derivation is here to help us know how to split accordingly the context Θ when needed, while Υ 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 Figure 5.

▶ Remark 28. While μMALL is presented in a one-sided way, we write $\Sigma \vdash \Phi$ for $\vdash \Sigma^\perp, \Phi$ in order to stay closer to the formalism of the type system of isos.

▶ **Definition 29.** *$circ(\omega, S, \alpha, \beta) = \pi$ takes a well-typed iso, a singleton set S of an iso-variable corresponding to the last iso-variable seen in the induction definition of ω and two fresh addresses α, β and produces a circular derivation of the variant of μMALL described above with back-edges. $circ(\omega, S, \alpha, \beta)$ is defined inductively on ω:*

- $circ(\mathbf{fix}\ f.\omega, S, \alpha, \beta) = circ(\omega, \{f\}, \alpha, \beta)$
- $circ(f, \{f\}, \alpha, \beta) = \dfrac{}{A_\alpha \vdash B_\beta}\ be(f)$
- $circ(\{(v_i \leftrightarrow e_i)_{i\in I}\} : A \leftrightarrow B, \{f\}, \alpha, \beta) = \left\llbracket \dfrac{\mathtt{Neg}(([v_i], \xi_i)_{i\in I})}{A_\alpha \vdash^f B_\beta} \right\rrbracket$ *where ξ_i is the typing derivation of e_i.*

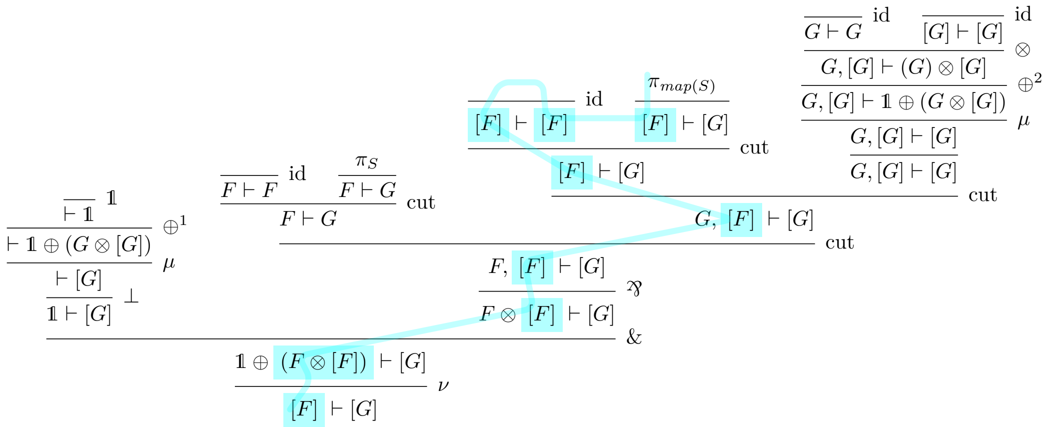
▶ **Example 30.** The translation $\llbracket circ(\omega, \emptyset, \alpha, \beta) \rrbracket$ of the iso ω from Example 9 is, with $F = A_{\alpha l}, G = B_{\alpha r l}, H = C_{\alpha r r}$ and $F' = A_{\beta r l}, G' = B_{\beta r r}, H' = C_{\beta l}$:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{[]; a : F \vdash F'}\ \mathrm{id}}{[]; a : F \vdash F' \oplus G'}\ \oplus^1}{[]; a : F \vdash H' \oplus (F' \oplus G')}\ \oplus^2}{[F]; \emptyset \vdash H' \oplus (F' \oplus G')}\ \mathrm{ex(a)} \qquad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{[]; b : G \vdash G'}\ \mathrm{id}}{[]; b : G \vdash F' \oplus G'}\ \oplus^2}{[]; b : G \vdash H' \oplus (F' \oplus G')}\ \oplus^2}{[G]; \emptyset \vdash H' \oplus (F' \oplus G')}\ \mathrm{ex(b)} \qquad \dfrac{\dfrac{\dfrac{}{[]; c : H \vdash H'}\ \mathrm{id}}{[]; c : H \vdash H' \oplus (F' \oplus G')}\ \oplus^1}{[H]; \emptyset \vdash H' \oplus (F' \oplus G')}\ \mathrm{ex(c)}}{[G \oplus H]; \emptyset \vdash H' \oplus (F' \oplus G')}\ \parr}{[F \oplus (G \oplus H)]; \emptyset \vdash H' \oplus (F' \oplus G')}\ \parr$$

▶ **Example 31.** Considering the iso swap of type $A \otimes B \leftrightarrow B \otimes A$ and its μMALL proof

$$\pi_S = \dfrac{\dfrac{\dfrac{}{A_{\gamma l} \vdash A_{\gamma' r}}\ id \qquad \dfrac{}{B_{\gamma r} \vdash B_{\gamma' l}}\ id}{A_{\gamma l}, B_{\gamma r} \vdash (B \otimes A)_{\gamma'}}\ \otimes}{(A \otimes B)_\gamma \vdash (B \otimes A)_{\gamma'}}\ \parr$$

Following Example 10 we give its corresponding proof $\pi_{map(S)}$ where $F = (A \otimes B)_{\alpha i r l}$ and $G = (B \otimes A)_{\beta i r l}$, then $[F]$ and $[G]$ are respectively of address α and β:



We painted in blue the pre-thread that follows the *focus* of the structurally recursive criterion. During the *negative phase* which consists of the $\nu, \&, \parr, \perp$ rules the pre-thread is going up, at each time going into the subformula corresponding to the *focus*. Then, during the *positive phase* the pre-thread is not active 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 [2] and that it is valid: among the formulas it visits infinitely often, the minimal formula is $[F]$, which is a ν formula as since $[F]$ is on the left hand side of the sequent, we get $[F] = (\mu X.\mathbb{1} \oplus (F \otimes X))^{\perp}$.

▶ **Lemma 32.** *Given $\pi = circ(\omega)$, for each infinite branch of π, 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. ◀

Among the terms that we translate, the translation of a value yields what we call a *Purely Positive Proof*: a finite derivation whose only rules have for active formula the sole formula on the right of the sequent. Any such derivation is trivially a valid pre-proof.

▶ **Definition 33** (Purely Positive Proof)**.** *A Purely Positive Proof is a finite, cut-free proof whose rules are only $\oplus^i, \otimes, \mu, \mathbb{1}, id$ for $i \in \{1, 2\}$.*

▶ **Lemma 34** (Values are Purely Positive Proofs)**.** *Given $x_1 : A^1, \ldots, x_n : A^n \vdash v : A$ then* $\dfrac{[\![v]\!]}{[]; x_1 : A^1_{\alpha_1}, \ldots, x_n : A^n_{\alpha_n} \vdash A_\alpha}$ *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 :* $\dfrac{\dfrac{\pi}{\Sigma \vdash G} \quad \dfrac{}{G \vdash F}\, be(f)}{\Sigma \vdash F}\, cut$

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 $[\![circ(\omega)]\!]$ is a premise of a bouncing-cut.*

$$\frac{\texttt{Neg}(\{(\texttt{inj}_l\ v_j :: l_j, \xi_j)_{j\in J}\} \cup \{(\texttt{inj}_r\ v_k :: l_k, \xi_k)_{k\in K}\})}{F_1 \oplus F_2 :: \Upsilon; \Theta \vdash G} =$$

$$\cfrac{\cfrac{\texttt{Neg}(\{(v_j :: l_j, \xi_j)_{j\in J}\})}{F_1 :: \Upsilon; \Theta \vdash G} \qquad \cfrac{\texttt{Neg}(\{(v_k :: l_k, \xi_k)_{k\in K}\})}{F_2 :: \Upsilon; \Theta \vdash G}}{F_1 \oplus F_2 :: \Upsilon; \Theta \vdash G}\ \&$$

$$\frac{\texttt{Neg}(\{([], \xi)\})}{[]; \Theta \vdash G} = \frac{\texttt{Pos}(\xi)}{[]; \Theta \vdash G} \qquad \frac{\texttt{Neg}(\{(() :: l, \xi)\})}{\mathbb{1} :: \Upsilon; \Theta \vdash G} = \cfrac{\cfrac{\texttt{Neg}(\{l, \xi\})}{\Upsilon; \Theta \vdash G}}{\mathbb{1} :: \Upsilon; \Theta \vdash G}\ \top$$

$$\frac{\texttt{Neg}(\{(\langle v_i^1, v_i^2\rangle :: l_i, \xi_i)_{i\in I}\})}{F_1 \otimes F_2 :: \Upsilon; \Theta \vdash G} = \cfrac{\cfrac{\texttt{Neg}(\{(v_i^1 :: v_i^2 :: l_i, \xi_i)_{i\in I}\})}{F_1, F_2 :: \Upsilon; \Theta \vdash G}}{F_1 \otimes F_2 :: \Upsilon; \Theta \vdash G}\ \parr$$

$$\frac{\texttt{Neg}(\{(\texttt{fold}\ v_i :: l_i, \xi_i)_{i\in I}\})}{\mu X.F :: \Upsilon; \Theta \vdash G} = \cfrac{\cfrac{\texttt{Neg}(\{(v_i :: l_i, \xi_i)_{i\in I}\})}{F[X \leftarrow \mu X.F] :: \Upsilon; \Theta \vdash G}}{\mu X.F :: \Upsilon; \Theta \vdash G}\ \nu$$

$$\frac{\texttt{Neg}(\{(x :: l, \xi)\})}{F :: \Upsilon; \Theta \vdash G} = \cfrac{\cfrac{\texttt{Neg}(\{l, \xi\})}{\Upsilon; \Theta, x : F \vdash G}}{F :: \Upsilon; \Theta \vdash G}\ ex(x)$$

Figure 4 Negative Phase.

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* $\vdash_\omega \omega : A \leftrightarrow B$ *and* $\pi = \llbracket circ(\omega, \emptyset, \alpha, \beta) \rrbracket$ *then* π *satisfies* μ*MALL validity criterion from [2].*

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 get that any infinite branch is completely defined by a single iso-variable, which allows us to reason entirely about a single recursive iso $\textbf{fix}\ f.\{v_1 \leftrightarrow e_1 \mid \ldots \mid v_n \leftrightarrow e_n\}$. For each infinite branch, we will build a pre-thread that follows the focus of the primitive recursive criterion. We know that the focus is a *strict subvariable* of the argument that is called recursively, as a consequence we can split the constructed thread into two parts, p_0 and p_1, corresponding respectively to the *negative phase* and the *positive phase*. We also know that, each argument of a recursive call gives us a *purely positive proof* which is made only of tensors. We can show that the size of p_0 is bigger than p_1 and also that p_1 is a prefix of p_0. This allows us to make sure that our pre-thread is a thread where the *visible part* always encounters a ν formula. Finally, the inductive type is decomposed in the negative phase and not in the positive phase (as the right-hand side of a recursive call is purely made of tensors), we can show that (i) the thread is never stationary and (ii) the thread has for minimal recurring formula that is visited infinitely often a ν formula, hence satisfying validity. ◀

$$\texttt{Pos}\left(\frac{}{\vdash_e ():\mathbb{1}}\right) = \frac{}{[];\emptyset\vdash\mathbb{1}_\alpha}\ \mathbb{1}$$

$$\texttt{Pos}\left(\frac{}{x:A\vdash_e x:A}\right) = \frac{}{[];x:A_\alpha\vdash A_\beta}\ id$$

$$\texttt{Pos}\left(\frac{\frac{\xi}{\underline{\Theta}\vdash_e t:A_1}}{\underline{\Theta}\vdash_e \texttt{inj}_l\ t:A_1\oplus A_2}\right) = \frac{\frac{\texttt{Pos}(\xi)}{\Theta\vdash(A_1)_{\alpha l}}}{[];\Theta\vdash(A_1\oplus A_2)_\alpha}\ \oplus^1$$

$$\texttt{Pos}\left(\frac{\frac{\xi}{\underline{\Theta}\vdash_e t:A_2}}{\underline{\Theta}\vdash_e \texttt{inj}_r\ t:A_1\oplus A_2}\right) = \frac{\frac{\texttt{Pos}(\xi)}{[];\Theta\vdash(A_2)_{\alpha r}}}{[];\Theta\vdash(A_1\oplus A_2)_\alpha}\ \oplus^2$$

$$\texttt{Pos}\left(\frac{\frac{\xi_1}{\underline{\Theta}_1\vdash_e t_1}\quad\frac{\xi_2}{\underline{\Theta}_2\vdash_e t_2:A_2}}{\underline{\Theta}_1,\underline{\Theta}_2\vdash_e\langle t_1,t_2\rangle:A_1\otimes A_2}\right) = \frac{\frac{\texttt{Pos}(\xi_1)}{[];\Theta_1\vdash(A_1)_{\alpha l}}\quad\frac{\texttt{Pos}(\xi_2)}{[];\Theta_2\vdash(A_2)_{\alpha r}}}{[];\Theta_1,\Theta_2\vdash(A_1\otimes A_2)_\alpha}\ \otimes$$

$$\texttt{Pos}\left(\frac{\frac{\xi}{\underline{\Theta}\vdash_e t:A[X\leftarrow\mu X.A]}}{\underline{\Theta}\vdash_e \texttt{fold}\ t:\mu X.A}\right) = \frac{\frac{\texttt{Pos}(\xi)}{[];\Theta\vdash(A[X\leftarrow\mu X.A])_{\alpha i}}}{[];\Theta\vdash(\mu X.A)_\alpha}\ \mu$$

$$\texttt{Pos}\left(\frac{\frac{\xi_1}{\underline{\Theta}_1\vdash_e t_1:A_1\otimes\cdots\otimes A_n}\quad\frac{\xi_2}{\underline{\Theta}_2,x_1:A_1,\dots,x_n:A_n\vdash_e t_2:B}}{\underline{\Theta}_1,\underline{\Theta}_2\vdash_e \texttt{let}\,(x_i)_{i\in I}=t_1\ \texttt{in}\ t_2:B}\right) =$$

$$\frac{\frac{\texttt{Pos}(\xi_1)}{[];\Theta_1\vdash F_1\otimes\cdots\otimes F_n}\quad\frac{\texttt{Neg}(([(x_i)_{i\in I}],\xi_2))}{[F_1\otimes\cdots\otimes F_n];\Theta_2\vdash B_\alpha}}{[];\Theta_1,\Theta_2\vdash B_\alpha}\ \texttt{cut}$$

$$\texttt{Pos}\left(\frac{\Psi\vdash_\omega \omega:A\leftrightarrow B\quad\frac{\xi}{\underline{\Theta}\vdash_e t:A}}{\underline{\Theta};\Psi\vdash_e \omega\ t:B}\right) = \frac{\frac{\texttt{Pos}(\xi)}{[];\Theta\vdash A}\quad\frac{circ(\omega,\{f\},\alpha,\beta)}{[A];\emptyset\vdash B_\beta}}{[];\Theta\vdash B_\beta}\ \texttt{cut}$$

Figure 5 Positive Phase.

We can also show that the rewriting rules of the language simulate the cut-elimination procedure, as it is described in [2]:

▶ **Theorem 38** (Simulation)**.** *Provided an iso $\vdash_\omega\ \omega : A \leftrightarrow B$ and values $\vdash_e v : A$ and $\vdash_e v' : B$, let $\pi = \texttt{Pos}(\omega\ v)$ and $\pi' = \texttt{Pos}(v)$, if $\omega\ v \to^* v'$ then $\pi \to^*_{cut\text{-}elim} \pi'$.*

Proof sketch. The proof relies on the definition of a novel explicit substitution rewriting system for the language, called $\to_{e\beta}$. Explicit substitution are represented as a series of *let* constructs where the substitution of a variable by a value only occurs when we reach the term $\texttt{let}\,x = v\ \texttt{in}\ x$. Each rewriting step of this system represents exactly one step of the cut-elimination procedure of μMALL. Then we only need to show that the explicit substitution rewriting system matches, meaning that if $\sigma = \{\overrightarrow{x} \mapsto \overrightarrow{v}\}$ then $\texttt{let}\ \overrightarrow{x} = \overrightarrow{v}\ \texttt{in}\ e \to^*_{e\beta} \sigma(e)$. ◀

This leads to the following corollary:

▶ **Corollary 39** (Isomorphism of proofs.)**.** *Given a well-typed iso $\vdash_\omega\ \omega : A \leftrightarrow B$ and two well-typed close value v_1 of type A and v_2 of type B and the proofs $\pi : F_1 \vdash G_1$, $\pi^\perp : G_2 \vdash F_1$, $\phi : F_3$, $\psi : G_2$ corresponding respectively to the translation of $\omega, \omega^\perp, v_1, v_2$ then:*

$$\frac{\phi}{\vdash F_3} \leftrightsquigarrow \frac{\frac{\frac{\phi}{\vdash F_3}\quad\frac{\pi}{F_1\vdash G_1}}{\vdash G_1}\,cut\quad\frac{\pi^\perp}{G_2\vdash F_2}}{\vdash F_2}\ \texttt{cut} \qquad \frac{\frac{\frac{\psi}{\vdash G_3}\quad\frac{\pi^\perp}{F_2\vdash G_2}}{\vdash F_2}\,cut\quad\frac{\pi}{F_1\vdash G_1}}{\vdash G_1}\ \texttt{cut} \rightsquigarrow \frac{\psi}{\vdash G_3}$$

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 μMALL proofs. The latter result rests on the fact that our isos are *structurally recursive*.

Future works. We showned a one-way encoding from isos to proofs of μMALL, it is clear that there exists proof-isomorphisms of μMALL that does not correspond to an iso of our language, for instance taking the reversible map function on streams. Therefore, a first extension to our work would be to consider a two-way encoding and adding coinductive types in the language. 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 generalised quantum loops: in [20], lists are the only recursive type which is captured and recursion is terminating. The logic μMALL would help in providing a finer understanding of termination and non-termination

References

1. David Baelde. Least and greatest fixed points in linear logic. *ACM Transactions on Computational Logic (TOCL)*, 13(1):1–44, 2012.
2. 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. doi:10.1145/3531130.3533375.
3. 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. doi:10.4230/LIPIcs.CSL.2016.42.
4. Henk Barendregt. The lambda calculus: its syntax and semantics. *Studies in logic and the foundations of Mathematics*, 1984.
5. Yves Bertot and Pierre Castéran. *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions.* Springer Science & Business Media, 2013.
6. 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.
7. Haskell B Curry. Functionality in combinatory logic. *Proceedings of the National Academy of Sciences*, 20(11):584–590, 1934.
8. Edward Fredkin and Tommaso Toffoli. Conservative logic. *International Journal of theoretical physics*, 21(3):219–253, 1982.
9. 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.

10 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.

11 Rosham P. James and Amr Sabry. Theseus: A high-level language for reversible computing. Draft, available at `https://legacy.cs.indiana.edu/~sabry/papers/theseus.pdf`, 2014.

12 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.

13 Robin Kaarsgaard and Mathys Rennela. Join inverse rig categories for reversible functional programming, and beyond, 2021. Draft, available at `arXiv:2105.09929`.

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

15 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.

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

17 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`.

18 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.

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

20 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`.

21 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.