Toward a Curry-Howard Equivalence for Linear, Reversible Computation

Kostia Chardonnet, Alexis Saurin, Benoît Valiron

Toward a Curry-Howard Equivalence for Linear, Reversible Computation

Work-in-progress

Kostia Chardonnet[1,2,3], Alexis Saurin[2,3], and Benoît Valiron[1,2]

[1] Université Paris-Saclay, CNRS, CentraleSupélec, Laboratoire de Recherche en Informatique, 91405, Orsay, France.
[2] Université de Paris, IRIF, CNRS, 75013, Paris, France.
[3] Équipe πr^2, Inria

Abstract. In this paper, we present a linear and reversible language with inductive and coinductive types, together with a Curry-Howard correspondence with the logic μMALL: linear logic extended with least and greatest fixed points allowing inductive and coinductive 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.

Keywords: Reversible computation · Linear logic · Curry-Howard

1 Introduction

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

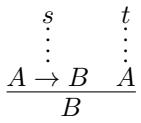

Fig. 1. Modus-Ponens

This connection is known as the *Curry-Howard correspondence* [4,8]. In this general framework, types correspond to formulas and programs to proofs, while program evaluation is mirrored with proof simplification (the so-called cut-elimination). The Curry-Howard correspondence formalizes the fact that the proof s of $A \to B$ can be regarded as a *function* —parametrized by an argument of type A— that produces a proof of B whenever it is fed with a proof of A. Therefore, the computational interpretation of Modus-Ponens corresponds to the *application* of an argument (i.e. t) of type

A to a function (i.e. s) of type $A \to B$. When computing the corresponding program, one substitutes the parameter of the function with t and get a result of type B. On the logical side, this corresponds to substituting every axiom introducing A in the proof s with the full proof t of A. This yields a direct proof of B without any invocation of the "lemma" $A \to B$.

Paving the way toward the verification of critical softwares, the Curry-Howard correspondence provides a versatile framework. It has been used to mirror first and second-order logics with dependent-type systems [3,10], separation logics with memory-aware type systems [13,9], resource-sensitive logics with differential privacy [6], logics with monads with reasoning on side-effects [17,11], etc.

This paper is concerned with the case of reversible computation, a sub-class of *pure* quantum computation. 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 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 [5] and several formal approaches based on logical systems have been proposed to relate to this model of computation [16,12,14]. 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. 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 and coinductive types for a purely reversible language, first step towards a rich quantum type system. We base our study on the approach presented in [15]. In this model, reversible computation is restricted to two main types: the tensor, written $a \otimes b$ and the co-product, written $a \oplus b$. The former corresponds to the type of all pairs of elements of type a and elements of type b, while the latter represents the disjoint union of all elements of type a and elements of type b. For instance, a bit can be typed with $\mathbb{1} \oplus \mathbb{1}$, where $\mathbb{1}$ is a type with only one element. The language in [15] 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 [15] 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 [15]. We capitalize on the logic μMALL [2,1]: an extension of the additive and multiplicative fragment of linear logic with least and greatest fixed points allowing inductive and coinductive statements. This logic con-

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

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

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

$$\frac{A \vdash B[X \leftarrow \mu X.B]}{A \vdash \mu X.B}\ \mu_R \qquad \frac{A[X \leftarrow \nu X.A] \vdash B}{\nu X.A \vdash B}\ \nu_L \qquad \frac{A \vdash B[X \leftarrow \nu X.B]}{A \vdash \nu X.B}\ \nu_R$$

Fig. 2. Rules for μMALL.

tains both a tensor and a co-product, and its strict linearity makes it a good fit for a reversible type system.

2 Background on μMALL

The logic μMALL [2,1] is an extension of the additive and multiplicative fragment of linear logic [7]. 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 and ν connectives. Note that our system only deals with closed formulas. Our syntax of formulas is $A, B ::= \mathbb{1} \mid X \mid A \otimes B \mid A \oplus B \mid \mu X.A \mid \nu X.A$. The derivation rules are shown in Figure 2. 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 $\mu X.\mathbb{1} \oplus X$, of lists of type A as $\mu X.\mathbb{1} \oplus (A \otimes X)$ 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 3. 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 4.

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 or an infinite number of rules ν_R. The intuition

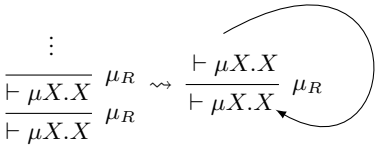

Fig. 3. Circular representation of proofs.

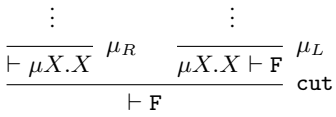

Fig. 4. Degenerated proof.

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. For the other case, since $\nu X.A$ formulas represent greatest fixed points, their object are infinite. We therefore want to ensure that we can produce infinite objects: hence the infinite number of rules ν_R. 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 [15]. 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. The language is defined by layers. Terms and types are presented in Table 2, while typing derivations, based on μMALL, can be found in Tables 3 and 4. 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` and `pack` respectively represent inductive and coinductive structure of for the types $\mu X.A$ and $\nu X.A$. 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.

First-order isos. An iso of type α acts on terms of base types. An iso is a function of type $A \leftrightarrow B$, defined as a set of clauses of the form $\{e_1 \leftrightarrow e'_1 \mid \ldots \mid e_n \leftrightarrow e'_n\}$. The tokens e_i and e'_i in the clauses are expressions. Compared to the original language in [15], we allow general expressions both on the left and on the right of a clause. 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 `OD` predicate (taken from [15] 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.

$$
\begin{array}{lll}
\text{(Base types)} & A, B ::= & \mathbb{1} \mid A \oplus B \mid A \otimes B \mid \mu X.A \mid \nu X.A \\
\text{(Isos, first-order)} & \alpha ::= & A \leftrightarrow B \\
\text{(Isos, higher-order)} & T ::= & \alpha_1 \to \cdots \to \alpha_n \to \alpha \\
\text{(Values)} & v ::= & () \mid x \mid \mathtt{inj}_l\ v \mid \mathtt{inj}_r\ v \mid \langle v_1, v_2 \rangle \mid \\
 & & \quad \mathtt{fold}\ v \mid \mathtt{pack}\ v \\
\text{(Generalized pattern)} & \mathtt{v_g} ::= & () \mid x \mid \langle \mathtt{v_g}, \mathtt{v_g} \rangle \mid \omega\ \mathtt{v_g} \mid \mathtt{let}\ \mathtt{v_g} = \mathtt{v_g}\ \mathtt{in}\ \mathtt{v_g} \mid \\
 & & \quad \mathtt{fold}\ \mathtt{v_g} \mid \mathtt{pack}\ \mathtt{v_g} \\
\text{(Expressions)} & e ::= & \mathtt{v_g} \mid \mathtt{inj}_r\ e \mid \mathtt{inj}_l\ e \mid \langle e, e \rangle \mid \\
 & & \quad \mathtt{fold}\ e \mid \mathtt{pack}\ e \mid \mathtt{let}\ \mathtt{v_g} = \mathtt{v_g}\ \mathtt{in}\ e \\
\text{(Isos)} & \omega ::= & \{e_1 \leftrightarrow e_1' \mid \ldots \mid e_n \leftrightarrow e_n'\} \mid \lambda f.\omega \mid \\
 & & \quad \mu f.\omega \mid f \mid \omega_1\, \omega_2 \mid \mathtt{inv}\ \omega \\
\text{(Terms)} & t ::= & () \mid x \mid \mathtt{inj}_l\ t \mid \mathtt{inj}_r\ t \mid \langle t_1, t_2 \rangle \mid \\
 & & \quad \mathtt{fold}\ t \mid \mathtt{pack}\ t \mid \omega\ t \mid \mathtt{let}\ \mathtt{v_g} = \mathtt{v_g}\ \mathtt{in}\ t
\end{array}
$$

Table 2. Terms and types

Higher-order isos. An iso of type T manipulate other isos as basic blocks. Since isos represent closed computations, iso-variable are non-linear and can be duplicated at will while term-variable are linear. The constructions $\lambda f.\omega$ and $\omega_1\ \omega_2$ represent respectively the abstraction of a function and the application of an iso to another. The construction $\mu g.\omega$ 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 isos can be non-terminating (because of coinduction), productivity is important to ensure that we work with total functions. 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 $B \leftrightarrow A$.

Finally, our language is equipped with a rewrite system ($\to$) 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. 3.1).

Example 3.1. Encoding of the isomorphism *map* in our language, where $[\,]$ is the empty list and :: is the list construction. The iso *map* is of type $(A \leftrightarrow B) \to ([A] \leftrightarrow [B])$ where $[A]$ is the type of lists of type A. This iso takes an iso of type $A \leftrightarrow B$ as argument and apply it to each element of the list given as argument:

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

Example 3.2. We can define the iso of type : $A \oplus (B \oplus C) \leftrightarrow C \oplus (A \oplus B)$ as

$$\left\{ \begin{array}{ll} \mathtt{inj}_l\ a & \leftrightarrow \mathtt{inj}_r\ \mathtt{inj}_l\ a \\ \mathtt{inj}_r\ \mathtt{inj}_l\ b & \leftrightarrow \mathtt{inj}_r\ \mathtt{inj}_r\ b \\ \mathtt{inj}_r\ \mathtt{inj}_r\ c & \leftrightarrow \mathtt{inj}_l\ c \end{array} \right\}.$$

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

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

$$\frac{\Psi \vdash_\omega \omega : A \leftrightarrow B \quad \Delta;\Psi \vdash_e t : A}{\Delta;\Psi \vdash_e \omega\ t : B} \quad \frac{\Delta;\Psi \vdash_e t : A[X \leftarrow \mu X.A]}{\Delta;\Psi \vdash_e \mathtt{fold}\ t : \mu X.A}$$

$$\frac{\Gamma;\Psi \vdash_e \mathtt{v}_{g_1} : A \qquad \Delta_1;\Psi \vdash_e \mathtt{v}_{g_2} : A \qquad \Gamma,\Delta_2;\Psi \vdash_e t : B}{\Delta_1,\Delta_2;\Psi \vdash_e \mathtt{let}\ \mathtt{v}_{g_1} = \mathtt{v}_{g_2}\ \mathtt{in}\ t : B}$$

Table 3. Typing of terms and expressions

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

$$\frac{\Psi, f : \alpha \vdash_\omega \omega : T}{\Psi \vdash_\omega \lambda f.\omega : \alpha \to T} \qquad \frac{}{\Psi, f : \alpha \vdash_\omega f : \alpha} \qquad \frac{\Psi \vdash_\omega \omega_1 : \alpha \to T \quad \Psi \vdash_\omega \omega_2 : \alpha}{\Psi \vdash_\omega \omega_1\omega_2 : T}$$

$$\frac{\Psi \vdash_\omega \omega : T^\perp}{\Psi \vdash_\omega \mathtt{inv}\ \omega : T} \qquad \frac{\Psi, f : \alpha \vdash_\omega \omega : \alpha_1 \to \dots \to \alpha_n \to \alpha \qquad \mu f.\omega \text{ is productive}}{\Psi \vdash_\omega \mu f.\omega : \alpha_1 \to \dots \to \alpha_n \to \alpha}$$

Table 4. Typing of isos

Remark 3.1. In our two examples, the left and right-hand side of the $\leftrightarrow$ on each function respect both the criteria of exhaustivity —every-value of each type is being covered by at least one expression— and non-overlapping —no two expressions cover the same value. Both isos are therefore bijections.

Property 3.1. *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 $t_2 \to^* t_3$.* □

We conjecture that well-typed isos are indeed isomorphisms:

Conjecture 3.1. *For all $\omega : A \leftrightarrow B$, $v : A$ and $u : B$ then $((\mathtt{inv}\ \omega) \circ \omega)\ v \to^* v$ and $(\omega \circ \mathtt{inv}\ \omega)\ u \to^* u$.*

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.

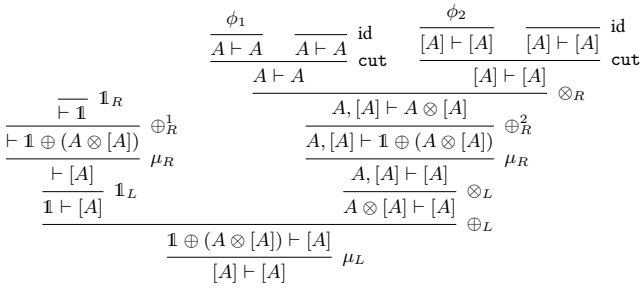

Fig. 5. Proof corresponding to example 4.1.

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 4.1. 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*

$$\frac{}{A \vdash A}\ id \quad \rightsquigarrow \quad \frac{\dfrac{\pi^{\perp}}{B \vdash A} \quad \dfrac{\pi}{A \vdash B}}{A \vdash A}\ \texttt{cut} \qquad \frac{\dfrac{\pi}{A \vdash B} \quad \dfrac{\pi^{\perp}}{B \vdash A}}{B \vdash B}\ \texttt{cut} \quad \rightsquigarrow \quad \frac{}{B \vdash B}\ id$$

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 ω t, and if π' is the proof corresponding to v, then $\pi \to^* \pi'$ with the cut-elimination procedure.*

Example 4.1. Consider the iso that, given an iso f and a list $[x_1, x_2, \ldots, x_n]$ returns the list $[f\ x_1, (\texttt{inv}\ f)\ x_2, f\ x_3, (\texttt{inv}\ f)\ x_4, \ldots]$ written as:

$$\mu g.\lambda f. \left\{ \begin{array}{l} [\,] \leftrightarrow [\,] \\ h :: t \leftrightarrow (f\ h) :: ((g\ (\texttt{inv}\ f)\ )t) \end{array} \right\} : (A \leftrightarrow A) \to ([A] \leftrightarrow [A]) \tag{1}$$

We define the two mutually recursive proofs π_1 and π_2 by $\pi_1 = \Pi(\psi_f, \pi_2)$ and $\pi_2 = \Pi(\psi_{f^{\perp}}, \pi_1)$ where ψ_f and $\psi_{f^{\perp}}$ correspond to the isos f and $\texttt{inv}\ f$. The proof associated with the iso in Eq. (1) is π_1. The proof $\Pi(\phi_1, \phi_2)$ is shown in Figure 5.

5 Conclusion

We presented a higher-order, linear, reversible language with inductive and coinductive types together with an interpretation of programs into derivations in the logic μMALL. This work is still in progress: A number of proofs still need to be completed.

After completing the proofs of our current conjectures, we want to extend our language to linear combinations of terms in order to study purely quantum recursive types and generalized quantum loops: in [15], 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.

Acknowledgments. This work was supported in part by the French National Research Agency (ANR) under the research project SoftQPRO ANR-17-CE25-0009-02, and by the DGE of the French Ministry of Industry under the research project PIA-GDN/QuantEx P163746-484124.

References

1. 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)
2. Baelde, D., Miller, D.: Least and greatest fixed points in linear logic. In: Proc. of LPAR. LNCS, vol. 4790, pp. 92–106. Springer (2007). https://doi.org/10.1007/978-3-540-75560-9_9
3. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development - Coq'Art. Springer (2004). https://doi.org/10.1007/978-3-662-07964-5
4. Curry, H.B.: Functionality in combinatory logic. Proceedings of the National Academy of Sciences of the United States of America **20**(11), 584 (1934)
5. Fingerhuth, M., Babej, T., Wittek, P.: Open source software in quantum computing. PLOS ONE **13**(12), 1–28 (2018). https://doi.org/10.1371/journal.pone.0208561
6. Gaboardi, M., Haeberlen, *et al.*: Linear dependent types for differential privacy. In: Proc. of POPL. pp. 357–370. ACM (2013). https://doi.org/10.1145/2429069.2429113
7. Girard, J.Y.: Linear logic. Theoretical computer science **50**(1), 1–101 (1987)
8. Howard, W.A.: The formulae-as-types notion of construction. To HB Curry: essays on combinatory logic, lambda calculus and formalism **44**, 479–490 (1980)
9. Jung, R., Jourdan, *et al.*: RustBelt: securing the foundations of the Rust programming language. PACMPL **2**(POPL), 66:1–66:34 (2018). https://doi.org/10.1145/3158154
10. Leroy, X.: Formal verification of a realistic compiler. Commun. ACM **52**(7), 107–115 (2009). https://doi.org/10.1145/1538788.1538814
11. Maillard, K., Hritcu, C., Rivas, E., Muylder, A.V.: The next 700 relational program logics. PACMPL **4**(POPL), 4:1–4:33 (2020). https://doi.org/10.1145/3371072
12. Paykin, J., Rand, R., Zdancewic, S.: QWIRE: a core language for quantum circuits. In: Proc. POPL. pp. 846–858. ACM (2017). https://doi.org/10.1145/3009837.3009894
13. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: Proc. LICS. pp. 55–74. IEEE Computer Society (2002). https://doi.org/10.1109/LICS.2002.1029817
14. Rios, F., Selinger, P.: A categorical model for a quantum circuit description language. In: Prof. QPL. ENTCS, vol. 266, pp. 164–178 (2017). https://doi.org/10.4204/EPTCS.266.11
15. Sabry, A., Valiron, B., Vizzotto, J.K.: From symmetric pattern-matching to quantum control. In: Proc. FoSSACS. LNCS 10803, pp. 348-364. Springer (2018). https://doi.org/10.1007/978-3-319-89366-2_19
16. Selinger, P., Valiron, B.: A lambda calculus for quantum computation with classical control. Mathematical Structures in Computer Science **16**(3), 527–552 (2006)
17. Swamy, N., Hritcu, C., Keller, C., *et al.*: Dependent types and multi-monadic effects in F. In: Proc. POPL. pp. 256–270. ACM (2016). https://doi.org/10.1145/2837614.2837655