Semantics for a Turing-Complete Reversible Programming Language with Inductive Types

Kostia Chardonnet, Louis Lemonnier, Benoît Valiron

▶ To cite this version:

Kostia Chardonnet, Louis Lemonnier, Benoît Valiron. Semantics for a Turing-Complete Reversible Programming Language with Inductive Types. FSCD 2024 - 9th International Conference on Formal Structures for Computation and Deduction, Jul 2024, Tallinn, Estonia. pp.19:1-19:19, 10.4230/LIPIcs.FSCD.2024.19 . hal-04636603

HAL Id: hal-04636603
https://hal.science/hal-04636603

Submitted on 5 Jul 2024

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Semantics for a Turing-Complete Reversible Programming Language with Inductive Types

Kostia Chardonnet
Department of Computer Science and Engineering, University of Bologna, Italy

Louis Lemonnier[1]

Université Paris-Saclay, CNRS, ENS Paris-Saclay, Inria, Laboratoire Méthodes Formelles, Gif-sur-Yvette, France

Benoît Valiron
Université Paris-Saclay, CNRS, CentraleSupélec, ENS Paris-Saclay, Inria, Laboratoire Méthodes Formelles, Gif-sur-Yvette, France

Abstract

This paper is concerned with the expressivity and denotational semantics of a functional higher-order reversible programming language based on Theseus. In this language, pattern-matching is used to ensure the reversibility of functions. We show how one can encode any Reversible Turing Machine in said language. We then build a sound and adequate categorical semantics based on join inverse categories, with additional structures to capture pattern-matching and to interpret inductive types and recursion. We then derive a notion of completeness in the sense that any computable, partial, first-order injective function is the image of a term in the language.

2012 ACM Subject Classification Theory of computation → Program semantics

Keywords and phrases Reversible programming, functional programming, Computability, Denotational Semantics

Digital Object Identifier 10.4230/LIPIcs.FSCD.2024.19

Funding This work has been partially funded by the French National Research Agency (ANR) by the projects ANR-21-CE48-0019, ANR-19-CE48-0014, ANR-22-CE47-0012, and within the framework of "Plan France 2030", under the project ANR-22-PETQ-0007, ANR-22-PNCQ-0001.
Kostia Chardonnet: is partially supported by the MIUR FARE project CAFFEINE, "Compositional and Effectful Program Distances", R20LW7EJ7L.

Acknowledgements The authors thank Vladimir Zamdzhiev for his expert insight on specific parts of the denotational semantics.

1 Introduction

Originally, reversible computation has emerged as an energy-preserving model of computation in which no data is ever erased. This comes from Laundauer's principle which states that the erasure of information is linked to the dissipation of energy as heat [30, 5]. In reversible computation, given some process f, there always exists an inverse process f^{-1} such that their composition is equal to the identity: it is always possible to "*go back in time*" and recover the input of your computation. Although this can be seen as very restrictive, non-reversible computation can be emulated in a reversible setting by keeping track of intermediate results. As discussed in [4], the simulation of standard computation with reversible computation can be understood as a notion of *Turing completeness* – provided we accept that the final result comes together with auxiliary, intermediate computation.

[1] Corresponding author.

© Kostia Chardonnet, Louis Lemonnier, and Benoît Valiron;
licensed under Creative Commons License CC-BY 4.0
9th International Conference on Formal Structures for Computation and Deduction (FSCD 2024).
Editor: Jakob Rehof; Article No. 19; pp. 19:1–19:19

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Reversible computation has since been shown to be a versatile model. In the realm of quantum computation, reversible computing is at the root of the construction of *oracles*, subroutines describing problem instances in quantum algorithms [34]. Most of the research in reversible circuit design can then been repurposed to design efficient quantum circuits. On the theoretical side, reversible computing serves the main ingredient in several operational models of linear logic, whether through token-based Geometry of Interaction [32] or through the Curry-Howard correspondence for μMALL [8, 6].

Reversible programming has been approached in two different ways. The first one, based on Janus and later R-CORE and R-WHILE [31, 40, 15, 39], considers imperative and flow-chart languages. The other one follows a functional approach [38, 37, 20, 19, 36, 8]: a function $A \to B$ in the language represents a function – a bijection – between values of type A and values of type B. In this approach, types are typically structured, and functional reversible languages usually feature pattern-matching to discriminate on values.

One of the issues reversible programming has to deal with is *non-termination*: in general, a reversible program computes a *partial injective map*. This intuition can be formalised with the concept of *inverse categories* [27, 9, 10, 11]: categories in which every morphism comes with a partial inverse, for which the category **PInj** of sets and partial injective maps is the emblematic concrete instance. This categorical setting has been successfully used in the study of reversible programming semantics, whether based on flow-charts [14, 22], with recursion [2, 24, 23, 26], with side effects [18, 17], *etc.*

Although much work has been dedicated to the categorical analysis of reversible computation, the *adequacy* of the developed categorical constructs with reversible functional programming languages has only recently been under scrutiny, either in *concrete* categories of partial isomorphisms [26, 25], or for simple, *non Turing-complete* languages [7]. A formal, categorical analysis of a (reversible) Turing-complete, reversible language is still missing.

Contributions. In this paper, we aim at closing this gap: we propose a Turing-complete (understood as in the reversible setting), reversible language, together with a categorical semantics. In particular, the contributions of this paper are as follows.

- A (reversible) Turing-complete, higher-order reversible language with inductive types. Building on the Theseus-based family of languages studied in [36, 7, 8, 6], we consider an extension with *inductive types*, general *recursion* and *higher-order* functions.
- Sound and adequate categorical semantics. We show how the language can be interpreted in join inverse rig categories. The result relies on the **DCPO**-enrichments of join inverse rig categories.
- A notion of completeness. We finally discuss how the interpretation of the language in the category **PInj** is complete in the sense that any *first-order* computable, partial injective function on the images of types is realisable within the language.

2 Language

In this section, we present a reversible language, unifying and extending the Theseus-based variants presented in the literature [36, 7, 8]. In particular, the language we propose features higher-order (unlike [7]), pairing, injection, inductive types (unlike [36]) and general recursion (unlike [8]). Functions in the language are based on pattern-matching, following a strict syntactic discipline: term variables in patterns should be used linearly, and clauses should be non-overlapping on the left *and* on the right (therefore enforcing non-ambiguity and injectivity). In [36, 7, 8] one also requires exhaustivity for totality. In this paper, we drop this condition in order to allow non-terminating behaviour.

Table 1 Grammar for terms and types.

$$
\begin{array}{lrl}
\text{(Base types)} & A, B ::= & \mathbb{1} \mid A \oplus B \mid A \otimes B \mid \mu X.A \mid X \\
\text{(Isos)} & T ::= & A \leftrightarrow B \mid T_1 \to T_2 \\
\text{(Values)} & v ::= & () \mid x \mid \mathtt{inj}_\ell\, v \mid \mathtt{inj}_r\, v \mid \langle v_1, v_2 \rangle \mid \mathtt{fold}\, v \\
\text{(Patterns)} & p ::= & x \mid \langle p_1, p_2 \rangle \\
\text{(Expressions)} & e ::= & v \mid \mathtt{let}\, p_1 = \omega\ p_2\ \mathtt{in}\ e \\
\text{(Isos)} & \omega ::= & \{v_1 \leftrightarrow e_1 \mid \cdots \mid v_n \leftrightarrow e_n\} \mid \mathtt{fix}\ \phi.\omega \mid \lambda\psi.\omega \mid \phi \mid \omega_1\ \omega_2 \\
\text{(Terms)} & t ::= & () \mid x \mid \mathtt{inj}_\ell\, t \mid \mathtt{inj}_r\, t \mid \langle t_1, t_2 \rangle \mid \\
& & \mathtt{fold}\, t \mid \omega\ t \mid \mathtt{let}\, p = t_1\ \mathtt{in}\ t_2
\end{array}
$$

The language is presented in Table 1. It consist of two layers.

- Base types: The base types consist of the unit type $\mathbb{1}$ along with its sole constructor $()$, coproduct $A \oplus B$ and tensor product $A \otimes B$ with their respective constructors, $\mathtt{inj}_\ell\,(t), \mathtt{inj}_r\,(t)$ and $\langle t_1, t_2 \rangle$. Finally, the language features inductive types of the form $\mu X.A$ where X is a type variable occurring in A and μ is its binder. Its associated constructor is $\mathtt{fold}\,(t)$. The inductive type $\mu X.A$ can then be unfolded into $A[\mu X.A/X]$, i.e., substituting each occurrence of X by $\mu X.A$ in A. Typical examples of inductive types that can be encoded this way are the natural number, as $\mathtt{nat} = \mu X.(\mathbb{1} \oplus X)$ or the lists of types A, noted $[A] = \mu X.\mathbb{1} \oplus (A \otimes X)$. Note that we only work with closed types. We shall denote term-variables with x, y, z.
- Isos types: The language features isos, denoted ω, higher order reversible functions whose types T consist either of a pair of base type, noted $A \leftrightarrow B$ or function types between isos, $T_1 \to T_2$. Note that the word *iso* comes from isomorphism. However, in this paper, we have freed some constraints; in our case, isos are *forward deterministic* and *backward deterministic*, meaning that each value has at most one image and at most one value that has the former as image. A first-order iso of type $A \leftrightarrow B$ consists of a finite set of *clauses*, written $v \leftrightarrow e$ where v is a value of type A and e an expression of type B. An expression consists of a succession of applications of isos to some argument, described by $\mathtt{let}$ constructions: $\mathtt{let}\,(x_1, \dots, x_n) = \omega\ (y_1, \dots, y_n)\ \mathtt{in}\ e$. Isos can take other isos as arguments through the $\lambda\phi.\omega$ construction. Finally, isos can also represent *recursive computation* through the $\mathtt{fix}\ \phi.\omega$ construction, where ϕ is an *iso-variable*. In general, we shall denote iso-variable by $\phi_1, \phi_2, \dots$ and we use the shorthands $\mathtt{fix}\ \overrightarrow{\phi}$ or $\mathtt{fix}\ \phi_1, \dots, \phi_n$ and $\lambda\overrightarrow{\phi}$ or $\lambda\phi_1, \dots, \phi_n$ for $\mathtt{fix}\ \phi_1.\mathtt{fix}\ \phi_2.\dots\mathtt{fix}\ \phi_n.$ and $\lambda\phi.\lambda\phi_2.\dots\lambda\phi_n$.

Convention. We write $(t_1, \dots, t_n)$ for $\langle t_1, \langle \dots, t_n \rangle\rangle$ and $\bigoplus^n A$ (resp. $\bigotimes^n A$) for $A \oplus \cdots \oplus A$ (resp. $A \otimes \cdots \otimes A$) n times and $\omega_1 \dots \omega_n t$ for a succession of $\mathtt{let}$ constructions applying ω_n to ω_1. We also consider constructors to be right-associative, meaning that $\mathtt{fold}\ \mathtt{inj}_r\ \langle x, y \rangle$ should be read as $\mathtt{fold}\ (\mathtt{inj}_r\ (\langle x, y \rangle))$. To avoid conflicts between variables, we will always work up to α-conversion and use Barendregt's convention [3, p.26], which consists of keeping the names of all bound and free variables distinct, even when this remains implicit.

Typing judgements. Both base terms and isos feature their typing judgements, given in Table 2 and Table 3. Term typing judgements are of the form $\Psi; \Delta \vdash t \colon A$ where Δ is a context of term-variables of type A and Ψ is a context of iso-variables of type T and isos

Table 2 Typing rules for terms.

$$\frac{}{\Psi;\emptyset \vdash ()\colon \mathbb{1}} \qquad \frac{}{\Psi; x\colon A \vdash x\colon A} \qquad \frac{\Psi;\Delta \vdash t\colon A}{\Psi;\Delta \vdash \texttt{inj}_\ell\, t\colon A \oplus B} \qquad \frac{\Psi;\Delta \vdash t\colon B}{\Psi;\Delta \vdash \texttt{inj}_r\, t\colon A \oplus B}$$

$$\frac{\Psi;\Delta_1 \vdash t_1\colon A \quad \Psi;\Delta_2 \vdash t_2\colon B}{\Psi;\Delta_1,\Delta_2; \vdash \langle t_1, t_2\rangle\colon A \otimes B} \qquad \frac{\Psi;\Delta \vdash t\colon A[\mu X.A/X]}{\Psi;\Delta \vdash \texttt{fold}\, t\colon \mu X.A} \qquad \frac{\Psi \vdash_\omega \omega\colon A \leftrightarrow B \quad \Psi;\Delta \vdash t\colon A}{\Psi;\Delta \vdash \omega\ t\colon B}$$

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

Table 3 Typing rules for isos.

$$\frac{}{\Psi,\phi\colon T \vdash_\omega \phi\colon T} \qquad \frac{\Psi,\phi\colon T \vdash_\omega \omega\colon T}{\Psi \vdash_\omega \texttt{fix}\ \phi.\omega\colon T}$$

$$\frac{\Psi \vdash_\omega \omega_1\colon T_1 \quad \Psi \vdash_\omega \omega_2\colon T_1 \to T_2}{\Psi \vdash_\omega \omega_2\ \omega_1\colon T_2} \qquad \frac{\Psi,\phi\colon T_1 \vdash_\omega \omega\colon T_2}{\Psi \vdash_\omega \lambda\phi.\omega\colon T_1 \to T_2}$$

$$\frac{\begin{array}{ccc} \Psi;\Delta_1 \vdash v_1\colon A \quad \ldots \quad \Psi;\Delta_n \vdash v_n\colon A & & \forall i \neq j, v_i \perp v_j \\ \Psi;\Delta_1 \vdash e_1\colon B \quad \ldots \quad \Psi;\Delta_n \vdash e_n\colon B & & \forall i \neq j, e_i \perp e_j \end{array}}{\Psi \vdash_\omega \{v_1 \leftrightarrow e_1 \mid \cdots \mid v_n \leftrightarrow e_n\}\colon A \leftrightarrow B.}$$

typing judgements are of the form $\Psi \vdash_\omega \omega\colon T$. While Δ is a *linear* context, Ψ is not, as an iso represents a closed computation, and can be duplicated or erased at will. In the last rule of Table 3, the term variables in Δ are bound by the pattern-matching construction: they are not visible outside of the term, thus not appearing anymore in the typing context of the conclusion.

While [8] and [36] require isos to be exhaustive (i.e. to cover all the possible values of their input types) and non-overlapping (i.e. two clauses cannot match the same value), we relax the exhaustivity requirement in this paper, in the spirit of what was done in [7]. Non-overlapping is formalised by the notion of *orthogonality* between values, noted $v_1 \perp v_2$.

▶ **Definition 1** (Orthogonality). *We introduce a binary relation $\perp$ on terms. Given two terms t_1, t_2, $t_1 \perp t_2$ holds if it can be derived inductively with the rules below; we say that t_1 and t_2 are orthogonal. The relation $\perp$ is defined as the smallest relation such that:*

$$\frac{}{\texttt{inj}_\ell\, t_1 \perp \texttt{inj}_r\, t_2} \qquad \frac{}{\texttt{inj}_r\, t_1 \perp \texttt{inj}_\ell\, t_2} \qquad \frac{t_1 \perp t_2}{C_\perp[t_1] \perp C_\perp[t_2]},$$

where the contexts $C_\perp$ are defined using the following grammar:

$$C_\perp ::= [-] \mid \texttt{inj}_\ell\, C_\perp \mid \texttt{inj}_r\, C_\perp \mid \langle C_\perp, t\rangle \mid \langle t, C_\perp\rangle \mid \texttt{fold}\, C_\perp \mid \texttt{let}\, p = t\ \texttt{in}\ C_\perp$$

Operational semantics. The language comes equipped with a rewriting system $\to$ on terms, defined in Table 4. As usual, we write $\to^*$ for the reflexive transitive closure of $\to$. The evaluation contexts $C_\to$ are defined by the grammar $[\,] \mid \texttt{inj}_\ell\, C_\to \mid \texttt{inj}_r\, C_\to \mid \{v_1 \leftrightarrow e_1 \mid \cdots \mid v_n \leftrightarrow e_n\}\ C_\to \mid \texttt{let}\, p = C_\to\ \texttt{in}\ t \mid \langle C_\to, v\rangle \mid \langle v, C_\to\rangle \mid C_\to\ t \mid \texttt{fold}\, C_\to$. Note how the rewriting system follows a *call-by-value* strategy on terms and values, requiring that

Table 4 Evaluation relation $\to$.

$$\frac{}{\texttt{fix}\ \phi.\omega \to \omega[\texttt{fix}\ \phi.\omega/\phi]} \qquad \frac{}{(\lambda\phi.\omega_1)\omega_2 \to \omega_1[\omega_2/\phi]} \qquad \frac{\omega_1 \to \omega_1'}{\omega_1\omega_2 \to \omega_1'\omega_2} \qquad \frac{\omega \to \omega'}{\omega\ t \to \omega'\ t}$$

$$\frac{\sigma(v_i) = v'}{\{v_1 \leftrightarrow e_1 \mid \cdots \mid v_n \leftrightarrow e_n\}\ v' \to \sigma(e_i)} \qquad \frac{t_1 \to t_2}{C_\to[t_1] \to C_\to[t_2]} \qquad \frac{\sigma(p) = v}{\texttt{let}\ p = v\ \texttt{in}\ t \to \sigma(t)}$$

the argument of an iso be fully evaluated to a value before firing the substitution. On the contrary, we follow a *call-by-name* strategy to simplify the manipulation of the fixpoint. Note that unlike [8, 36], we do not require any form of termination and isos are not required to be exhaustive: the rewriting system can diverge or be stuck. The evaluation of an iso applied to a value is dealt with by pattern-matching: the input value will try to match one of the values from the clauses and potentially create a substitution if the two values match, giving the corresponding expression as an output under that substitution. A substitution σ is a mapping from a set of variables to terms. The substitution of σ on an expression t, written $\sigma(t)$, is defined in the usual way by $\sigma(()) = ()$; $\sigma(x) = v$ if $\{x \mapsto v\} \subseteq \sigma$; $\sigma(\texttt{inj}_r\ (t)) = \texttt{inj}_r\ (\sigma(t))$; $\sigma(\texttt{inj}_\ell\ (t)) = \texttt{inj}_\ell\ (\sigma(t))$; $\sigma(\texttt{fold}\ (t)) = \texttt{fold}\ (\sigma(t))$ $\sigma(\langle t, t'\rangle) = \langle \sigma(t), \sigma(t')\rangle$; $\sigma(\omega\ t) = \omega\ \sigma(t)$ and $\sigma(\texttt{let}\ p = t_1\ \texttt{in}\ t_2) = (\texttt{let}\ p = \sigma(t_1)\ \texttt{in}\ \sigma(t_2))$. The support of a substitution, written $\operatorname{supp}(\sigma)$, is defined as $\{x \mid (x \mapsto v) \in \sigma\}$.

▶ **Lemma 2** (Subject Reduction). *If $\Psi; \Delta \vdash t\colon A$ and $t \to t'$, then $\Psi; \Delta \vdash t'\colon A$.* ⌟

The proof is similar to what has been done in [8]. As the rewriting system is deterministic, confluence is direct; meanwhile, as we are concerned with partial functions, progress is not guaranteed: a term can be stuck, for example, $\{\texttt{inj}_\ell\ (x) \leftrightarrow e\}\ \texttt{inj}_r\ (v)$ does not reduce.

Inversion. Finally, any iso $\omega\colon T$ can be inverted into an iso $\omega^{-1}\colon T^{-1}$, such that their composition makes up the identity. Intuitively, if ω is of type $A \leftrightarrow B$, then ω^{-1} will be of type $B \leftrightarrow A$. Inversion is defined as follows. Given an iso-type T, we define its inverse T^{-1} as: $(A \leftrightarrow B)^{-1} = B \leftrightarrow A$ and $(T_1 \to T_2)^{-1} = T_1^{-1} \to T_2^{-1}$. Given an iso ω, we define its dual ω^{-1} as: $\phi^{-1} = \phi$; $(\texttt{fix}\ \phi.\omega)^{-1} = \texttt{fix}\ \phi.\omega^{-1}$; $(\omega_1\ \omega_2)^{-1} = (\omega_1)^{-1}(\omega_2)^{-1}$; $(\lambda\phi.\omega)^{-1} = \lambda\phi.(\omega)^{-1}$ and $\{(v_i \leftrightarrow e_i)_{i\in I}\}^{-1} = \{((v_i \leftrightarrow e_i)^{-1})_{i\in I}\}$ and

$$\begin{pmatrix} v_1 & \leftrightarrow & \texttt{let}\ p_1 = \omega_1\ p_1'\ \texttt{in} \\ & & \cdots \\ & & \texttt{let}\ p_n = \omega_n\ p_n'\ \texttt{in}\ v_1' \end{pmatrix}^{-1} := \begin{pmatrix} v_1' & \leftrightarrow & \texttt{let}\ p_n' = \omega_n^{-1}\ p_n\ \texttt{in} \\ & & \cdots \\ & & \texttt{let}\ p_1' = \omega_1^{-1}\ p_1\ \texttt{in}\ v_1 \end{pmatrix}.$$

▶ **Property 3** (Inversion is an involution). *For any well-typed iso ω, we have $(\omega^{-1})^{-1} = \omega$.*

Proof. By a straightforward induction on ω, notice that if $\omega = \{v_1 \leftrightarrow e_1 \mid \cdots \mid v_n \leftrightarrow e_n\}$ then by definition we swap twice the order of the `let` construction, hence recovering the original term. ◀

▶ **Lemma 4** (Inversion is well-typed). *If $\phi_1\colon A_1 \leftrightarrow B_1 \ldots \phi_n\colon A_n \leftrightarrow B_n \vdash_\omega \omega\colon T$, then $\phi_1\colon B_1 \leftrightarrow A_1 \ldots \phi_n\colon B_n \leftrightarrow A_n \vdash_\omega \omega^{-1}\colon T^{-1}$.* ⌟

▶ **Lemma 5** (Inversion is preserved by evaluation). *If $\omega \to \omega'$ then $\omega^{-1} \to \omega'^{-1}$.* ⌟

▶ **Theorem 6** (Semantics of isos and their inversions [8]). *For all well-typed isos $\vdash_\omega \omega\colon A \leftrightarrow B$, and for all well-typed values $\vdash v\colon A$, if $(\omega\ (\omega^{-1}\ v)) \to^* v'$ then $v = v'$.* ⌟

▶ **Example 7.** Remember that $[A] = \mu X.\mathbb{1} \oplus (A \otimes X)$. One can define the *map* operator on lists with an iso of type $(A \leftrightarrow B) \to [A] \leftrightarrow [B]$, defined as

$$\lambda\psi.\texttt{fix}\ \phi.\{[\,] \leftrightarrow [\,] \ \mid\ h :: t \leftrightarrow \texttt{let}\, h' = \psi\ h\ \texttt{in}\ \texttt{let}\, t' = \phi\ t\ \texttt{in}\ h' :: t'\}\,,$$

with the terms $[\,] = \texttt{fold}\,(\texttt{inj}_\ell\,(()))$, representing the empty list, while the head and tail of the list is represented with $h :: t = \texttt{fold}\,(\texttt{inj}_r\,(\langle h, t\rangle))$. Its inverse map^{-1} is

$$\lambda\psi.\texttt{fix}\ \phi.\{[\,] \leftrightarrow [\,] \ \mid\ h' :: t' \leftrightarrow \texttt{let}\, t = \phi\ t'\ \texttt{in}\ \texttt{let}\, h = \psi\ h'\ \texttt{in}\ h :: t\}\,.$$

Note that in the latter, the variable ψ has type $B \leftrightarrow A$. If we consider the inverse of the term $(\text{map}\ \omega)$ we would obtain the term $(\text{map}^{-1}\ \omega^{-1})$ where ω^{-1} would be of type $B \leftrightarrow A$.

▶ **Example 8** (Cantor Pairing). One can encode the Cantor Pairing between $\mathbb{N} \otimes \mathbb{N} \leftrightarrow \mathbb{N}$. First recall that the type of natural number $\texttt{nat}$ is given by $\mu X.\mathbb{1} \oplus X$, then define $\overline{n}$ as the encoding of natural numbers into a closed value of type $\texttt{nat}$ as $\overline{0} = \texttt{fold}\,(\texttt{inj}_\ell\,())$ and given a variable x of type $\texttt{nat}$, its successor is $\overline{S(x)} = \texttt{fold}\,(\texttt{inj}_r\,(x))$. Omitting the $\overline{\cdot}$ operator for readability, the pairing is then defined as:

$$\omega_1 : \texttt{nat} \otimes \texttt{nat} \leftrightarrow (\texttt{nat} \otimes \texttt{nat}) \oplus \mathbb{1} = \left\{\begin{array}{lcl} \langle S(i), j\rangle & \leftrightarrow & \texttt{inj}_\ell\,(\langle i, S(j)\rangle) \\ \langle 0, S(S(j))\rangle & \leftrightarrow & \texttt{inj}_\ell\,(\langle S(j), 0\rangle) \\ \langle 0, S(0)\rangle & \leftrightarrow & \texttt{inj}_\ell\,(\langle 0, 0\rangle) \\ \langle 0, 0\rangle & \leftrightarrow & \texttt{inj}_r\,(()) \end{array}\right\},$$

$$\omega_2 : (\texttt{nat} \otimes \texttt{nat}) \oplus \mathbb{1} \leftrightarrow \texttt{nat} = \left\{\begin{array}{lcl} \texttt{inj}_\ell\,(x) & \leftrightarrow & \texttt{let}\, y = \phi\ x\ \texttt{in}\ S(y) \\ \texttt{inj}_r\,(()) & \leftrightarrow & 0 \end{array}\right\},$$

$$\text{CantorPairing} : \texttt{nat} \otimes \texttt{nat} \leftrightarrow \texttt{nat} = \texttt{fix}\ \phi.\left\{\begin{array}{lcl} x & \leftrightarrow & \texttt{let}\, y = \omega_1\ x\ \texttt{in} \\ & & \texttt{let}\, z = \omega_2\ y\ \texttt{in}\ z \end{array}\right\},$$

where the variable ϕ in ω_2 is the one being bound by the $\texttt{fix}$ of the CantorPairing iso. Intuitively, ω_1 realises one step of the Cantor Pairing evaluation while ω_2 checks if we reached the end of the computation and either applies a recursive call, or stops.

For instance, CantorPairing $\langle 1, 1\rangle$ will match with the first clause of ω_1, evaluating into $\texttt{inj}_\ell\,\langle 0, 2\rangle$, and then, inside ω_2 the reduction CantorPairing $\langle 0, 2\rangle$ will be triggered through the recursive call, evaluating the second clause of ω_1, reducing to $\texttt{inj}_\ell\,\langle 1, 0\rangle$, etc.

3 Expressivity

This section is devoted to assessing the expressivity of the language. To that end, we rely on Reversible Turing Machine (RTM) [1]. We describe how to encode an RTM as an iso, and prove that the iso realises the string semantics of the RTM.

3.1 Recovering duplication, erasure and manipulation of constants

Although the language is linear and reversible, since closed values are all finite, and one can build isos to encode notions of duplication, erasure, and constant manipulation thanks to partiality.

▶ **Definition 9** (Duplication). We define Dup^S_A the iso of type $A \leftrightarrow A \otimes A$ which can duplicate any closed value of type A by induction on A, where S is a set of pairs of a type-variable X and an iso-variable ϕ, such that for every free-type-variable $X \subseteq A$, there exists a unique pair $(X, \phi) \in S$ for some ϕ.

The iso is defined by induction on A: $\text{Dup}^S_{\mathbb{1}} = \{() \leftrightarrow \langle (), ()\rangle\}$, and

$$\text{Dup}^S_{A \otimes B} = \left\{\begin{array}{lcl} \langle x, y\rangle & \leftrightarrow & \texttt{let}\,\langle x_1, x_2\rangle = \text{Dup}^S_A\ x\ \texttt{in}\ \texttt{let}\,\langle y_1, y_2\rangle = \text{Dup}^S_B\ y\ \texttt{in} \\ & & \langle\langle x_1, y_1\rangle, \langle x_2, y_2\rangle\rangle \end{array}\right\};$$

- $\mathrm{Dup}^S_{A\oplus B} = \left\{ \begin{array}{lcl} \mathtt{inj}_\ell\ (x) & \leftrightarrow & \mathtt{let}\ \langle x_1, x_2\rangle = \mathrm{Dup}^S_A\ \ x\ \mathtt{in}\ \langle \mathtt{inj}_\ell\ (x_1), \mathtt{inj}_\ell\ (x_2)\rangle \\ \mathtt{inj}_r\ (y) & \leftrightarrow & \mathtt{let}\ \langle y_1, y_2\rangle = \mathrm{Dup}^S_B\ \ y\ \mathtt{in}\ \langle \mathtt{inj}_r\ (y_1), \mathtt{inj}_r\ (y_2)\rangle \end{array} \right\}$;
- If $(X, _) \notin S$: $\mathrm{Dup}^S_{\mu X.A} = \mathtt{fix}\ \phi. \left\{ \begin{array}{c} \mathtt{fold}\ (x) \leftrightarrow \mathtt{let}\ \langle x_1, x_2\rangle = Dup^{S\cup\{(X,\phi)\}}_{A[\mu X.A/X]}\ x\ \mathtt{in} \\ \langle \mathtt{fold}\ (x_1), \mathtt{fold}\ (x_2)\rangle \end{array} \right\}$;
- If $(X, \phi) \in S$: $\mathrm{Dup}^S_{\mu X.A} = \{x \leftrightarrow \mathtt{let}\ \langle x_1, x_2\rangle = \phi\ x\ \mathtt{in}\ \langle x_1, x_2\rangle\}$.

Remember that bound variables are assumed distinct following Barendregt's convention, allow for the well-definition of the isos above.

▶ **Lemma 10** (Properties of Duplication). *Given a closed type A, then $\mathrm{Dup}^\emptyset_A$ is well-defined, and the iso $\mathrm{Dup}^\emptyset_A$ is well typed of type $A \leftrightarrow A \otimes A$.* ⌟

▶ **Lemma 11** (Semantics of Duplication). *Given a closed type A and a closed value v of type A, then $\mathrm{Dup}^\emptyset_A\ \ v \to^* \langle v_1, v_2\rangle$ and $v = v_1 = v_2$.* ⌟

▶ **Definition 12** (Constant manipulation). We define $\mathrm{erase}_v\colon A \otimes \Sigma^T \leftrightarrow A$ which erases its second argument when its value is v as $\{\langle x, v\rangle \leftrightarrow x\}$. Reversed, it turns any x into $\langle x, v\rangle$.

3.2 Definition of Reversible Turing Machine

▶ **Definition 13** (Reversible Turing Machine [1]). Let $M = (Q, \Sigma, \delta, b, q_s, q_f)$ be a Turing Machine, where Q is a set of states, $\Sigma = \{b, a_1, \ldots, a_n\}$ is a finite set of tape symbols (in the following, a_i and b always refer to elements of Σ), $\delta \subseteq \Delta = (Q \times [(\Sigma\times\Sigma) \cup \{\leftarrow, \downarrow, \rightarrow\}] \times Q)$ is a partial relation defining the transition relation such that there must be no transitions leading out of q_f nor into q_s, b a blank symbol and q_s and q_f the initial and final states. We say that M is a *Reversible Turing Machine* (RTM) if it is:

- *forward* deterministic: for any two distinct pairs of triples (q_1, a_1, q_1') and (q_2, a_2, q_2') in δ, if $q_1 = q_2$ then $a_1 = (s_1, s_1')$ and $a_2 = (s_2, s_2')$ and $s_1 \neq s_2$.
- *Backward* deterministic: for any two distinct pairs of triples (q_1, a_1, q_1') and (q_2, a_2, q_2') in δ, if $q_1' = q_2'$ then $a_1 = (s_1, s_1')$ and $a_2 = (s_2, s_2')$ and $s_1' \neq s_2'$.

▶ **Definition 14** (Configurations [1]). A *configuration* of a RTM is a tuple $(q, (l, s, r)) \in \mathrm{Conf} = Q \times (\Sigma^* \times \Sigma \times \Sigma^*)$ where q is the internal state, l, r are the left and right parts of the tape (as string) and $s \in \Sigma$ is the current symbol being scanned. A configuration is *standard* when the cursor is on the immediate left of a finite, blank-free string $s \in (\Sigma \setminus \{b\})^*$ and the rest is blank, i.e. it is in configuration $(q, (\epsilon, b, s))$ for some q, where ϵ is the empty string, representing an infinite sequence of blank symbols b.

▶ **Definition 15** (RTM Transition [1]). An RTM M in configuration $C = (q, (l, s, r))$ goes to a configuration $C' = (q', (l', s', r'))$, written $T \vdash C \rightsquigarrow C'$ in a single step if there exists a transition $(q, a, q') \in \delta$ where a is either (s, s'), and then $l = l'$ and $r = r'$ or $a \in \{\leftarrow, \downarrow, \rightarrow\}$, and we have for the case $a = \leftarrow$: $l' = l \cdot s$ and for $r = x \cdot r_2$ we have $s' = x$ and $r' = r_2$, similarly for the case $a = \rightarrow$ and for the case $a = \downarrow$ we have $l' = l$ and $r' = r$ and $s = s'$.

The semantics of an RTM is given on *standard configurations* of the form $(q, (\epsilon, b, s))$ where q is a state, ϵ is the finite string standing for a blank-filled tape, and s is the blank-free, finite input of the RTM.

▶ **Definition 16** (String Semantics [1]). The semantics of a RTM M, written $\mathrm{Sem}(M)$ is defined on standards configurations and is given by the set $\mathrm{Sem}(M) = \{(s, s') \in ((\Sigma\backslash\{b\})^* \times (\Sigma\backslash\{b\})^*) \mid M \vdash (q_s, (\epsilon, b, s)) \rightsquigarrow^* (q_f, (\epsilon, b, s'))\}$.

▶ **Theorem 17** (Properties of RTM [1]). *For all RTM M,* $\mathrm{Sem}(M)$ *is the graph of an injective function. Conversely, all injective computable functions (on a tape) are realisable by a RTM. Finally, any Turing Machine can be simulated by a Reversible Turing Machine.* ⌟

3.3 Encoding RTMs as Isos

A RTM configuration is a set-based construction that we can model using the type constructors available in our language. Because the transition relation δ is backward and forward deterministic, it can be encoded as an iso. Several issues need to be dealt with; we discuss them in this section.

Encoding configurations. The set of states $Q = \{q_1, \dots, q_n\}$ is modeled with the type $Q^T = \mathbb{1} \oplus \cdots \oplus \mathbb{1}$ (n times). The encoding of the state q_i is then a closed value q_i^T. They are pairwise orthogonal. The set Σ of tape symbols is represented similarly by $\Sigma^T = \mathbb{1} \oplus \cdots \oplus \mathbb{1}$, and the encoding of the tape symbol a is a^T. We then define the type of configurations in the obvious manner: a configuration $C = (q, (l, s, r))$ corresponds to a closed value $\mathrm{isos}(C)$ of type $Q^T \otimes ([\Sigma^T] \otimes \Sigma^T \otimes [\Sigma^T])$.

▶ **Definition 18** (Encoding of Configurations). We define the type of configurations as $C^T = (Q^T \otimes ([\Sigma^T] \otimes \Sigma \otimes [\Sigma^T]))$. Given a configuration $C = (q, ((\epsilon, a_1, \dots, a_n), a, (a'_1, \dots, a'_m, \epsilon)))$, it is encoded as $\mathrm{isos}(C) = (q^T, ([a_n^T, \dots, a_1^T], a^T, [a_1'^T, \dots, a_m'^T]))$. For example, the standard configuration $C = (q_s, (\epsilon, b, [a_1, \dots, a_n]))$ is represented as $\mathrm{isos}(C) = (q_s^T, ([], b^T, [a_1^T, \dots, a_n^T]))$.

Encoding the transition relation δ. A limitation of our language is that every subcomputation has to be reversible and does not support infinite data structures such as streams. In the context of RTMs, the empty string ϵ is identified with an infinite string of blank symbols. If this can be formalised in set theory, in our limited model, we cannot emit blank symbols out of thin air without caution.

In order to simulate an infinite amount of blank symbols on both sides of the tape during the evaluation, we provide an iso that grows the size of the two tapes on both ends by blank symbols at each transition step. The iso `growth` is shown in Table 5. It is built using three auxiliary functions, written in a Haskell-like notation. `len` sends a closed value $[v_1, \dots, v_n]$ to $\langle [v_1, \dots, v_n], \overline{n} \rangle$. `snoc′` sends $\langle [v_1, \dots, v_n], v, \overline{n} \rangle$ to $\langle [v_1, \dots, v_n, v], v, \overline{n} \rangle$. `snoc` sends $\langle [v_1, \dots, v_n], v \rangle$ to $\langle [v_1, \dots, v_n, v], v \rangle$. Finally, `growth` sends $\langle [a_1^T, \dots, a_n^T], [a_1'^T, \dots, a_m'^T] \rangle$ to $\langle [a_1^T, \dots, a_n^T, b^T], [a_1'^T, \dots, a_m'^T, b^T] \rangle$.

Now, given a RTM $M = (Q, \Sigma, \delta, b, q_s, q_f)$, a relation $(q, r, q') \in \delta$ is encoded as a clause between values $\mathrm{iso}(q, r, q') = v_1 \leftrightarrow v_2$ of type $C^T \leftrightarrow C^T$. These clauses are defined by case analysis on r as follows. When x, x', z, y and y' are variables:

- $\mathrm{iso}(q, \rightarrow, q') = (q^T, (x', z, y :: y')) \leftrightarrow \texttt{let}\,(l, r) = \texttt{growth}\ (x', y')\ \texttt{in}\ (q'^T, (z :: l, y, r))$,
- $\mathrm{iso}(q, \leftarrow, q') = (q^T, (x :: x', z, y')) \leftrightarrow \texttt{let}\,(l, r) = \texttt{growth}\ (x', y')\ \texttt{in}\ (q'^T, (l, x, z :: r))$,
- $\mathrm{iso}(q, \downarrow, q') = (q^T, (x', z, x')) \leftrightarrow \texttt{let}\,(l, r) = \texttt{growth}\ (x', y')\ \texttt{in}\ (q'^T, (l, z, r))$,
- $\mathrm{iso}(q, (s, s'), q') = (q^T, (x', s^T, y')) \leftrightarrow \texttt{let}\,(l, r) = \texttt{growth}\ (x', y')\ \texttt{in}\ (q'^T, (l, s'^T, r))$.

The encoding of the RTM M is then the iso $\mathrm{isos}(M)$ whose clauses are the encoding of each rule of the transition relation δ, of type $\mathrm{Conf}^T \leftrightarrow \mathrm{Conf}^T$.

Encoding successive applications of δ. The transition δ needs to be iterated until the final state is reached. This behavior can be emulated in our language using the iso `It`, defined in Table 5. The iso $\texttt{It}\,\omega$ is typed with $(A \leftrightarrow A \otimes \texttt{nat})$. Fed with a value of type A, it iterates ω until `ff` is met. It then returns the result together with the number of iterations.

Table 5 Some useful isos for the encoding.

$$\begin{array}{l} \texttt{len} : [A] \leftrightarrow [A] \otimes \texttt{nat} \\ \texttt{len}\,[\,] \quad \leftrightarrow ([\,], 0) \\ \texttt{len}\, h :: t \leftrightarrow \texttt{let}\,(t', n) = \texttt{len}\ t\ \texttt{in} \\ \qquad\qquad (h :: t', S(n)) \end{array}$$

$$\begin{array}{l} \texttt{snoc}' : [A] \otimes A \otimes \texttt{nat} \leftrightarrow [A] \otimes A \otimes \texttt{nat} \\ \texttt{snoc}'\,([\,], x, 0) \leftrightarrow \texttt{let}\,(x_1, x_2) = \mathrm{Dup}^{\emptyset}_A\ x\ \texttt{in} \\ \qquad\qquad ([x_1], x_2, 0) \\ \texttt{snoc}'\,(h :: t, x, S(n)) \leftrightarrow \texttt{let}\,(t', x', n') = \texttt{snoc}'(t, x, n)\ \texttt{in} \\ \qquad\qquad (h :: t', x', S(n')) \end{array}$$

$$\begin{array}{l} \texttt{snoc} : [A] \otimes A \leftrightarrow [A] \otimes A \\ \texttt{snoc}\,(x, y) \leftrightarrow \texttt{let}\,(x', n) = \texttt{len}\ x\ \texttt{in} \\ \qquad \texttt{let}\,(x'', y', n') = \texttt{snoc}'\ (x', y, n)\ \texttt{in} \\ \qquad \texttt{let}\, n'' = \{x\ \leftrightarrow\ Sx\}\ n'\ \texttt{in} \\ \qquad \texttt{let}\, z = \texttt{len}^{-1}\ (x'', n'')\ \texttt{in}\ (z, y') \end{array}$$

$$\begin{array}{l} \texttt{growth} : [\Sigma^T] \otimes [\Sigma^T] \leftrightarrow [\Sigma^T] \otimes [\Sigma^T] \\ \texttt{growth}\,(l, r) \leftrightarrow \texttt{let}\,\langle l', b_1\rangle = \texttt{snoc}\langle l, b^T\rangle\ \texttt{in} \\ \qquad \texttt{let}\,\langle r', b_2\rangle = \texttt{snoc}\langle r, b^T\rangle\ \texttt{in} \\ \qquad \texttt{let}\, l'' = \texttt{erase}_b\langle l', b_1\rangle\ \texttt{in} \\ \qquad \texttt{let}\, r'' = \texttt{erase}_b\langle r', b_2\rangle\ \texttt{in}\ (l'', r'') \end{array}$$

$$\begin{array}{l} \texttt{It} : (A \leftrightarrow A \otimes (\mathbb{1} \oplus \mathbb{1})) \to (A \leftrightarrow A \otimes \texttt{nat}) \\ \texttt{It}\,\psi\, x \leftrightarrow \texttt{let}\, y = \psi\ x\ \texttt{in} \\ \qquad \texttt{let}\, z = \left\{ \begin{array}{l} (y, \mathtt{tt}) \leftrightarrow \texttt{let}\,(z, n) = (\texttt{It}\,\psi)\ y\ \texttt{in}\ (z, S\ n) \\ (y, \mathtt{ff}) \leftrightarrow (y, 0) \end{array} \right\}\ y\ \texttt{in}\ z \end{array}$$

$$\begin{array}{l} \texttt{rmBlank} : [\Sigma] \leftrightarrow [\Sigma] \otimes \mathbb{N} \\ \texttt{rmBlank}\,[] \leftrightarrow ([], 0) \\ \texttt{rmBlank}\, b^T :: t \leftrightarrow \texttt{let}\,(t', n) = \texttt{rmBlank}\ t\ \texttt{in}\ (t', S(n)) \\ \texttt{rmBlank}\, a_1^T :: t \leftrightarrow ((a_1^T :: t), 0) \\ \vdots \qquad \vdots \qquad \vdots\ \vdots \\ \texttt{rmBlank}\, a_n^T :: t \leftrightarrow ((a_n^T :: t), 0) \end{array}$$

$$\begin{array}{l} \texttt{rev}_{\mathrm{aux}} : [A] \otimes [A] \leftrightarrow [A] \otimes [A] \\ \texttt{rev}_{\mathrm{aux}}\,([], y) \leftrightarrow ([], y) \\ \texttt{rev}_{\mathrm{aux}}\,(h :: t, y) \leftrightarrow \texttt{let}\,(h_1, h_2) = \mathrm{Dup}^{\emptyset}_A\ \ h\ \texttt{in} \\ \qquad \texttt{let}\,(t_1, t_2) = \phi(t, h_2 :: y)\ \texttt{in} \\ \qquad (h_1 :: t_1, t_2) \end{array}$$

$$\begin{array}{l} \texttt{rev} : [A] \leftrightarrow [A] \otimes [A] \\ \texttt{rev} = \{x \leftrightarrow \texttt{let}\,(t_1, t_2) = \texttt{rev}_{\mathrm{aux}}\ (x, [])\ \texttt{in}\ (t_1, t_2)\} \end{array}$$

$$\begin{array}{l} \texttt{cleanUp} : C^T \otimes \texttt{nat} \leftrightarrow C^T \otimes \texttt{nat} \otimes \texttt{nat} \otimes \texttt{nat} \otimes [\Sigma^T] \\ \texttt{cleanUp}\,((x, (l, y, r)), n) \leftrightarrow \texttt{let}\,(l', n_1) = \texttt{rmBlank}\ l\ \texttt{in} \\ \qquad \texttt{let}\,(r_{\mathrm{ori}}, r_{\mathrm{rev}}) = \texttt{rev}\ r\ \texttt{in} \\ \qquad \texttt{let}\,(r', n_2) = \texttt{rmBlank}\ r_{\mathtt{rev}}\ \texttt{in} \\ \qquad ((x, (l', y, r')), n, n_1, n_2, r_{\mathrm{ori}}) \end{array}$$

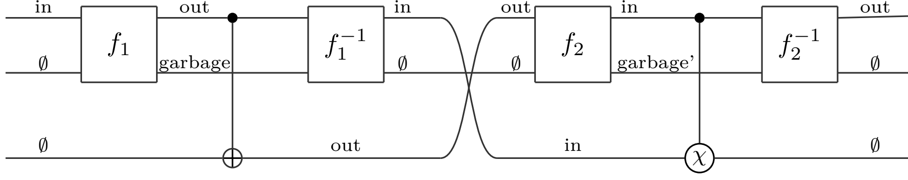

Figure 1 Reversibly removing additional garbage from some process.

To iterate iso(M), we then only need to modify iso to return a boolean stating whether q_f was met. This can be done straightforwardly, yielding an iso $\text{isos}_{\mathbb{B}}(M)$) of type $\text{Conf}^T \leftrightarrow \text{Conf}^T \otimes (\mathbb{1} \oplus \mathbb{1})$. With such an iso, given M be a RTM such that $M \vdash (q_s, (\epsilon, b, s)) \rightsquigarrow^{n+1} (q_f, (\epsilon, b, (a_1, \ldots, a_n)))$, then $\texttt{It}(\text{isos}_{\mathbb{B}}(M))\ (q_s^T, ([b^T], b^T, s^T))$ reduces to the encoding term $((q_f^T, ([b^T, \ldots, b^T], b^T, [a_1^T, \ldots, a_n^T, b^T, \ldots, b^T])), \overline{n})$. If it were not for the additional blank tape elements, we would have the encoding of the final configuration.

Recovering a canonical presentation. Removing blank states at the *beginning* of a list is easy: for instance, it can be done with the iso rmBlank, shown in Table 5. Cleaning up the tail of the list can then be done by reverting the list, using, e.g. rev in the same table. By abuse of notation, we use constants in some patterns: an exact representation would use Definition 12. Finally, we can define the operator `cleanUp`, solving the issue raised in the previous paragraph. In particular, given a RTM M and an initial configuration C such that $M \vdash C \rightsquigarrow C' = (q, (\epsilon, b, (a_1, \ldots, a_n)))$, then we have that $\texttt{cleanUp}\ (\texttt{It}(\text{isos}_{\mathbb{B}}(M))C^T) \to^* ((q^T, ([], b^T, [a_1^T, \ldots, a_n^T])), v)$, where v is of type $\texttt{nat} \otimes \texttt{nat} \otimes \texttt{nat} \otimes [\Sigma^T]$. If we want to claim that we indeed capture the operational behaviour of RTMS, we need to get rid of this value v.

Getting rid of the garbage. To discard this value v, we rely on Bennett's trick [4], shown in Figure 1. Given two Turing machines f_1 and f_2 and some input `in` such that if $f_1(\texttt{in}) = \texttt{out} \otimes \texttt{garbage}$ and $f_2(\texttt{out}) = \texttt{in} \otimes \texttt{garbage}'$, then the process consists of taking additional tapes in the Turing Machine in order to reversibly duplicate (represented by the $\oplus$) or reversibly erase some data (represented by the χ) in order to recover only the output of f_1, without any garbage.

Given an iso $\omega\colon A \leftrightarrow B \otimes C$ and $\omega'\colon B \leftrightarrow A \otimes C'$ where C, C' represent garbage, we can build an iso from $A \leftrightarrow B$ as follows, where the variables x, y, z (and their indices) respectively correspond to the first, second, and third wire of Figure 1. This operator makes use of the iso Dup discussed in Section 3.1.

$$\begin{array}{lcl} \texttt{GarbRem}(\omega, \omega')\ x_1 & \leftrightarrow & \texttt{let}\ \langle x_2, y\rangle = \omega\ x_1\ \texttt{in let}\ \langle x_3, z\rangle = \text{Dup}_B^{\emptyset}\ \ x_2\ \texttt{in} \\ & & \texttt{let}\ x_4 = \omega^{-1}\ \langle x_3, y\rangle\ \texttt{in let}\ \langle z_2, y_2\rangle = \omega'\ z\ \texttt{in} \\ & & \texttt{let}\ z_3 = (\text{Dup}_B^{\emptyset})^{-1}\ \langle z_2, x_4\rangle\ \texttt{in let}\ z_4 = \omega'^{-1}\ \langle z_3, y_2\rangle\ \texttt{in}\ z_4. \end{array}$$

▶ **Theorem 19** (Capturing the exact semantics of a RTM). *For all RTM M with standard configurations $C = (q_s, (\epsilon, b, s))$ and $C' = (q_f, (\epsilon, b, s'))$ such that $M \vdash C \rightsquigarrow^* C'$, we have*

$$\texttt{GarbRem}(\texttt{cleanUp}(\texttt{It}(\text{isos}_{\mathbb{B}}(M))), \texttt{cleanUp}(\texttt{It}(\text{isos}_{\mathbb{B}}(M^{-1}))))\ \ \text{isos}(C) \to^* \text{isos}(C')$$

The behavior of RTMs is thus captured by the language. ◀

4 Categorical Background

We aim at providing a denotational semantics for the programming language introduced above, meaning a mathematical interpretation abstract to the syntax. Our approach is categorical, in the spirit of many others before us. Programs are compositional by design, making it natural to interpret in a framework ruled by compositionality. Types are usually interpreted as objects in a category $\mathcal{C}$, and terms as morphisms in this category. We have seen that the main feature of our programming language is reversibility and its terms can be seen as partial isomorphisms, or partial injections. We want this property to be carried on the interpretation, and we present in this section the proper categories to do so. The category of sets and partial injective functions, written **PInj**, will be the recurring example throughout this section to help the intuition.

4.1 Join inverse rig category

The axiomatisation of join inverse rig categories gives the conditions for the morphisms of a category to be *partial injections*. First, the notion of restriction allows to capture the *actual* domain of a morphism through a partial identity function. Historically, *inverse* categories [27] were introduced before *restriction* categories, but the latter are more convenient to introduce the subject.

▶ **Definition 20** (Restriction [9]). A restriction structure is an operator that maps each morphism $f : A \to B$ to a morphism $\overline{f} : A \to A$ such that for all g and h such that the domain of g is A and the domain of h is B we have $f \circ \overline{f} = f$, $\overline{f} \circ \overline{g} = \overline{g} \circ \overline{f}$, $\overline{f \circ \overline{g}} = \overline{f} \circ \overline{g}$ and $\overline{h} \circ f = f \circ \overline{h \circ f}$. A morphism f is said to be *total* if $\overline{f} = 1_A$. A category with a restriction structure is called a *restriction category*. A functor $F : \mathcal{C} \to \mathcal{D}$ is a *restriction functor* if $\overline{F(f)} = F\left(\overline{f}\right)$ for all morphism f of $\mathcal{C}$. The definition is canonically extended to bifunctors. When unambiguous, we write gf for the composition $g \circ f$.

▶ **Example 21.** Given sets A, B and a partial function $f : A \to B$ defined on $A' \subseteq A$ and undefined on $A \setminus A'$, the restriction of f is $\overline{f} : A \to A$, the identity on $A' \subseteq A$ and undefined on $A \setminus A'$. This example shows that **PInj** is a restriction category.

To interpret reversibility, we need to introduce a notion of reversed process, a process that exactly reverses another process. This is given by a generalised notion of inverse.

▶ **Definition 22** (Inverse category [24]). An *inverse category* is a restriction category where all morphisms are partial isomorphisms; meaning that for $f : A \to B$, there exists a unique $f^\circ : B \to A$ such that $f^\circ \circ f = \overline{f}$ and $f \circ f^\circ = \overline{f^\circ}$.

▶ **Example 23.** In **PInj**, let us consider the partial function $f : \{0, 1\} \to \{0, 1\}$ as $f(0) = 1$ and undefined on 1. Its restriction $\overline{f}$ is undefined on 1 also but $\overline{f}(0) = 0$. Its *inverse* f° is undefined on 0 and such that $f^\circ(1) = 0$.

The example above generalises and **PInj** is an actual inverse category. Even more, it is *the* inverse category: [27] proves that every locally small inverse category is isomorphic to a subcategory of **PInj**.

▶ **Definition 24** (Restriction compatible [24]). Two morphisms $f, g : A \to B$ in a restriction category $\mathcal{C}$ are restriction compatible if $f\overline{g} = g\overline{f}$. The relation is written $f \smile g$. If $\mathcal{C}$ is an inverse category, they are inverse compatible if $f \smile g$ and $f^\circ \smile g^\circ$, noted $f \asymp g$. A set S of morphisms of the same type $A \to B$ is restriction compatible (*resp.* inverse compatible) if all elements of S are pairwise restriction compatible (*resp.* inverse compatible).

▶ **Definition 25** (Partial order [9]). Let $f, g : A \to B$ be two morphisms in a restriction category. We then define $f \leq g$ as $g\overline{f} = f$.

▶ **Definition 26** (Joins [16]). A restriction category $\mathcal{C}$ is equipped with joins if for all restriction compatible sets S of morphisms $A \to B$, there exists $\bigvee_{s \in S} s : A \to B$ morphism of $\mathcal{C}$ such that, whenever $t \colon A \to B$ and whenever for all $s \in S$, $s \leq t$, $s \leq \bigvee_{s \in S} s$, $\bigvee_{s \in S} s \leq t$, $\overline{\bigvee_{s \in S} s} = \bigvee_{s \in S} \overline{s}$, $f \circ \left(\bigvee_{s \in S} s\right) = \bigvee_{s \in S} fs$, $\left(\bigvee_{s \in S} s\right) \circ g = \bigvee_{s \in S} sg$. Such a category is called a *join restriction category*. An inverse category with joins is called a *join inverse category*.

Building up from Definition 20, a *join restriction functor* is a restriction functor that preserves all thus constructed joins.

▶ **Definition 27** (Zero [24]). Since $\emptyset \subseteq \mathrm{Hom}_{\mathcal{C}}(A, B)$, and since all of its elements are restriction compatible, there exists a morphism $0_{A,B} \doteq \bigvee_{s \in \emptyset} s$, called *zero*. It satisfies the following equations: $f0 = 0$, $0g = 0$, $0^{\circ}_{A,B} = 0_{B,A}$, $\overline{0_{A,B}} = 0_{A,A}$.

▶ **Definition 28** (Restriction Zero). *A restriction category $\mathcal{C}$ has a* restriction zero *object 0 iff for all objects A and B, there exists a unique morphism $0_{A,B} \colon A \to B$ that factors through* 0 *and satisfies $\overline{0_{A,B}} = 0_{A,A}$.*

▶ **Definition 29** (Disjointness tensor [13]). An inverse category $\mathcal{C}$ is said to have a *disjointness tensor* if it is equipped with a symmetric monoidal restriction bifunctor $. \oplus . : \mathcal{C} \times \mathcal{C} \to \mathcal{C}$, with as unit a restriction zero 0 and morphisms $\iota_l : A \to A \oplus B$ and $\iota_r : B \to A \oplus B$ that are total, jointly epic, and such that their inverses are jointly monic and $\overline{\iota_l^{\circ}}\,\overline{\iota_r^{\circ}} = 0_{A \oplus B}$.

▶ **Definition 30** ([25]). Let us consider a join inverse category equipped with a symmetric monoidal tensor product $(\otimes, 1)$ and a disjointness tensor $(\oplus, 0)$ that are join preserving, and such that there are isomorphisms $\delta_{A,B,C} \colon A \otimes (B \oplus C) \to (A \otimes B) \oplus (A \otimes C)$ and $\nu_A \colon A \otimes 0 \to 0$. This is called a *join inverse rig category*.

4.2 DCPO-category

We use the vocabulary of enriched category theory to shorten the discussion in this section. The notions of enrichment required to understand the semantics later is basic and should not frighten the reader. Categories in computer science are usually *locally small*, meaning that given two objects A and B, there is a *set* of morphisms $A \to B$. Enrichment is the study of the structure of those sets of morphisms, which could be vector spaces or topological spaces for example, more details can be found in [28, 29, 33]. It turns out that homsets in join inverse rig categories are dcpos – directed-complete partial orders, i.e. a partial-ordered set with all directed joins. This allows us to consider fixpoints in homsets. **DCPO** is the category of directed complete partial orders and Scott-continuous functions – monotone functions preserving joins. Dcpos are often used for the denotational interpretation of different sorts of λ-calculi, and more generally, to interpret recursive functions or indefinite loops.

▶ **Definition 31** ([12]). A category enriched over **DCPO**, also called a **DCPO**-category, is a locally small category whose hom-sets are directly partial ordered and where composition is a continuous operation (*i.e.* a morphism in **DCPO**).

It is proven in [24] that a join inverse category can be considered enriched in **DCPO** without loss of generality.

▶ **Lemma 32.** *Let $\mathcal{C}$ be a join inverse rig category. The functors: $- \otimes - \colon \mathcal{C} \times \mathcal{C} \to \mathcal{C}$, $- \oplus - \colon \mathcal{C} \times \mathcal{C} \to \mathcal{C}$, $-^{\circ} \colon \mathcal{C}^{op} \to \mathcal{C}$ are* **DCPO***-functors, meaning that they preserve the dcpo structure of homsets.*

4.3 Compactness

Inductive data types are written in the syntax as some least fixed point. As said earlier, types are represented as objects in the category, and thus a type judgement is an object mapping, or rather an endofunctor. Here, we show how to consider fixed points of endofunctors in our categorical setting.

▶ **Definition 33** (Initial Algebra)**.** Given an endofunctor $F\colon \mathcal{C} \to \mathcal{C}$, an F-algebra is a pair of an object A and a morphism $f\colon FA \to A$. F-algebras form a category with F-algebras homomorphisms. An initial F-algebra is an initial object in the category of F-algebras.

▶ **Theorem 34** (Lambek's theorem)**.** *Given an endofunctor $F : \mathcal{C} \to \mathcal{C}$ and an F-initial algebra $(X, \alpha\colon FX \to X)$, α is an isomorphism.* ⌟

With Lambek's theorem, we know that an initial algebra provides an object X such that $X \cong FX$; X is a fixed point of the endofunctor F, as requested. The existence of such fixed points is given by the next theorem [12, Corollary 7.2.4].

▶ **Definition 35** (Ep-pair)**.** *Given a* **DCPO***-category $\mathcal{C}$, a morphism $e\colon X \to Y$ in $\mathcal{C}$ is called an* embedding *if there exists a morphism $p\colon Y \to X$ such that $p \circ e = \mathrm{id}_X$ and $e \circ p \leq \mathrm{id}_Y$. The morphisms e and p form an* embedding-projection pair *(e, p), also called* ep-pair*.*

We recall that an *ep-zero* [12, Definition 7.1.1], is an initial object such that every morphism with it as source is an embedding, and is also a terminal object such that every morphism with it as target is a projection.

▶ **Theorem 36.** *A* **DCPO***-category with an ep-zero and colimits of ω-chains of embeddings is parametrised* **DCPO***-algebraically ω-compact; meaning that for every* **DCPO***-functor $F\colon \mathcal{C} \times \mathcal{D} \to \mathcal{D}$, there is a pair consisting of a* **DCPO***-functor $F^{\between}\colon \mathcal{C} \to \mathcal{D}$ and an indexed family $\alpha^F = \{\alpha^F_A\colon F(A, F^{\between}A) \to F^{\between}A\}$ of initial $F(A, -)$-algebras. This pair is called a* parametrised *initial algebra.* ⌟

The hypotheses of the theorem above are verified by the categories we want to work with, without loss of generality.

▶ **Proposition 37** ([24])**.** *Any join inverse rig category can be faithfully embedded in a rig join inverse category with colimits of ω-chains of embeddings.* ⌟

5 Denotational semantics

We now show how to build a denotational semantics for the language we presented thus far. The semantics is akin to the one presented in [7] but with extra structure to handle inductive types and recursive functions. While the semantics is sound and adequate w.r.t. a notion of operational equivalence between terms, the main interest of the semantics rest in showing that, given some RTM M whose semantics is a function f, we show that the semantics of $\mathrm{isos}(M)$ is the same as f. This would provide us with a formal proof that any computable reversible function can be captured by an iso.

Types. Let us consider $\mathcal{C}$ a join inverse rig category (Definition 30). We can assume without loss of generality that $\mathcal{C}$ satisfies the hypothesis of Theorem 36. In order to deal with open types, we make use an auxiliary judgement for types, of the form $X_1, \dots, X_n \vDash A$, where $\{X_i\}_i$ is a subset of the free type variables appearing in A. We interpret this

kind of judgement as a **DCPO**-functor $\mathcal{C}^{|\Theta|} \to \mathcal{C}$ written $[\![\Theta \vDash A]\!]$. This can be formally defined as a (simple) inductive relation, and the semantics is defined similarly to what is done in [12, 21]. $[\![\Theta \vDash \mathbb{1}]\!]$ is the constant functor that maps to the tensor product unit. $[\![\Theta, X \vDash X]\!]$ is a projection. The other judgements are obtained by induction: if $[\![\Theta \vDash A]\!] = f$ and $[\![\Theta \vDash B]\!] = g$, then $[\![\Theta \vDash A \oplus B]\!] = \oplus \circ \langle f, g\rangle$ and $[\![\Theta \vDash A \otimes B]\!] = \otimes \circ \langle f, g\rangle$. Finally, $[\![\Theta \vDash \mu X.A]\!] = ([\![\Theta, X \vDash A]\!])^{\not\subset}$. All this is summed up in Table 6.

Table 6 Interpretation of types.

$$
\begin{aligned}
[\![\Theta \vDash A]\!] &\colon \mathcal{C}^{|\Theta|} \to \mathcal{C} \\
[\![\Theta, X \vDash X]\!] &= \Pi \\
[\![\Theta \vDash I]\!] &= K_1 \\
[\![\Theta \vDash A \oplus B]\!] &= \oplus \circ \langle [\![\Theta \vDash A]\!], [\![\Theta \vDash B]\!] \rangle \\
[\![\Theta \vDash A \otimes B]\!] &= \otimes \circ \langle [\![\Theta \vDash A]\!], [\![\Theta \vDash B]\!] \rangle \\
[\![\Theta \vDash \mu X.A]\!] &= ([\![\Theta, X \vDash A]\!])^{\not\subset}
\end{aligned}
$$

Lemma 32 and Theorem 36 ensure that this is well-defined. For closed types, we have $[\![\mathbb{1}]\!] = 1$, $[\![A \oplus B]\!] = [\![A]\!] \oplus [\![B]\!]$, $[\![A \otimes B]\!] = [\![A]\!] \otimes [\![B]\!]$ and $[\![\mu X.A]\!] \cong [\![A[\mu X.A/X]]\!]$. Ground iso types are represented by dcpos of morphisms in $\mathcal{C}$, written $[\![A \leftrightarrow B]\!] = \mathrm{Hom}_{\mathcal{C}}([\![A]\!], [\![B]\!])$. The type of iso functions $T_1 \to T_2$ is interpreted by the dcpo of Scott continuous maps between the two dcpos $[\![T_1]\!]$ and $[\![T_2]\!]$, written $[[\![T_1]\!] \to [\![T_2]\!]]$. The terms used to build isos are dependent in two contexts: variables in Δ and isos in Ψ. In general, if $\Delta = x_1 \colon A_1, \dots, x_m \colon A_m$ and $\Psi = \phi_1 \colon T_1, \dots, \phi_n \colon T_n$, then we set $[\![\Delta]\!] = [\![A_1]\!] \otimes \cdots \otimes [\![A_m]\!]$ and $[\![\Psi]\!] = [\![T_1]\!] \times \cdots \times [\![T_2]\!]$, with $\otimes$ being the monoidal product in $\mathcal{C}$ and $\times$ the cartesian product in **DCPO**.

Terms. A well-formed term judgement $\Psi; \Delta \vdash t \colon A$ has for semantics a Scott continuous map $[\![\Psi; \Delta \vdash t \colon A]\!] \in \mathbf{DCPO}([\![\Psi]\!], \mathcal{C}([\![\Delta]\!], [\![A]\!]))$, defined as in Table 7 when $g \in [\![\Psi]\!]$. All this is well-defined in **DCPO** provided that $[\![\Psi \vdash_\omega \omega \colon A \leftrightarrow B]\!]$ is. This last point is the focus of the next section.

▶ **Lemma 38.** *Given two judgements $\Psi; \Delta_1 \vdash t_1 \colon A$ and $\Psi; \Delta_2 \vdash t_2 \colon A$, such that $t_1 \perp t_2$, we have for all $g \in [\![\Psi]\!]$ the equality $[\![t_1]\!](g)^\circ \circ [\![t_2]\!](g) = 0_{[\![\Delta_2]\!], [\![\Delta_1]\!]}$.* ⌟

Isos. Isos do only depend on function variables, but they are innately morphisms, so their denotation will be similar to terms – a Scott continuous map. We define the denotation of an iso by induction on the typing rules. The interpretation of an iso-variable is direct, it is the projection on the last component. The interpretations of evaluations and λ-abstractions are usual in a cartesian closed category, in our case, **DCPO**. All the rules apart for the iso-abstraction are found in Table 7. The remaining rule, building an iso abstraction $\{v_i \leftrightarrow e_i\}_{i \in I}$, needs more details.

▶ **Lemma 39.** *Given a well-formed iso abstraction $\Psi \vdash_\omega \{v_i \leftrightarrow e_i\}_{i \in I} \colon A \leftrightarrow B$, for all $g \in [\![\Psi]\!]$, the morphisms in $\mathcal{C}$ given by $[\![\Psi; \Delta_i \vdash e_i \colon B]\!](g) \circ [\![\Psi; \Delta_i \vdash v_i \colon A]\!](g)^\circ$, with $i \in I$ are pairwise inverse compatible.* ⌟

Each clause $v_i \leftrightarrow e_i$ of an iso abstraction is given an interpretation $[\![e_i]\!] \circ [\![v_i]\!]^\circ$. The previous lemma shows that in the case of an iso abstraction, the interpretations of all clauses can be joined (in the sense of Definition 26). This join also generalises to the join in **DCPO** as shown by the lemma below.

Table 7 Denotational semantics of the language in a join inverse rig **DCPO**-category.

$$\begin{aligned}
[\![\Psi; \Delta \vdash t \colon A]\!](g) &\in \mathcal{C}([\![\Delta]\!], [\![A]\!]) \\
[\![\Psi; \emptyset \vdash * \colon I]\!](g) &= \mathrm{id}_{[\![I]\!]} \\
[\![\Psi; x \colon A \vdash x \colon A]\!](g) &= \mathrm{id}_{[\![A]\!]} \\
[\![\Psi; \Delta \vdash \mathtt{inj}_\ell\, t \colon A \oplus B]\!](g) &= \iota_l \circ [\![\Psi; \Delta \vdash t \colon A]\!](g) \\
[\![\Psi; \Delta \vdash \mathtt{inj}_r\, t \colon A \oplus B]\!](g) &= \iota_r \circ [\![\Psi; \Delta \vdash t \colon B]\!](g) \\
[\![\Psi; \Delta_1, \Delta_2 \vdash t_1 \otimes t_2 \colon A \otimes B]\!](g) &= [\![\Psi; \Delta_1 \vdash t_1 \colon A]\!](g) \otimes [\![\Psi; \Delta_2 \vdash t_2 \colon B]\!](g) \\
[\![\Psi; \Delta \vdash \mathtt{fold}\, t \colon \mu X.A]\!](g) &= \alpha^{[\![X \vdash A]\!]} \circ [\![\Psi; \Delta \vdash t \colon A[\mu X.A/X]]\!](g)
\end{aligned}$$

$$\begin{aligned}
[\![\Psi \vdash_\omega \omega \colon T]\!] &\in \mathbf{DCPO}([\![\Psi]\!], [\![T]\!]) \\
[\![\Psi, \phi \colon T \vdash_\omega \phi \colon T]\!] &= \pi_{[\![T]\!]} \\
[\![\Psi \vdash_\omega \omega_2\omega_1 \colon T_2]\!] &= \mathrm{eval} \circ \langle [\![\Psi \vdash_\omega \omega_2 \colon T_1 \to T_2]\!], [\![\Psi \vdash_\omega \omega_1 \colon T_1]\!] \rangle \\
[\![\Psi \vdash_\omega \lambda\phi.\omega \colon T_1 \to T_2]\!] &= \mathrm{curry}([\![\Psi, \phi \colon T_1 \vdash_\omega \omega \colon T_2]\!]) \\
[\![\Psi \vdash_\omega \mathtt{fix}\ \phi.\omega \colon T]\!] &= \mathtt{fix}\ ([\![\Psi, \phi \colon T \vdash_\omega \omega \colon T]\!])
\end{aligned}$$

▶ **Lemma 40.** *Given a dcpo Ξ, two objects X and Y of $\mathcal{C}$, a set of indices I and a family of Scott continuous maps $\xi_i \colon \Xi \to \mathcal{C}(X, Y)$ that are pairwise inverse compatible, the function $\bigvee_{i\in I} \xi_i : \Xi \to \mathcal{C}(X, Y)$ defined by $x \mapsto \bigvee_{i\in I} \xi_i(x)$ is Scott continuous.* ⌟

The interpretation of an iso abstraction is then given by:

$$[\![\Psi \vdash_\omega \{v_i \leftrightarrow e_i\}_{i\in I} \colon A \leftrightarrow B]\!] = \bigvee_{i\in I} (\mathrm{comp} \circ \langle [\![\Psi; \Delta_i \vdash e_i \colon B]\!], [\![\Psi; \Delta_i \vdash v_i \colon A]\!]^\circ \rangle)$$

The semantics is well-defined, in the sense that the interpretation of $\Psi \vdash_\omega \{v_i \leftrightarrow e_i\}_{i\in I} \colon A \leftrightarrow B$ is a Scott continuous map between the dcpos $[\![\Psi]\!]$ and $\mathcal{C}([\![A]\!], [\![B]\!])$.

6 Adequacy

We show a strong relationship between the operational semantics and the denotational semantics of the language. First, we fix a mathematical interpretation $[\![-]\!]$ in a join inverse rig category $\mathcal{C}$, that is **DCPO**-enriched and whose objects 0 and 1 are distinct.

Since the language handles non-termination, our adequacy statement links the denotational semantics to the notion of termination in the operational semantics: Given $\vdash t \colon A$, t is said to be *terminating* if there exists a value v such that $t \to^* v$. We either write $t\downarrow$, or $t \downarrow v$.

▶ **Theorem 41** (Adequacy). *Given $\vdash t \colon A$, $t\downarrow$ iff $[\![\vdash t \colon A]\!] \neq 0_{[\![A]\!]}$.*

Soundness. We start by showing the simple implication in Theorem 41 amount to soundness: the denotational semantics is stable w.r.t. computation.

▶ **Proposition 42** (Soundness). *Given a valid term judgement $\vdash t \colon A$, provided that $t \to t'$, then we have $[\![\vdash t \colon A]\!] = [\![\vdash t' \colon A]\!]$.* ⌟

We can conclude that if $\vdash t \colon A$ with $t\downarrow$, we have $[\![\vdash t \colon A]\!] \neq 0_{[\![A]\!]}$. This shows one of the implications in Theorem 41. For the proof of the other implication, we follow a syntactic approach, inspired by the proof in [35].

Proof of Adequacy. Our proof of adequacy involves a finitary sublanguage, where the number of recursive calls is controlled syntactically: instead of general fixpoints, we introduce a family of finitary fixpoints $\mathtt{fix}^n\ \phi.\omega$, unfolding n times before reducing to the empty iso $\{\}$, corresponding to the diverging iso.

We show the adequacy result for the finitary terms thanks to strong normalisation, and then show that it implies adequacy for the whole language; this is achieved by observing that a normalising finitary term is also normalising in its non-finitary form.

7 Semantics preservation

In this section, we fix the interpretation $[\![-]\!]$ of the language in **PInj**, the category of sets and partial injections. This choice comes without any loss of generality (see [27]), and allows us to consider *computable* functions. In this section, we show that given a computable, reversible function $f\colon [\![A]\!] \to [\![B]\!]$, there exists an iso $\omega\colon A \leftrightarrow B$ such that $[\![\omega]\!] = f$. In order to do that, we fix a canonical flat representation of our types.

7.1 A Canonical Representation

We define a canonical representation of closed values of some type A into a new type $\mathrm{Enc} = \mathbb{B} \oplus \mathbb{1} \oplus \mathbb{1} \oplus \mathbb{1} \oplus \mathbb{1} \oplus \mathtt{nat}$ (recall that $\mathbb{B} = \mathbb{1} \oplus \mathbb{1}$ and $\mathtt{nat} = \mu X.\mathbb{1} \oplus X$). For simplicity let us name each the following terms of type Enc : $\mathbf{tt} = \mathtt{inj}_\ell\ (\mathtt{inj}_\ell\ ())$, $\mathbf{ff} = \mathtt{inj}_\ell\ (\mathtt{inj}_r\ ())$, $S = \mathtt{inj}_r\ (\mathtt{inj}_\ell\ ())$, $D^{\oplus} = \mathtt{inj}_r\ (\mathtt{inj}_r\ (\mathtt{inj}_\ell\ ()))$, $D^{\otimes} = \mathtt{inj}_r\ (\mathtt{inj}_r\ (\mathtt{inj}_r\ (\mathtt{inj}_\ell\ ())))$, $D^{\mu} = \mathtt{inj}_r\ (\mathtt{inj}_r\ (\mathtt{inj}_r\ (\mathtt{inj}_r\ (\mathtt{inj}_\ell\ ()))))$, and for every natural number n, we write $\tilde{n}$ for the term $\mathtt{inj}_r\ (\mathtt{inj}_r\ (\mathtt{inj}_r\ (\mathtt{inj}_r\ (\mathtt{inj}_r\ (\mathtt{inj}_r\ (\overline{n}))))))$, where $\overline{n}$ is the encoding of natural numbers, as given in Example 8. Now, given some closed type A, we can define $\lfloor - \rfloor_A \colon A \leftrightarrow [\mathrm{Enc}]$ the iso that transform any close value of type A into a list of Enc. The iso is defined inductively over A: $\lfloor - \rfloor_{\mathbb{1}} = \{() \leftrightarrow [S]\}$, and

$$\lfloor - \rfloor_{A\oplus B} = \left\{ \begin{array}{lcl} \mathtt{inj}_\ell\ (x) & \leftrightarrow & \mathtt{let}\ y = \lfloor x \rfloor_A\ \mathtt{in}\ D^{\oplus} :: \mathbf{ff} :: y \\ \mathtt{inj}_r\ (x) & \leftrightarrow & \mathtt{let}\ y = \lfloor x \rfloor_B\ \mathtt{in}\ D^{\oplus} :: \mathbf{tt} :: y \end{array} \right\},$$

$$\lfloor - \rfloor_{A\otimes B} = \left\{ \begin{array}{lcl} \langle x, y\rangle & \leftrightarrow & \mathtt{let}\ x' = \lfloor x \rfloor_A\ \mathtt{in}\ \mathtt{let}\ y' = \lfloor y \rfloor_B\ \mathtt{in} \\ & & \mathtt{let}\ \langle z, n\rangle = +\!+\ \langle x', y'\rangle\ \mathtt{in}\ D^{\otimes} :: \tilde{n} :: z \end{array} \right\},$$

$$\lfloor - \rfloor_{\mu X.A} = \left\{ \ \mathtt{fold}\ x \ \leftrightarrow\ \mathtt{let}\ y = \lfloor x \rfloor_{A[\mu X.A/X]}\ \mathtt{in}\ D^{\mu} :: y \ \right\},$$

where the iso $+\!+\colon [A] \otimes [A] \leftrightarrow [A] \otimes \mathtt{nat}$ which concatenate two lists is defined as:

$$\mathtt{fix}\ f. \left\{ \begin{array}{lcl} \langle [], x\rangle & \leftrightarrow & \langle x, 0\rangle \\ \langle h :: t, x\rangle & \leftrightarrow & \mathtt{let}\ \langle y, n\rangle = f\ \langle t, x\rangle\ \mathtt{in}\ \langle h :: y, S(n)\rangle \end{array} \right\}.$$

7.2 Capturing every computable injection

With this encoding, every iso $\omega\colon A \leftrightarrow B$ can be turned into another iso $\lfloor \omega \rfloor\colon [\mathrm{Enc}] \leftrightarrow [\mathrm{Enc}]$ by composing $\lfloor - \rfloor_A$, followed by ω, followed by $\lfloor - \rfloor_B^{-1}$. This is in particular the case for isos that are the images of a Turing Machine. We are now ready to see how every computable function f from $[\![A]\!] \to [\![B]\!]$ can be turned into an iso whose semantics is f. Given a computable function $f\colon [\![A]\!] \to [\![B]\!]$, call M_f the RTM computing f. Since f is in **PInj**, its output uniquely determines its input. Following [4], given the output of the machine M_f there exists another Turing Machine M'_f which takes this output and recover the original input of M_f. In our encoding of a RTM, the iso will have another additional garbage which consist of a natural

number, i.e. the number of steps of the RTM M_f. Using $\texttt{GarbRem}(\text{isos}(M_f), \text{isos}(M'_f))$ we can obtain a single iso, from the encoding of A to the encoding of B, without any garbage left. This also ensures that $\left[\!\left[\texttt{GarbRem}(\text{isos}(M_f), \text{isos}(M'_f))\right]\!\right](x) = (\llbracket \text{isos}(M_f) \rrbracket (x))_1$, for any input x.

▶ **Theorem 43** (Computable function as Iso). *Given a computable function* $f \colon \llbracket A \rrbracket \to \llbracket B \rrbracket$*, let* $g \colon \llbracket [\text{Enc}] \otimes [\text{Enc}] \rrbracket \to \llbracket [\text{Enc}] \otimes [\text{Enc}] \rrbracket$ *be defined as* $g = \llbracket \lfloor - \rfloor_B \rrbracket \circ f \circ \llbracket \lfloor - \rfloor_A^{-1} \rrbracket$*, and let* $\omega \colon A \leftrightarrow B$ *be defined as* $\{x \leftrightarrow \texttt{let}\, y = \lfloor x \rfloor_A \;\texttt{in let}\, y' = \texttt{GarbRem}(\text{isos}(M_g), \text{isos}(M'_g))\; y \;\texttt{in let}\, z = \lfloor y' \rfloor_B^{-1} \;\texttt{in}\; z\}$*. Then* $\llbracket \omega \rrbracket = f$*.* ⌟

8 Conclusion

In this paper, we built upon the language presented in [7, 8, 36] in order to represent any partial injective function which can manipulate inductive types. We showed how one can encode any Reversible Turing Machine, hence the (reversible) Turing Completeness, and we gave a denotational semantics based on join inverse rig categories, together with a soundness and adequacy theorem. Most notably, we showed that for any computable function f from **PInj**, there exists an iso whose semantics is f, thus our language fully characterises all of the computable morphisms in **PInj**.

References

1. Holger Bock Axelsen and Robert Glück. A simple and efficient universal reversible turing machine. In Adrian-Horia Dediu, Shunsuke Inenaga, and Carlos Martín-Vide, editors, *Language and Automata Theory and Applications*, pages 117–128, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg. doi:10.1007/978-3-642-21254-3_8.
2. Holger Bock Axelsen and Robin Kaarsgaard. Join inverse categories as models of reversible recursion. In Bart Jacobs and Christof Löding, editors, *Proceedings of the 19th International Conference on Foundations of Software Science and Computation Structures (FOSSACS'16)*, volume 9634 of *Lecture Notes in Computer Science*, pages 73–90, Eindhoven, The Netherlands, 2016. Springer. doi:10.1007/978-3-662-49630-5_5.
3. Henk Barendregt. 'the lambda calculus: its syntax and semantics'. *Studies in logic and the foundations of Mathematics*, 1984.
4. Charles H Bennett. Logical reversibility of computation. *IBM Journal of Research and Development*, 17(6):525–532, 1973. doi:10.1147/rd.176.0525.
5. Antoine Bérut, Artak Arakelyan, Artyom Petrosyan, Sergio Ciliberto, Raoul Dillenschneider, and Eric Lutz. Experimental verification of landauer's principle linking information and thermodynamics. *Nature*, 483(7388):187–189, 2012.
6. Kostia Chardonnet. *Towards a Curry-Howard Correspondence for Quantum Computation*. Theses, Université Paris-Saclay, January 2023. URL: https://theses.hal.science/tel-03959403.
7. Kostia Chardonnet, Louis Lemonnier, and Benoît Valiron. Categorical semantics of reversible pattern-matching. *Electronic Proceedings in Theoretical Computer Science*, 351:18–33, December 2021. doi:10.4204/eptcs.351.2.
8. Kostia Chardonnet, Alexis Saurin, and Benoît Valiron. A Curry-Howard correspondence for linear, reversible computation. In Bartek Klin and Elaine Pimentel, editors, *31st EACSL Annual Conference on Computer Science Logic (CSL 2023)*, volume 252 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 13:1–13:18, Dagstuhl, Germany, 2023. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.CSL.2023.13.
9. J. Robin B. Cockett and Stephen Lack. Restriction categories I: Categories of partial maps. *Theoretical Computer Science*, 270(1):223–259, 2002. doi:10.1016/S0304-3975(00)00382-0.
10. J. Robin B. Cockett and Stephen Lack. Restriction categories ii: partial map classification. *Theoretical Computer Science*, 294(1):61–102, 2003. doi:10.1016/S0304-3975(01)00245-6.

11 Robin Cockett and Stephen Lack. Restriction categories III: Colimits, partial limits and extensivity. *Mathematical Structures in Computer Science*, 17(4):775–817, 2007. doi:10.1017/S0960129507006056.

12 M.P. Fiore. *Axiomatic Domain Theory in Categories of Partial Maps*. Distinguished Dissertations in Computer Science. Cambridge University Press, 2004. URL: https://books.google.co.uk/books?id=rsIAmbc2cIoC.

13 Brett Gordon Giles. *An Investigation of Some Theoretical Aspects of Reversible Computing*. PhD thesis, University of Calgary, 2014. doi:10.11575/PRISM/24917.

14 Robert Glück and Robin Kaarsgaard. A categorical foundation for structured reversible flowchart languages: Soundness and adequacy. *Log. Methods Comput. Sci.*, 14(3), 2018. doi:10.23638/LMCS-14(3:16)2018.

15 Robert Glück, Robin Kaarsgaard, and Tetsuo Yokoyama. Reversible programs have reversible semantics. In Emil Sekerinski, Nelma Moreira, José N. Oliveira, Daniel Ratiu, Riccardo Guidotti, Marie Farrell, Matt Luckcuck, Diego Marmsoler, José Campos, Troy Astarte, Laure Gonnord, Antonio Cerone, Luis Couto, Brijesh Dongol, Martin Kutrib, Pedro Monteiro, and David Delmas, editors, *Formal Methods. FM 2019 International Workshops - Porto, Portugal, October 7-11, 2019, Revised Selected Papers, Part II*, volume 12233 of *Lecture Notes in Computer Science*, pages 413–427. Springer, 2019. doi:10.1007/978-3-030-54997-8_26.

16 Xiuzhan Guo. *Products, Joins, Meets, and Ranges in Restriction Categories*. PhD thesis, University of Calgary, 2012. doi:10.11575/PRISM/4745.

17 Chris Heunen, Robin Kaarsgaard, and Martti Karvonen. Reversible effects as inverse arrows. In Sam Staton, editor, *Proceedings of the 34th Conference on the Mathematical Foundations of Programming Semantics (MFPS XXXIV)*, volume 341 of *Electronic Notes in Theoretical Computer Science*, pages 179–199, Dalhousie University, Halifax, Canada, 2018. Elsevier. doi:10.1016/j.entcs.2018.11.009.

18 Chris Heunen and Martti Karvonen. Reversible monadic computing. In Dan Ghica, editor, *Proceedings of the 31st Conference on the Mathematical Foundations of Programming Semantics (MFPS XXXI)*, volume 319 of *Electronic Notes in Theoretical Computer Science*, pages 217–237, Nijmegen, The Netherlands, 2015. doi:10.1016/j.entcs.2015.12.014.

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

20 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.

21 Xiaodong Jia, Bert Lindenhovius, Michael Mislove, and Vladimir Zamdzhiev. Commutative monads for probabilistic programming languages. In *2021 36th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*. IEEE, June 2021. doi:10.1109/lics52264.2021.9470611.

22 Robin Kaarsgaard. Condition/decision duality and the internal logic of extensive restriction categories. In Barbara König, editor, *Proceedings of the 35th Conference on the Mathematical Foundations of Programming Semantics (MFPS XXXV)*, volume 347 of *Electronic Notes in Theoretical Computer Science*, pages 179–202, London, UK, 2019. doi:10.1016/j.entcs.2019.09.010.

23 Robin Kaarsgaard. Inversion, iteration, and the art of dual wielding. In Michael Kirkedal Thomsen and Mathias Soeken, editors, *Proceedings of the 11th International Conference on Reversible Computation (RC 2019)*, volume 11497 of *Lecture Notes in Computer Science*, pages 34–50, Lausanne, Switzerland, 2019. Springer. doi:10.1007/978-3-030-21500-2_3.

24 Robin Kaarsgaard, Holger Bock Axelsen, and Robert Glück. Join inverse categories and reversible recursion. *Journal of Logical and Algebraic Methods in Programming*, 87:33–50, 2017. doi:10.1016/j.jlamp.2016.08.003.

25 Robin Kaarsgaard and Mathys Rennela. Join inverse rig categories for reversible functional programming, and beyond. In Ana Sokolova, editor, Proceedings 37th Conference on *Mathematical Foundations of Programming Semantics,* Hybrid: Salzburg, Austria and Online, 30th August - 2nd September, 2021, volume 351 of *Electronic Proceedings in Theoretical Computer Science*, pages 152–167. Open Publishing Association, 2021. `doi:10.4204/EPTCS.351.10`.

26 Robin Kaarsgaard and Niccolò Veltri. En garde! unguarded iteration for reversible computation in the delay monad. In Graham Hutton, editor, *Proceedings of the 13th International Conference on Mathematics of Program Construction (MPC 2019)*, volume 11825 of *Lecture Notes in Computer Science*, pages 366–384, Porto, Portugal, October 2019. Springer Verlag. `doi:10.1007/978-3-030-33636-3_13`.

27 J. Kastl. Inverse categories. In *Algebraische Modelle, Kategorien und Gruppoide*, Studien zur Algebra und ihre Anwendungen, Band 7, pages 51–60. Berlin, Akademie-Verlag, 1979.

28 G.M Kelly. Tensor products in categories. *Journal of Algebra*, 2(1):15–37, 1965. `doi:10.1016/0021-8693(65)90022-0`.

29 Max Kelly. *Basic concepts of enriched category theory*, volume 64. CUP Archive, 1982.

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

31 Christopher Lutz. Janus: a time-reversible language. Letter to Rolf Landauer, posted online by Tetsuo Yokoyama on `http://www.tetsuo.jp/ref/janus.html`, 1986.

32 Ian Mackie. The geometry of interaction machine. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL'95*, pages 198–208. ACM Press, 1995. `doi:10.1145/199448.199483`.

33 J.-M. Maranda. Formal categories. *Canadian Journal of Mathematics*, 17:758–801, 1965. `doi:10.4153/CJM-1965-076-0`.

34 Michael A. Nielsen and Isaac L. Chuang. *Quantum Computation and Quantum Information.* Cambridge University Press, 2002.

35 Michele Pagani, Peter Selinger, and Benoît Valiron. Applying quantitative semantics to higher-order quantum computing. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, pages 647–658, New York, NY, USA, 2014. Association for Computing Machinery. `doi:10.1145/2535838.2535879`.

36 Amr Sabry, Benoît 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`.

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

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

39 Tetsuo Yokoyama, Holger Bock Axelsen, and Robert Glück. Fundamentals of reversible flowchart languages. *Theoretical Computer Science*, 611:87–115, 2016. `doi:10.1016/j.tcs.2015.07.046`.

40 Tetsuo Yokoyama and Robert Glück. A reversible programming language and its invertible self-interpreter. In G. Ramalingam and Eelco Visser, editors, *Proceedings of the 2007 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation, PEPM 2007, Nice, France, January 15-16, 2007*, pages 144–153, 2007. `doi:10.1145/1244381.1244404`.