Towards a Curry-Howard Correspondence for Linear, Reversible Computation

Kostia Chardonnet, Alexis Saurin, Benoît Valiron

From the proceedings of Trends in Linear Logic and Applications 2021 (TLLA 2021, eds E. Pimentel, Ch. Retoré)
https://hal.archives-ouvertes.fr/TLLA2021

Towards a Curry-Howard Correspondence for Linear, Reversible Computation

Kostia Chardonnet* Alexis Saurin † Benoît Valiron‡

Abstract

In this paper, we present a linear and reversible language with inductive and coinductive types, together with a Curry-Howard correspondence with the positive fragment of the logic μMALL: linear logic extended with least fixed points allowing inductive statements. Linear, reversible computation makes an important sub-class of quantum computation without measurement. In the latter, the notion of purely quantum recursive type is not yet well understood. Moreover, models for reasoning about quantum algorithms only provide complex types for classical datatypes: there are usually no types for purely quantum objects beside tensors of quantum bits. This work is a first step towards understanding purely quantum recursive types.

1 Introduction

Computation and logic are two faces of the same coin: a 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. Known as the *Curry-Howard correspondence*, this connection between proofs and programs provides a versatile framework. It has been used to mirror first and second-order logics with dependent-type systems, separation logics with memory-aware type systems [Rey02], resource-sensitive logics with differential privacy [GH+13], logics with monads with reasoning on side-effects [SHK+16, MHRM20], etc.

This paper is concerned with the case of reversible computation, a sub-class of *pure* quantum computation (without measurement). In the latter, the notion of purely quantum recursive type is not yet well understood. Moreover, models for reasoning about quantum algorithms only provide complex types for classical datatypes: there are usually no types for purely quantum objects beside tensors of quantum bits. In general quantum computation, one has access to a co-processor holding a "quantum" memory. This memory consists of "quantum" bits having a peculiar property: their state cannot be duplicated, and the operations one can

*Univ. Paris Saclay - LMF, Univ. Paris - IRIF
†Univ. Paris - IRIF, CNRS
‡Univ. Paris Saclay - LMF, CentraleSupélec

perform on them are unitary, reversible operations. The co-processor comes with an interface to which one can send instructions to allocate, update or read quantum registers. Quantum memories can be used to solve classical problems faster than with purely conventional means. Quantum programming languages are nowadays pervasive [FBW18] and several formal approaches based on logical systems have been proposed to relate to this model of computation. However, all of these languages rely on a purely *classical* control-flow: quantum computation is reduced to describing a list of instructions —a quantum circuit— to be sent to the co-processor. While some notion of purely quantum control flow of a program exists, such as quantum conditional expression where the execution of two express becomes itself a superposition of execution, richer notion of quantum control flow does not exists in their more general case, which is the case for recursion. In particular, in this model operations performed on the quantum memory only act on quantum bits and tensors thereof, while the classical computer enjoys the manipulation of any kind of data with the help of rich type systems.

This extended abstract aims at proposing a type system featuring inductive types for a purely reversible language, first step towards a rich quantum type system. We base our study on the approach presented in [SVV18]. 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 $\mathbf{1} \oplus \mathbf{1}$, where $\mathbf{1}$ is a type with only one element. The language in [SVV18] offers the possibility to code isos —reversible maps— with pattern matching. An iso is for instance the swap operation, typed with $A \otimes B \leftrightarrow B \otimes A$. The language also permits higher-order operations on isos, so that an iso can be parametrized by another iso, and is extended with lists (denoted with $[A]$). For instance, one can type a map operation acting on all the elements of a list with $(A \leftrightarrow B) \to ([A] \leftrightarrow [B])$. However, if [SVV18] hints at an extension toward pure quantum computation, the type system is not formally connected to any logical system.

The main contribution of this work is a Curry-Howard correspondence for a purely reversible typed language in the style of [SVV18], with more general recursions, inductive type and better computational expressivity results. However, while higher-order can be consider and do not impact the results, we dismiss it in this extended abstract in order to keep things more intuitive. In particular, in [SVV18] the only recursive type they have is the one of list, which is hard-coded into the language. We capitalize on the logic μMALL [BM07, BDA16]: 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.

$$\frac{}{A \vdash A}\ id \qquad \frac{\Gamma_1, A \vdash \Delta_1 \quad \Gamma_2 \vdash \Delta_2, A}{\Gamma_1, \Gamma_2 \vdash \Delta_1, \Delta_2}\ cut \qquad \frac{\Delta \vdash A}{\Delta, \mathbf{1} \vdash A}\ \mathbf{1}_L$$

$$\frac{}{\vdash \mathbf{1}}\ \mathbf{1}_R \qquad \frac{\Delta, A, B \vdash C}{\Delta, A \otimes B \vdash C}\ \otimes_L \qquad \frac{\Delta \vdash A \quad \Gamma \vdash B}{\Delta, \Gamma \vdash A \otimes B}\ \otimes_R$$

$$\frac{\Delta, A \vdash C \quad \Delta, B \vdash C}{\Delta, A \oplus B \vdash C}\ \oplus_L \qquad \frac{\Delta \vdash A_i}{\Delta \vdash A_1 \oplus A_2}\ \oplus_R^i\ i \in \{1,2\} \qquad \frac{A[X \leftarrow \mu X.A] \vdash B}{\mu X.A \vdash B}\ \mu_L$$

$$\frac{A \vdash B[X \leftarrow \mu X.B]}{A \vdash \mu X.B}\ \mu_R$$

Figure 1: Rules for μMALL.

2 Background on μMALL

The logic μMALL [BM07, BDA16] is an extension of the additive and multiplicative fragment of linear logic [Gir87]. The syntax of linear logic is extended with the formulas $\mu X.A$ and its dual $\nu X.A$ (where X is a type variable occuring in A), which can be understood at the least and greatest fixed points of the operator $X \mapsto A$. These permit inductive and coinductive statements. We are only interested in a fragment of μMALL which contains the tensor, the plus, the unit and the μ connective. Note that our system only deals with closed formulas. The syntax of formulas of μMALL is $A, B ::= \mathbf{1} \mid X \mid A \otimes B \mid A \oplus B \mid \mu X.A$ The derivation rules are shown in Figure 1. They defined a binary relation $\Delta \vdash \Gamma$ on set of formulas defined inductively. For each rule the assumptions are above the line while the conclusion is under. In the rules, the comma stands for the disjoint union: observe that each formula has to be used exactly once and cannot be duplicated or erased. In μMALL one can for instance define the type of natural numbers as $\mu X.\mathbf{1} \oplus X$, of lists of type A as $\mu X.\mathbf{1} \oplus (A \otimes X)$ and of streams of type A as $\nu X.A \otimes X$.

We consider proofs to be potentially non-well-founded derivation trees: they are not necessarily finite as we can for instance consider the formula $\mu X.X$ and apply the rule μ_R an infinite number of times. Among non well-founded proof-objects we distinguish the regular derivation trees that we call circular pre-proofs. These trees can then be represented in a compact manner, see Figure 2. One problem with such a proof-system is to determine whether or not infinite derivations are indeed proofs. Indeed, if every infinite derivation is accepted as a proof, it would be possible to prove any formula F, as shown in Figure 3.

To answer this problem, μMALL comes with a validity criterion for derivations. It roughly says that a derivation is valid if, in every infinite branch of the derivation, there exists an infinite number of rules μ_L. The intuition is that since $\mu X.A$ formulas represent least fixed points, their objects are finite. An infinite number of rule μ_R would mean producing an infinite object, which is not possible. On the other hand, we can explore an arbitrarily large object as input with the rule μ_L.

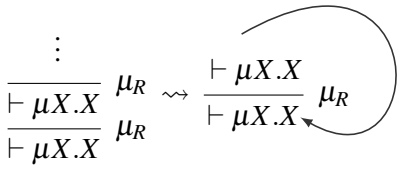

Figure 2: Circular representation of pre-proofs.

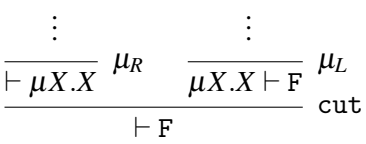

Figure 3: Degenerated proof.

This criterion can be understood in a more operational way as a requirement for productivity.

3 Our language

Our language is based on the one presented in [SVV18]. We build on the reversible part of the paper by extending the language to support both a more general rewriting system and inductive and coinductive types. Terms and types are presented in Table 2, while typing derivations, based on μMALL, can be found in Tables 3 and 4. In the typing rules, Δ is a context of term-variable and their type while Ψ is a context of iso-variable, the semi-colon is just a separator between those two different context. Notice that Ψ is not linear but this is harmless since isos represent close computation. The language consists of the following pieces.

Basic type. They are first-order and typed with base types. The constructors $\texttt{inj}_l$ and $\texttt{inj}_r$ 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 $\otimes$); `fold` represent inductive structure of the types $\mu X.A$. In this works in progress, we do not look at co-inductive type. A value can serve both as a result and as a pattern in the clause of an iso. Generalized patterns are used as special patterns: $\texttt{v}_\texttt{g} : A$ can match any value of type A. Terms are expressions at "surface-level": applying an iso always gives a term, whereas it is an expression only when the argument is a generalized pattern.

Isos. An iso is a function of type $A \leftrightarrow B$, acting on base type, defined as a set of clauses (pair of expressions) of the form $\{e_1 \leftrightarrow e_1' \mid \ldots \mid e_n \leftrightarrow e_n'\}$. The tokens e_i and e_i' in the clauses 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 context Δ_i correspond to the free-variable of the expressions e_i and e_i', they must therefore contain the same free-variable. The `OD` predicate (taken from [SVV18] and not described in this paper) syntactically enforces the exhaustivity and non-overlapping conditions that the left-hand-side and right-hand-side of clauses should satisfy. Exhaustivity for an iso $\{e_1 \leftrightarrow e_1' \mid \ldots \mid e_n \leftrightarrow e_n'\}$ 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 $\texttt{inj}_l\ e$ and $\texttt{inj}_r\ e'$ are non-overlapping while a pattern $\texttt{v}_\texttt{g}$ is always exhaustive. The construc-

(Base types)	$A,B ::=$	$\mathbf{1} \mid A \oplus B \mid A \otimes B \mid \mu X.A$
(Isos)	$\alpha ::=$	$A \leftrightarrow B$
(Values)	$v ::=$	$() \mid x \mid \mathtt{inj}_l\, v \mid \mathtt{inj}_r\, v \mid \langle v_1, v_2 \rangle \mid \mathtt{fold}\, v$
(Generalized pattern)	$\mathtt{v_g} ::=$	$() \mid x \mid \langle \mathtt{v_g}, \mathtt{v_g} \rangle \mid \omega\ \mathtt{v_g} \mid \mathtt{let}\, \mathtt{v_g} = \mathtt{v_g}\, \mathtt{in}\, \mathtt{v_g} \mid \mathtt{fold}\, \mathtt{v_g}$
(Expressions)	$e ::=$	$\mathtt{v_g} \mid \mathtt{inj}_r\, e \mid \mathtt{inj}_l\, e \mid \langle e, e \rangle \mid \mathtt{fold}\, e \mid \mathtt{let}\, \mathtt{v_g} = \mathtt{v_g}\, \mathtt{in}\, e$
(Isos)	$\omega ::=$	$\{e_1 \leftrightarrow e'_1 \mid \ldots \mid e_n \leftrightarrow e'_n\} \mid \mu f.\omega \mid f \mid \mathtt{inv}\, \omega$
(Terms)	$t ::=$	$() \mid x \mid \mathtt{inj}_l\, t \mid \mathtt{inj}_r\, t \mid \langle t_1, t_2 \rangle \mid \mathtt{fold}\, t \mid \omega\, t \mid \mathtt{let}\, \mathtt{v_g} = \mathtt{v_g}\, \mathtt{in}\, t$

Table 2: Terms and types

tion $\mu g.\omega$ represents the creation of a recursive function, rewritten as $\omega[g := \mu g.\omega]$ by the operational semantics. The typing rule for $\mu g.\omega$ has a productivity criterion. Indeed, since we can write non-terminating isos (such as $\mu f.\{x \leftrightarrow f\ x\}$), productivity is important to ensure that we work with total functions. We do not fully details the productivity criterion in this extended abstract, but it simply check that each recursive call is made on a smaller argument, similar as to what is done in the Coq proof assistant. These checks are crucial to make sure that our isos are indeed bijections in the mathematical sense. The construction $\mathtt{inv}\ \omega$ corresponds to the inversion of the iso ω. If ω is of type $A \leftrightarrow B$ then $\mathtt{inv}\ \omega$ is of type $B \leftrightarrow A$. The inversion of an iso of the form $\{e_1 \leftrightarrow e'_1 \mid \ldots \mid e_n \leftrightarrow e'_n\}$ will simply be swapping the expression on each clauses : $\{e'_1 \leftrightarrow e_1 \mid \ldots \mid e'_n \leftrightarrow e_n\}$.

Finally, our language is equipped with a rewrite system ($\rightarrow$) on terms. The evaluation of an iso applied to an argument works with pattern-matching. The non-overlapping and exhaustivity conditions guarantee subject-reduction (see Prop. 3).

Example 1. *Encoding of the isomorphism map(ω) in our language, where $[]$ is the empty list and $::$ is the list construction. The iso map(ω) is of type $([A] \leftrightarrow [B])$ where $[A]$ is the type of lists of type A. This iso is parametrize with an iso of type $A \leftrightarrow B$ which is applied to each element of the list given as argument:*

$$\mu g. \left\{ \begin{array}{ll} [\,] & \leftrightarrow [\,] \\ h :: t & \leftrightarrow (\omega\ h) :: (g\ t) \end{array} \right\} : ([A] \leftrightarrow [B]).$$

Remark 2. In our example, the left and right-hand side of the $\leftrightarrow$ respect both the criteria of exhaustivity —every-value of each type is being covered by at least one expression— and non-overlapping —no two expressions cover the same value.

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

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

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

$$\frac{\Gamma;\Psi \vdash_e \mathtt{v}_{\mathtt{g}_1} : A \qquad \Delta_1;\Psi \vdash_e \mathtt{v}_{\mathtt{g}_2} : A \qquad \Gamma,\Delta_2;\Psi \vdash_e t : B}{\Delta_1,\Delta_2;\Psi \vdash_e \mathtt{let}\, \mathtt{v}_{\mathtt{g}_1} = \mathtt{v}_{\mathtt{g}_2}\, \mathtt{in}\, t : B}$$

Table 3: Typing of terms and expressions

$$\frac{\begin{array}{cccc}\Delta_1;\Psi \vdash_e e_1 : A & \ldots & \Delta_n;\Psi \vdash_e e_n : A & \mathtt{OD}_A\{e_1,\ldots,e_n\} \\ \Delta_1;\Psi \vdash_e e'_1 : B & \ldots & \Delta_n;\Psi \vdash_e e'_n : B & \mathtt{OD}_B\{e'_1,\ldots,e'_n\}\end{array}}{\Psi \vdash_\omega \{e_1 \leftrightarrow e'_1 \mid \ldots \mid e_n \leftrightarrow e'_n\} : A \leftrightarrow B.}$$

$$\frac{\Psi \vdash_\omega \omega : T^\perp}{\Psi \vdash_\omega \mathtt{inv}\, \omega : T} \qquad \frac{}{\Psi, f : \alpha \vdash_\omega f : \alpha} \quad \frac{}{\Psi, f : \alpha^\perp \vdash_\omega f : \alpha}$$

$$\frac{\Psi, f : A \leftrightarrow B \vdash_\omega \omega : A \leftrightarrow B \qquad \mu f.\omega \text{ is productive}}{\Psi \vdash_\omega \mu f.\omega : A \leftrightarrow B}$$

Table 4: Typing of isos

The iso is therefore a bijection. In order to achieve higher-order, we need the λ-abstractions to be applied, otherwise we can't guarantee productivity.

Proposition 3. *The language features subject reduction: If $\vdash t : A$ and $t \to t'$ then we have $\vdash t' : A$. Moreover, it enjoys confluence: Let $\to^*$ be the reflexive, transitive closure of $\to$. If $t \to^* t_1$ and $t \to^* t_2$ then there exists t_3 such that $t_1 \to^* t_3$ and $t_2 \to^* t_3$.* □

Subject reduction is shown by a simple induction on the reduction while confluence is shown by using a parallel rewrite system.

Every well-typed isos is indeed an isomorphism. This is shown by induction: first on the length of the reduction, then on ω.

Theorem 4. *For all $\omega : A \leftrightarrow B$, $v : A$ and $u : B$ then $((\mathtt{inv}\, \omega) \circ \omega)\ v \to^* v$ and $(\omega \circ \mathtt{inv}\, \omega)\ u \to^* u$.*

We can encode any Primitive Recursive Function, using the language RPP of Recursive Primitive Permutation [LM17]:

Theorem 5. *Let f be a Primitive Recursion Function, there exists well-typed iso ω that simulate it.*

$$\dfrac{\dfrac{\dfrac{}{\vdash \mathbf{1}}\,\mathbf{1}_R}{\vdash \mathbf{1}\oplus(A\otimes[A])}\,\oplus_R^1}{\vdash [A]}\,\mu_R \qquad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\pi_\omega}{A\vdash A}\quad\dfrac{}{A\vdash A}\,\mathrm{id}}{A\vdash A}\,\mathtt{cut}\quad\dfrac{\dfrac{\dagger}{[A]\vdash[A]}\quad\dfrac{}{[A]\vdash[A]}\,\mathrm{id}}{[A]\vdash[A]}\,\mathtt{cut}}{A,[A]\vdash A\otimes[A]}\,\otimes_R}{A,[A]\vdash \mathbf{1}\oplus(A\otimes[A])}\,\oplus_R^2}{A,[A]\vdash[A]}\,\mu_R}{A\otimes[A]\vdash[A]}\,\otimes_L$$

$$\dfrac{\mathbf{1}\vdash[A] \qquad A\otimes[A]\vdash[A]}{\mathbf{1}\oplus(A\otimes[A])\vdash[A]}\,\oplus_L \qquad \dfrac{\mathbf{1}\oplus(A\otimes[A])\vdash[A]}{[A]\vdash^\dagger[A]}\,\mu_L$$

Figure 4: Proof corresponding to Example 7 where the † represent the loop.

4 Towards Curry-Howard

An iso $\vdash \omega : A \leftrightarrow B$ corresponds to both a computation sending a value of type A to a result of type B and a computation sending a value of type B to a result of type A. We can mechanically translate such an iso to a pair of derivations $\pi, \pi^\perp$ in μMALL, where π is a proof of $A \vdash B$ and $\pi^\perp$ is a proof of $B \vdash A$. This mechanical translation constructs circular pre-proofs, as discussed in Section 2. We however still need to show that the obtained derivations respect the validity criterion for circular proof.

Once proven, we would obtain a *static* correspondence between programs and proofs. We would however still need to show that this entails a *dynamic* correspondence between the evaluation procedure of our language and the cut-elimination procedure of μMALL. For that, we would need to make sure 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.

Conjecture 6. Validity of proofs. *If $\vdash \omega : A \leftrightarrow B$ then the μMALL derivations $\pi : A \vdash B$ and $\pi^\perp : B \vdash A$ of ω are valid.*
Isomorphism of proofs. *Provided that the above holds, we moreover have*

$$\dfrac{}{A\vdash A}\,id \;{}_{*}\!\!\leftsquigarrow\; \dfrac{\dfrac{\pi^\perp}{B\vdash A}\quad\dfrac{\pi}{A\vdash B}}{A\vdash A}\,\mathtt{cut} \qquad \dfrac{\dfrac{\pi}{A\vdash B}\quad\dfrac{\pi^\perp}{B\vdash A}}{B\vdash B}\,\mathtt{cut} \;\leadsto^*\; \dfrac{}{B\vdash B}\,id$$

Simulation of evaluation. *Provided that t is a value and v is a normal form, if $\omega\, t \to^* v$, if π is the proof corresponding to $\omega\, t$, and if π' is the proof corresponding to v, then $\pi \to^* \pi'$ with the cut-elimination procedure.*

The difficulty lies in the fact that the μMALL validity criterion is not complete: not every productive derivation are valid one. This make the relation between the productivity of terms and the validity of derivations non-trivial.

Example 7. *Consider the iso* map(ω) *from Example 1 and a proof π_ω which correspond to the parametrize iso ω. The derivation obtain from* map(ω) *is shown in Figure 4, notice that this derivation is indeed valid according to μMALL criterion.*

5 Conclusion and Future Work

We presented a first-order, linear, reversible language with inductive types together with an interpretation of programs into derivations in the logic μMALL. This work is still in progress: We still need to show that the produced μMALL derivations are indeed proofs and that they are isomorphisms. After completing the proofs of our current conjectures, we want to extend our language to linear combinations of terms and coinductive constructions in order to study purely quantum recursive types and generalized quantum loops: in [SVV18], lists are the only recursive type which is captured and recursion is terminating. The logic μMALL would help providing a finer understanding of termination and non-termination, which could in return helps us in defining notions of recursion in pure quantum computation.

References

[BDA16] Baelde, D., Doumane, A., Saurin, A.: Infinitary proof theory: the multiplicative additive case. In: Proc. of CSL. LIPIcs, vol. 62, pp. 42:1–42:17 (2016)

[BM07] Baelde, D., Miller, D.: Least and greatest fixed points in linear logic. In: Proc. of LPAR. LNCS, vol. 4790, pp. 92–106. Springer (2007).

[FBW18] Fingerhuth, M., Babej, T., Wittek, P.: Open source software in quantum computing. PLOS ONE **13**(12), 1–28 (2018).

[GH+13] Gaboardi, M., Haeberlen, *et al.*: Linear dependent types for differential privacy. In: Proc. of POPL. pp. 357–370. ACM (2013).

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

[JJ+18] Jung, R., Jourdan, *et al.*: RustBelt: securing the foundations of the Rust programming language. PACMPL **2**(POPL), 66:1–66:34 (2018).

[MHRM20] Maillard, K., Hritcu, C., Rivas, E., Muylder, A.V.: The next 700 relational program logics. PACMPL **4**(POPL), 4:1–4:33 (2020).

[Rey02] Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: Proc. LICS. pp. 55–74. IEEE Computer Society (2002).

[SVV18] Sabry, A., Valiron, B., Vizzotto, J.K.: From symmetric pattern-matching to quantum control. In: Proc. FoSSACS. LNCS 10803, pp. 348-364. Springer (2018).

[SHK+16] Swamy, N., Hritcu, C., Keller, C., *et al.*: Dependent types and multi-monadic effects in F. In: Proc. POPL. pp. 256–270. ACM (2016).

[BN99] Baader, Franz and Nipkow, Tobias: Term rewriting and all that (1999) Cambridge university press

[LM17] Luca Paolini and Mauro Piccolo and Luca Roversi: A class of Recursive Permutations which is Primitive Recursive complete