Taming Non-Compositionality Using New Binders

F. Prost

LIG
46, av Félix Viallet, F-38031 Grenoble, France
Frederic.Prost@imag.fr

Abstract. We propose an extension of the traditional λ-calculus in which terms are used to control an outside computing device (quantum computer, DNA computer...). We introduce two new binders: ν and ρ. In $\nu x.M$, x denotes an abstract resource of the outside computing device, whereas in $\rho x.M$, x denotes a concrete resource. These two binders have different properties (in terms of α-conversion, scope extrusion, convertibility) than the ones of standard λ-binder. We illustrate the potential benefits of our approach with a study of a quantum computing language in which these new binders prove meaningful. We introduce a typing system for this quantum computing framework in which linearity is only required for concrete quantum bits offering a greater expressiveness than previous propositions.

1 Introduction

λ-calculus [8] is a particularly well suited formalism to study functional programs. Any λ-variable can be replaced by any λ-term in the pure λ-calculus. λ-abstraction can be seen in the following way: the term $\lambda x.M$ represents an algorithm M in which x denotes another completely abstracted algorithm, thus x may be seen as a black box. This approach is deeply compositional. The "meaning" of M can be precisely defined with relation to the "meaning" of x. This inner compositionality of λ-calculus is reflected by the success of type systems for λ-caculi [9]. Those type systems can be used with good effect to statically analyze programs (e.g. [2, 20, 19]).

Some λ-calculus variants have been developed to study functional programing languages including imperative features like ML (e.g. [3, 14, 18]). The idea is to introduce a global state and to refer to it via variable names. Name management is handled via standard λ-abstraction mechanism. Typically in ML-like languages [16] one can define a variable of type `int` by `P` $\stackrel{def}{=}$ `let x=ref 0 in M` which is syntactic sugar for $(\lambda x.M \;\; (\texttt{new}\ 0))$, where `new` is a side-effect function that adds a new cell referred by x and whose value is 0 in the global state. α-conversion of the λ-abstraction insures hygienic properties of such computing models, it handles the fresh-name generation problem. For instance consider:

$$(\lambda y.\langle y, y\rangle \;\; P)$$

where P is duplicated. The actual evaluation of this term will have to handle the fact that x cannot refer to two different cells. From a a theoretical point of view this problem is handled by α-conversion.

Introducing a global state enhances the expressiveness of the language but it has a cost: compositionality is lost. Indeed, side effects on the global state lead to non compositional behavior of programs. Consider the following program:

$$Q \stackrel{def}{=} \lambda y.(x :=!x + 1; y :=!y + 1)$$

where $!x$ stands for the content of an imperative cell x. It is clear that $(Q\ x)$ increments the content of x by 2 whereas $(Q\ z)$, provided that $x \neq z$, increments the content of x by 1. The problems get more complicated if you consider a language in which variables can be references, and thus with potential aliasing of variables: if z is aliased to x (but is syntacticly different from x) then $(Q\ z)$ increments x by 2. Static analysis of such programs is not an easy task, it requires sophisticated approach very different from usual typing systems (see [11]).

Things can get even more complicated when the global state is not classical. If you consider quantum computing [17], the global state is an array of quantum bits. Several functional programming language have been proposed for quantum computation (e.g. [22, 21, 6]). In a quantum functional programing language, quantum bits can be entangled by side effects that go beyond the lexical scope of their definitions. and thus such programing languages exhibit a highly-non compositional behavior. Consider the program

$$Q \stackrel{def}{=} \lambda x, y.(\mathfrak{Cnot} \quad < x, y >)$$

Q executes the conditional not on two quantum bits x, y. Therefore, $(P\ q_1\ q_2)$ may entangle qbits q_1 and q_2. But if q_1 is entangled to q_0, and q_2 is entangled to q_3, then $(P\ q_1\ q_2)$ also entangles q_0 to q_3 ! This simplistic example shows that the classical binder, λ, is not well suited to deal with quantum bits and more generally with global references.

In this paper we propose to extend the λ-calculus by adding two new binders: ν and ρ. These binders exhibit different fundamental properties than the ones of the λ binder. We show that these binders are more adapted for the design of a functional programming language including a global state. νx declares a new resource that can be used. It is very close to the ν of the π-calculus [15] (hence the notation), but has not the scope extrusion property. In $\nu x.M$, x is an *a*bstract reference: x refers actually to nothing in the global state. On the other hand ρ binder defines a *c*oncrete reference to something actual in the global state. Because of possible side effects, ρ has the full scope extrusion property, indeed operations on a local variable may have implications outside the scope of its definition as explained earlier.

In section 2 we introduce λ_{GS}, a λ-calculus based framework to deal with global states. We define a specific λ_{GS} calculus for quantum computation: λ^{Q}_{GS} in section 3. We give a type system for λ^{Q}_{GS} allowing greater programming flexibility

than previous propositions of the literature. Finally, we conclude and discuss future works in section 4.

Knowledge of basics of quantum computation is supposed. We refer to [7, 1] for quantum computing tutorials.

2 Global State Calculus

2.1 λ_{GS} ideas

A λ_{GS} calculus is defined relatively to an outside computing device on which computations/actions can be performed (such as a quantum state, DNA computer etc.). Those actions controlled via λ_{GS} terms. λ_{GS} calculus may be viewed as a system providing a functional control of a computing device. Thus a program in λ_{GS} calculus has two parts: a hard one (the computing device viewed as a global state which is a physical artifact used to compute) and a soft one (the functional term that controls the computing process).

The functional part of λ_{GS} is implemented by standard λ-calculus terms with extra features to deal with the global state. Interactions with the global state are handled by two mechanisms:

1. Locations: from the term point of view, locations are just names. From the global state point of view, locations are the addresses of physical entities (the actual address of information stored in RAM, a specific quantum bit etc.).
2. Actions: from the term point of view, actions are terms that can be evaluated. The evaluation depend on the global state as well as on the parameters of the action (for instance the result of the measurement of a quantum bit). From the global state point of view, actions perform actual modifications of it (update of a RAM cell, application of a unary gate to quantum bits).

2.2 λ_{GS} definition

A λ_{GS} framework is defined up to the choice of a global state and the according actions.

Terms

Definition 1 (Terms). *The set of terms $\mathcal{T}$ is inductively defined as follows:*

$$\begin{aligned} M, N, P ::= & \; x \mid \ell \mid \lambda x.M \mid (M \quad N) \\ & \mid \langle M, N \rangle \mid \mathsf{let}\ \langle x, y \rangle = M\ \mathsf{in}\ N \\ & \mid \nu x.M \mid \rho \ell.M \mid !M \\ & \mid \mathsf{a}_i(M) \end{aligned}$$

where x, ℓ are respectively variable and location names defined over a countable set $\mathcal{V}$. $\mathcal{L}$ is the subset of $\mathcal{V}$ denoting locations.

There is no formal distinction between x and ℓ. However we use the following convention: ℓ is used to denote a location (that is a reference to the outside global state) whereas x denotes a standard λ-variable.
Contexts are defined as usual : a context $C[\cdot]$ is a term where $\cdot \in \mathcal{V}$ occurs exactly once. By $C[M]$ we denote the syntactical replacement of $\cdot$ with M (with name capture).

The first two lines of definition 1 define standard pure λ-terms with tuples ($\overrightarrow{M}$ denotes tuples of terms : $\overrightarrow{M} \stackrel{def}{=} \langle M_1, \langle M_2, \langle \ldots, M_n \rangle \ldots \rangle \rangle$). Tuples are native in λ_{GS} because of actions, indeed actions cannot be curryfied: they are not functions in a λ-calculus sense.

In λ_{GS}, there are three binders λ, ν and ρ. Thus, we have the following definition of free variables.

Definition 2 (Free variables). *Free variables are defined by:*

$$FV(x) = \{x\} \qquad FV(!M) = FV(\mathsf{a}_i(M)) = FV(M)$$

$$FV(\lambda x.M) = FV(\nu x.M) = FV(\rho x.M) = FV(M) \setminus \{x\}$$

$$FV((M \quad N)) = FV(\langle M, N \rangle) = FV(M) \cup FV(N)$$

$$FV(\mathsf{let}\ \langle x, y \rangle = M\ \mathsf{in}\ N) = (FV(M) \cup FV(N)) \setminus \{x, y\}$$

In $\nu x.M$, x is an abstract fresh location and is bound in M. The idea, is that $\nu x.M$ is a piece of code that will use a global state information referred by x. On the other hand $\rho\ell.M$ denotes a piece of code that *actually* uses a concrete global state information. ℓ is bound in M in $\rho\ell.M$. As usual we write $\rho\ell_1, \ldots, \ell_n.M$ as shorthand for $\rho\ell_1.\rho\ell_2. \ldots \rho\ell_n.M$.

$!M$ is the *realization* action. It transforms an abstract location into a concrete one. Intuitively (formal definition follows) we have the reduction:

$$C[!\nu x.M] \to C[\rho\ell.M[x := \ell]]$$

where ℓ is a fresh location relatively to $FV(C[!\nu x.M])$. The idea is that realization has a side effect on the global state: it actually creates a new resource.

a_i are actions and are relative to the global state. For example, if the global state stores terms actions are affectation and pointer dereferencing. If the state is made of quantum bits, then typical actions include unary gates (e.g. phase, Conditional Not, Hadamard) and measurement.

Equivalences We now define several equivalence relations over terms. Those relations exhibit differences between the three λ_{GS} binders.

Definition 3 (Term equivalence). *We have the following equivalences between terms:*

– *λ and ν binders generate α-equivalence:*

$$\lambda x.M =_\alpha \lambda z.M[x := z]$$
$$\nu x.M =_\alpha \nu z.M[x := z]$$

with $z \notin FV(M)$

– *ν and ρ binders generate commutation equivalence:*

$$\rho\ell.\rho\ell'.M =_\xi \rho\ell'.\rho\ell.M$$
$$\nu x.\nu y.M =_\xi \nu y.\nu x.M$$

– *ρ binder generates scope extrusion equivalence:*

$$C[\rho\ell.M] =_\sigma \rho\ell.C[M]$$

for any context $C[\cdot]$.

– *ρ binder generates garbage collection equivalence:*

$$\rho o.M =_\gamma M \; o \notin M$$

Let $\div$ be the union of $=_\alpha$, $=_\sigma$, $=_\xi$ and $=_\gamma$.

Scope extrusion is the main λ_{GS} mechanism to handle non-compositionality. Indeed, ℓ in $\rho\ell.M$ refers to some outside computing device, thus any action on ℓ can have side effects that go beyond the lexical scope of $\rho\ell.M$. Consider for instance a quantum computation model where locations refers to quantum bits. In the following term:

$$N \equiv \rho\ell.\langle \rho\ell_1.\rho\ell_2.M, \rho\ell_3.\mathfrak{Cnot}(\langle \ell_1, \ell_3 \rangle) \rangle$$

where $\mathfrak{Cnot}$ denotes the conditional not, the problem is the following : if ℓ, ℓ_1 and ℓ_2 are entangled in M, then a conditional not on ℓ, ℓ_3 can entangle ℓ_3 with ℓ_1, ℓ_2. The idea is that once a concrete physical object is defined in a term then this object may influence, or it may be influenced, by any other part of the surrounding context. The standard form of N (see def. 1 below) is:

$$\rho\ell, \ell_1, \ell_2, \ell_3.\langle M, \mathfrak{Cnot}(\langle \ell_1, \ell_3 \rangle) \rangle$$

The idea is that scope extrusion is a syntactic mean to take into account non compositionality. Indeed, in the standard form of N, the range of ℓ_3 is extended in such a way that it can interact, or more precisely that it is possible to describe within the term the interactions, with ℓ_2 for instance.

On the other hand ν binder only refer to an abstract pointer. It is an abstraction over outside computing devices that are going to be used to perform some computation. Consider the following program:

$$M1 = (\lambda x.\langle x, x \rangle \;\; \nu y.y)$$

we don't want this to be equivalent to

$$M2 = \nu y.(\lambda x.\langle x, x\rangle \;\; y)$$

since the evaluation of $!M1$ creates two different resources, whereas $M2$ only creates one resource.

Following these equivalences, a notion of standard form for terms naturally emerges (it corresponds to a distinguished term for each $\doteqdot$ equivalence class).

Fact 1 (Standard form) *For any term M, there exists $N \doteqdot M$ such that $N \equiv \rho\ell_1.\rho\ell_2.\ldots.\rho\ell_n.N_\rho$ with $\{\ell_1, \ldots, \ell_n\} \subseteq FV(N_\rho)$ and N_ρ has no subterm of the form $\rho\ell.N'$.*

In the following we work modulo $\doteqdot$. That is we suppose that terms are in standard form. Thus, when we consider a term t having a form different from $\rho\ell.M$, it implicitly means that t does not contain any ρ binder. Moreover $\rho\ell.M$ also implicitly means that ℓ occurs in M.

Operational semantics The global state is a mathematical view of a computing device. Let $\mathcal{S}$ be a function from $\mathbb{N}$ to some domain $\mathcal{D}$ which denotes the actual physical objects concerned (DNA, quantum bits etc). We write $\emptyset$ the state nowhere defined. The bridge between λ_{GS} terms and global state is done using a Linking function that maps locations (λ_{GS} terms) to naturals (the address of the physical entity). For instance in a quantum λ_{GS} system, the computing device can be an array of quantum bits. A location ℓ is associated to some qbits ($\mathcal{K}(\ell) = 12$ indicates that ℓ is the 12th qbit of the array of quantum bits).

A λ_{GS} system is defined up to a family of *actions* that can be applied to the global state. Take for instance a usual computer memory, classically you have three actions to manipulate it: you can allocate a new memory cell to store an information, read the content of a memory cell and update the content of an existing memory cell.

Each action $\mathtt{a}_i$, has a side effect on the computing device and yields a value that can be used for further computations. Thus, if the arity of $\mathtt{a}_i$ is n_i we must provide two functions :

$$\mathcal{F}_i^{\mathcal{D},\mathcal{K}} : (\mathcal{T}^{n_i} \times (\mathbb{N} \to \mathcal{D})) \to (\mathbb{N} \to \mathcal{D})$$
$$\mathcal{F}_i^{\mathcal{T},\mathcal{K}} : \mathcal{T}^{n_i} \to \mathcal{T}$$

$\mathcal{F}_i^{\mathcal{D},\mathcal{K}}$ models the side effect on the computing device by modifying the global state (modification of $\mathcal{S}$) and $\mathcal{F}_i^{\mathcal{T},\mathcal{K}}$ denotes the value returned to the λ_{GS} term.

As usual if f is a function we denote by $f \uplus \{x \mapsto v\}$ the function g such that $g(y) = f(y)$ for all $y \neq x$ and $g(x) = v$.

The evaluation of pure terms is done via β-reduction as in the standard λ-calculus. We write $M[x := N]$ the term M where all free occurrences of x have been replaced with N.

Definition 4 (Functional evaluation).

$$(\lambda x.M \quad N) \rightarrow_\beta M[x := N]$$

λ_{GS} evaluation process may alternate evaluation on pure terms as well as actions on the global state. We have the following definition of the computation:

Definition 5 (λ_{GS} computation). *We define $\rightsquigarrow$, the computation relation, between tuples $[\mathcal{S}, \mathcal{K}, M]$*

- *Function: if $M \rightarrow_\beta M'$:*

$$[\mathcal{S}, \mathcal{K}, M] \rightsquigarrow [\mathcal{S}, \mathcal{K}, M']$$

- *Action:*

$$[\mathcal{S}, \mathcal{K}, (a_i \quad \overrightarrow{M})] \rightsquigarrow [\mathcal{F}_i^{\mathcal{D},\mathcal{K}}(\overrightarrow{M}, \mathcal{S}), \mathcal{K}, \mathcal{F}_i^{\mathcal{T},\mathcal{K}}(\overrightarrow{M})]$$

- *Realization:*

$$[\mathcal{S}, \mathcal{K}, !\nu x.M] \rightsquigarrow [\mathcal{S} \uplus \{n+1 \mapsto d\}, \mathcal{K} \uplus \{o \mapsto n+1\}, \rho o.(M[x := o])]$$

with o a fresh name, d a distinguished value of $\mathcal{D}$ and $\mathcal{S}$ defined on $\{1, \ldots, n\}$.

2.3 λ_{GS} properties

Due to imperative nature of global state, it is clear that λ_{GS} might not be a confluent calculus. Actually λ_{GS} is intrinsically non confluent even with the simplest domain and actions. Consider the unit $\mathcal{D} = \{\mathfrak{d}\}$, and λ_{GS} with no actions. Now consider the following computation where β-reduction and realizations are alternatively performed. First, lets reduce the β-redex:

$$\begin{aligned}[\emptyset, \emptyset, (\lambda x.(x \quad x) \quad !\nu y.y)] &\rightsquigarrow [\emptyset, \emptyset, (!\nu y.y \quad !\nu y.y)] \\ &\rightsquigarrow [\{1 \mapsto \mathfrak{d}\}, \{o \mapsto 1\}, \rho o.(!\nu y.y \quad o)] \\ &\rightsquigarrow [\{1 \mapsto \mathfrak{d}, 2 \mapsto \mathfrak{d}\}, \{o \mapsto 1, o' \mapsto 2\}, \rho o'.\rho o.(o' \quad o)]\end{aligned}$$

Second, lets reduce the κ-redex:

$$\begin{aligned}[\emptyset, \emptyset, (\lambda x.(x \quad x) \quad \nu y.y)] &\rightsquigarrow [\{1 \mapsto \mathfrak{d}\}, \{o \mapsto 1\}, \rho o.(\lambda x.(x \quad x) \quad o)] \\ &\rightsquigarrow [\{1 \mapsto \mathfrak{d}\}, \{o \mapsto 1\}, \rho o.(o \quad o)]\end{aligned}$$

Thus, in order to have confluence, on has to fix a reduction strategy. We will not discuss this point further here, and let the study of reduction strategies for future work.

3 λ_{GS} for quantum computing

In this section we show how λ_{GS} can be used to define a functional quantum programming language. We develop a typing system which insures the "no-cloning" theorem. Our typing much more flexible than previous propositions based on a strict linear typing discipline for quantum bits (e.g. [21]). This typing system relies heavily on the new binders that make a syntactical distinction between abstract and concrete quantum bits.

3.1 λ^Q_{GS} definition

λ^Q_{GS} is a quantum programing system based on λ_{GS}. We suppose that the reader has basic knowledges of quantum computing (see [17]). It is a functional programming language for quantum computers based on the QRAM model [13]. The physical device of λ^Q_{GS} is an array of qbits. n qbits are represented as normalized vector of the Hilbert space $\otimes_{i=1}^{n} C^2$ noted as a ket vector : $|\varphi\rangle$.

quantum actions are standard unitary quantum gates like $\mathfrak{Cnot}, \mathfrak{T}, \mathfrak{H}$ (respectively control not, phase and Hadamard gates see [17] for precise definition) and measure $\mathfrak{M}$. λ^Q_{GS} requires two boolean constants 0 and 1 corresponding to the result of measurement.

Quantum measurement is probabilistic, therefore the measure action $\mathfrak{M}$ is probabilistic. However we do not focus on this particular point in this paper: we are just interested in developing a typing system for λ^Q_{GS}. Functions defining unitary actions are as follow:

$$\mathcal{F}_{\mathfrak{T}}^{|\varphi\rangle,\mathcal{K}}(q) = \mathfrak{T}_{\mathcal{K}_q}(|\varphi\rangle)$$

$$\mathcal{F}_{\mathfrak{H}}^{|\varphi\rangle,\mathcal{K}}(q) = \mathfrak{Cnot}_{\mathcal{K}_q}(|\varphi\rangle) \qquad \mathcal{F}_{\mathfrak{Cnot}}^{|\varphi\rangle,\mathcal{K}}(\langle q, q'\rangle) = \mathfrak{H}_{\mathcal{K}_q,\mathcal{K}(q')}(|\varphi\rangle)$$

$$\mathcal{F}_{\mathfrak{T}}^{\mathcal{T},\mathcal{K}}(q) = \mathcal{F}_{\mathfrak{H}}^{\mathcal{T},\mathcal{K}}(q) = q \qquad \mathcal{F}_{\mathfrak{Cnot}}^{\mathcal{T},\mathcal{K}}(\langle q, q'\rangle) = \langle q, q'\rangle$$

where $\mathfrak{T}_i, \mathfrak{H}_i, \mathfrak{Cnot}_{i,j}$ are the respective phase, hadamard and conditional not gates on quantum bits i and j (see [17]).

Let $|\varphi\rangle = \alpha\,|\varphi_O\rangle + \beta\,|\varphi_1\rangle$ be normalized with $|0\rangle$ and $|1\rangle$ being the i-th quantum bit and

$$|\varphi_O\rangle = \Sigma_i \alpha_i\,|\phi_i^0\rangle \otimes |0\rangle \otimes |\psi_i^0\rangle \qquad |\varphi_1\rangle = \Sigma_i \beta_i\,|\phi_i^1\rangle \otimes |1\rangle \otimes |\psi_i^1\rangle$$

then we define $\mu_0 =\mid \alpha \mid^2$ and $\mu_1 =\mid \beta \mid^2$. We use $=_p$ to define probabilistic functions: $f(x) =_p y$ means that $f(x)$ yields y with probability p.

$$\mathcal{F}_{\mathfrak{T}}^{\alpha|\varphi_O\rangle+\beta|\varphi_1\rangle,\mathcal{K}}(q) =_{\mu_0} 0 \qquad \mathcal{F}_{\mathfrak{T}}^{\alpha|\varphi_O\rangle+\beta|\varphi_1\rangle,,\mathcal{K}}(q) =_{\mu_1} 1$$

$$\mathcal{F}_{\mathfrak{T}}^{\alpha\mathcal{D}_O+\beta|\varphi_1\rangle,\mathcal{K}}(q) = \alpha\,|\varphi_O\rangle \qquad \mathcal{F}_{\mathfrak{T}}^{\alpha|\varphi_O\rangle+\beta|\varphi_1\rangle,\mathcal{K}}(q) = \beta\,|\varphi_1\rangle$$

Realization is implemented by choosing $|0\rangle$ as default value, hence:

$$[|\varphi\rangle, \mathcal{K}, !\nu x.M] \rightsquigarrow [\mathcal{S} \otimes |0\rangle, \mathcal{K} \uplus \{o \mapsto n+1\}, \rho o.(M[x := o])]$$

3.2 λ^Q_{GS} typing system

We now define a typing judgment for λ^Q_{GS} that ensures the no-cloning property of quantum states through a linear typing discipline.

Definition 6 (λ^Q_{GS} types). *Types are defined by:*

$$\tau ::= \mathbf{B} \mid \mathbf{Q} \mid \tau \to \tau \mid \tau \times \tau \mid \Diamond\tau$$

For the sake of simplicity we only consider two base types: $\mathbf{B}$ is the type of bits having $0, 1$ as constants, and $\mathbf{Q}$ is the type of quantum bits. Note that there are no constants of type $\mathbf{Q}$ in λ^Q_{GS}. Quantum bits are manipulated only through location names and linking function. It is because of the non-locality of quantum computations that quantum bits cannot be directly manipulated at the level of terms (see [21] for further explanations). Unlike [21] there is no primitive notion of linear types (there is no linear application function as well as exponential types): linearity is handled through ν and ρ binders only. Type $\Diamond\tau$ is used to identify ν redexes. Arrow and product types have their usual meanings.

In λ^Q_{GS}, ν and ρ only binds quantum bits locations. Thus x, ℓ respectively in $\nu x.N$ and $\rho\ell.M$ denote a location of a respectively abstract and concrete quantum bit and must be typed with $\mathbf{Q}$.

Typing contexts, written $\Gamma; \Delta$, are lists of couples made of variables and types in which variables are uniquely defined. In order to help the reading we cut typing contexts in two parts, the first part (Γ) is *c*lassical whereas the second one (Δ) is *l*inear. For instance in $x_1 : \tau_1, \ldots, x_n : \tau_n; x_1 : \tau_1, \ldots, x_m : \tau_m$, each x_i is classical and can be reused (contraction rule $[CTN]$) and discarded as will (weakening rule [WkgC]),and each y_i is linear and can be used exactly once but can be discarded (weakening rule [WkgL]).

It appears that linking function and global state do not play any role in the typing process since locations can only contains quantum bits. Therefore typing judgment is simply defined on λ_{GS} terms (and not in tuples including a global state a linking function and a term).

Definition 7 (Typing judgement). *λ^Q_{GS} typing judgment is a tuple $\Gamma; \Delta \vdash M : \tau$, where Γ, Δ are typing contexts, M is a λ^Q_{GS} term and τ is a λ^Q_{GS} type. Typing rules are given in Fig. 1.*

Note that when $.;. \vdash M : \tau$ implicitly means (remember our convention to consider terms in standard form), that ρ does not occur in M. In facts typing rules are defined on standard forms. Rule $[\rho I]$ is only applicable once at the root of a typing tree and removes all ρ of the term.

λ^Q_{GS} typing systems is both classical (relatively to typing context Γ) and linear (relatively to typing context Δ). The idea being that quantum data have to be treated in a linear way (because of the no cloning property) whereas classical data can be arbitrarily copied. Rules $[AxC], [WkgC], [ExC], [\to IC], [\times EC]$ are the classical counterparts of linear rules $[AxL], [WkgL], [ExL], [\to IL], [\times EL]$. Linearity is also to be found in rules $[\to E], [\times I], [\times EL], [\times EC]$ where linear context is split between premises of the rule. The fact that Γ is classical is achieved through the contraction rule $[CTN]$ (which has no counterpart for Δ hence insuring linearity on it) that allows the multiple use of a variable.

The premise $\cdot;\cdot \vdash M : \tau$ of rule $[AxL]$, together with the side condition $! \notin M$ is used to forbid declaration of a variable of a type τ containing a $\mathbf{Q}$ not

$$[CST0]\ \frac{}{\cdot;\cdot \vdash 0 : \mathbf{B}} \qquad [CST1]\ \frac{}{\cdot;\cdot \vdash 1 : \mathbf{B}}$$

$$[AxC]\ \frac{\cdot;\cdot \vdash M : \tau \qquad ! \notin M}{y:\tau;\cdot \vdash y : \tau} \qquad [AxL]\ \frac{}{\cdot; y:\tau \vdash y : \tau}$$

$$[ExC]\ \frac{\Gamma_1, y:\sigma', x:\sigma, \Gamma_2; \Delta \vdash M : \tau}{\Gamma_1, x:\sigma, y:\sigma', \Gamma_2; \Delta \vdash M : \tau} \qquad [ExL]\ \frac{\Gamma; \Delta_1, y:\sigma', x:\sigma, \Delta_2 \vdash M : \tau}{\Gamma; \Delta_1, x:\sigma, y:\sigma', \Delta_2 \vdash M : \tau}$$

$$[WkgC]\ \frac{\Gamma; \Delta \vdash M : \tau}{\Gamma, x:\sigma; \Delta \vdash M : \tau} \qquad [WkgL]\ \frac{\Gamma; \Delta \vdash M : \tau}{\Gamma; \Delta, x:\sigma \vdash M : \tau}$$

$$[\to IC]\ \frac{\Gamma, x:\sigma; \Delta \vdash M : \tau}{\Gamma; \Delta \vdash \lambda x.M : \sigma \to \tau} \qquad [\to IL]\ \frac{\Gamma; \Delta, x:\sigma \vdash M : \tau}{\Gamma; \Delta \vdash \lambda x.M : \sigma \to \tau}$$

$$[\to E]\ \frac{\Gamma; \Delta_1 \vdash M : \sigma \to \tau \qquad \Gamma; \Delta_2 \vdash N : \sigma}{\Gamma; \Delta_1, \Delta_2 \vdash (M\ \ N) : \tau} \qquad [\times I]\ \frac{\Gamma; \Delta_1 \vdash M : \tau \qquad \Gamma; \Delta_2 \vdash N : \sigma}{\Gamma; \Delta_1, \Delta_2 \vdash \langle M, N\rangle : \tau \times \sigma}$$

$$[\nu I]\ \frac{\Gamma; \Delta, x:\mathbf{Q} \vdash M : \sigma}{\Gamma; \Delta \vdash \nu x.M : \Diamond\tau} \qquad [\nu E]\ \frac{\Gamma; \Delta \vdash M : \Diamond\tau}{\Gamma; \Delta \vdash !M : \tau}$$

$$[CTN]\ \frac{\Gamma, x:\tau, y:\tau; \Delta \vdash M : \sigma}{\Gamma, z:\tau;\cdot \vdash M[x := z; y := z] : \sigma}$$

$$[\times EL]\ \frac{\Gamma'; \Delta_1 \vdash N : \tau \times \sigma \qquad \Gamma'; \Delta_2, x:\tau, y:\sigma \vdash M : \tau'}{\Gamma; \Delta_1, \Delta_2 \vdash \mathsf{let}\ \langle x, y\rangle = N\ \mathsf{in}\ M : \tau'}$$

$$[\times EC]\ \frac{\Gamma; \Delta_1 \vdash N : \tau \times \sigma \qquad \Gamma, x:\tau, y:\sigma; \Delta_2 \vdash M : \tau' \qquad \begin{array}{c}\cdot;\cdot \vdash Q : \tau \\ \cdot;\cdot \vdash P : \sigma\end{array} \qquad ! \notin \langle P, Q\rangle}{\Gamma; \Delta_1, \Delta_2 \vdash \mathsf{let}\ \langle x, y\rangle = N\ \mathsf{in}\ M : \tau'}$$

$$[\rho I]\ \frac{\Gamma; \Delta, \ell_1 : \mathbf{Q}, \ldots, \ell_n : \mathbf{Q} \vdash M : \tau}{\Gamma; \Delta \vdash \rho\ell_1, \ldots, \ell_n.M : \tau}$$

Fig. 1. λ^{Q}_{GS} typing rules

guarded by $\Diamond$ in the classical context (thus making possible the duplication of quantum bits). It is possible to introduce $y : \Diamond\mathbf{Q}$ in the classical context because $.;. \vdash \nu x.x : \Diamond\mathbf{Q}$. The same trick is used in rule $[\times EC]$, where we check that τ and σ do not contain unguarded $\mathbf{Q}$. In this paper we have only considered two $\times$ elimination rules: one is fully classical and the other one is fully linear. It would be easy to define typing rules of couples where one member is linear whereas the other one is classical.

Rules $[\nu I]$ and $[\nu E]$ are standard introduction and elimination rules for a connective (here $\Diamond$).

There is no elimination rule for binder ρ. This task will be fulfilled via garbage collection and since $[\rho I]$ does not modify the type of the expression, then it causes no harm to the subject reduction property.

The idea is that the programmer does not use ρ binders: they are completely managed by evaluation procedure. It causes no problem for subject reduction since there is no effect on the type of the term.

The standard subject reduction property is verified in λ^{Q}_{GS}.

Theorem 1 (Subject Reduction). *Let M be such that $\Gamma;\cdot \vdash M : \tau$ and $[|\varphi\rangle, \mathcal{K}, M] \rightsquigarrow [\mathcal{S}', \mathcal{K}, M']$, then $\Gamma;\cdot \vdash M' : \tau$*

Consider for instance the following piece of code :

$$cf \stackrel{def}{=} \nu x.\mathfrak{M}(\mathfrak{H}(x))$$

then $.;. \vdash cf : \Diamond\mathbf{B}$ is derivable. So one can use $x_{cf} : \Diamond\mathbf{B}$ in the classical typing context. Thus, it is possible to reuse x_{cf}(thanks to rule $[CTN]$). It is interesting since cf denotes a perfect coin flipping, and should be usable as many times as wanted in a probabilistic algorithm. This simple example is not possible to directly encode in [21] where qbits are linearly used: one version of x_{cf} must be defined for every use of the coin flipping or promotion must be used.

4 Conclusion

In this paper we present a variation of the pure λ-calculus: λ_{GS}. We introduce two new binders ν and ρ to bind variables that denote locations on a physical device implementing a global state over which a computation is performed. Usually variable name management (α-equivalence, fresh name generation) is handled through λ binding but it does create two kind of problems when names refer to an external device.

1. Compositionality is lost due to side effects that may occur in the external device: one cannot ensure that effects related to a given location name are limited to the lexical scope of this location name.
2. The description of an algorithm and its actual execution are two separate things in such a context. Think for instance at the no-cloning axiom in quantum computing: it only applies to *r*eal quantum bits, it should be possible to be able to duplicate abstract algorithms like the example of the fair coin-flipping.

Binder ρ is designed to take into account problem (1) whereas ν is designed to take into account problem (2). This distinction proves useful since it allows the definition of a type system for a quantum calculus that is more flexible than previous propositions of the literature [21, 6].

This work compares to the NEW calculus of Gabbay [12] where a context substitution is introduced through a new binder. Nevertheless, this works principally addresses problems related to α-equivalence and does not talk about scope extrusion. Another related work is the short note of Baro and Maurel [10] in which the λ binder is split into two constructions: a pure binder, ν and a combinator for the implementation of the β-reduction. In this case too the scope extrusion is not addressed, and there is no notion of state.

We believe that λ_{GS} will prove useful in other situations than the one presented in this paper. As future work we plan to work on a typing system to analyze entanglement/separation properties of quantum bits in λ^{Q}_{GS}. An idea would be to decorate **Q** types with entanglement classes (two quantum bits of the same class would be supposed entangled). As entanglement is typically a non compositional property, this would give an interesting application of ρ scope extrusion property.

Another line of work is to investigate other outside computing devices than quantum ones. For instance one can try to define and study functional computing languages for DNA computers [5] or chemical computers [4].

References

1. Quantiki - introductory tutotrials. http://www.quantiki.org/wiki/index.php/Category:Introductory_Tutorials.
2. M. Abadi, N. Banerjee, N. Heintze, and J. G. Riecke. A core calculus of dependency. In *Proceedings of the 26th Annual ACM Symposium on Principles of Programming Languages (POPL'99)*, pages 147–160, January 1999.
3. S. Abramsky, K. Honda, and G. McCusker. A fully abstract game semantics for general references. In *Proceedings of thirteenth Annual IEEE Symposium on Logic in Computer Science (LICS'98*, pages 334–344, 1998.
4. A. I. Adamatzky. Information-processing capabilities of chemical reaction-diffusion systems. 1. belousov-zhabotinsky media in hydrogel matrices and on solid supports. *Advanced Materials for Optics and Electronics*, 7(5):263–272, 1997.
5. L. M. Adleman. Molecular computation of solutions to combinatorial problems. *Science*, 266(11):1021–1024, 1994.
6. T. Altenkirch and J. Grattage. A functional quantum programming language. In *20th Annual IEEE Symposium on Logic in Computer Science*, 2005.
7. P. Arrighi. Quantum computation explained to my mother. *Bulletin of the EATCS*, 80:134–142, 2003.
8. H. P. Barendregt. *The Lambda Calculus; Its Syntax and Semantics*. North-Holland, 1984. Revised Edition.
9. H. P. Barendregt. Lambda calculi with types. In S. Abramsky, D. Gabbay, and T. Maibaum, editors, *Handbook of Logic in Computer Science*. Clarendon Press, Oxford, 1993.
10. S. Baro and F. Maurel. The qν and qνK calculi : name capture and control. Technical Report PPS//03/11//n16, Université Paris VII, March 2003.

11. M. Berger, K. Honda, and N. Yoshida. A logical analysis of aliasing in imperative higher-order functions. In O. Danvy and B. C. Pierce, editors, *Proceedings of the 10th ACM SIGPLAN International Conference on Functional Programming, ICFP 2005*, pages 280–293, 2005.
12. M. J. Gabbay. A NEW calculus of contexts. In *Proc of the 7th ACM SIGPLAN, Symposium on Principle and Practice of Declarative Programmning, PPDP'05*, pages 94–105. ACM Press, July 2005.
13. E. Knill. Convention for quantum pseudocode. Technical Report LAUR-96-2724, Los Alamos National Laboratory, 1996.
14. J. Launchbury and S. L. P. Jones. Lazy functional state threads. In *Proceedings of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation (PLDI'94)*, pages 24–35, 1994.
15. R. Milner. *Communicating and mobile systems: the π-calculus*. Cambridge University Press, 1999.
16. R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.
17. M. A. Nielsen and I. L. Chuang. *Quantum Computation and Quantum Information*. Cambridge University Press, 2000.
18. A. M. Pitts. Operational semantics and program equivalence. In G. Barthe, P. Dybjer, L. Pinto, and J. Saraiva, editors, *Applied Semantics, International Summer School, APPSEM 2000, Caminha, Portugal, September 9-15, 2000, Advanced Lectures*, volume 2395 of *Lecture Notes in Computer Science*, pages 378–412. Springer, 2002.
19. F. Pottier and V. Simonet. Information flow inference for ML. *ACM Transactions on Programming Languages and Systems*, 25(1):117–158, 2003.
20. F. Prost. A static calculus of dependencies for the λ-cube. In *Proc. of IEEE 15th Ann. Symp. on Logic in Computer Science (LICS'2000)*. IEEE Computer Society Press, 2000.
21. P. Selinger and B. Valiron. A lambda calculus for quantum computation with classical control. In P. Urzyczyn, editor, *Typed Lambda Calculi and Applications, 7th International Conference, TLCA 2005, Nara, Japan, April 21-23, 2005, Proceedings*, volume 3461 of *LNCS*, pages 354–368, 2005.
22. A. van Tonder. A lambda calculus for quantum computation. *SIAM J. Comput.*, 33(5):1109–1135, 2004.