Host-Core Calculi for Non-Classical Computations: A First Insight

Matteo Palazzo[1,†], Luca Paolini[1,†], Luca Roversi[1,†] and Margherita Zorzi[2,*,†]

[1]Università di Torino, Italy
[2]Università di Verona, Italy

Abstract

In many situations, programming languages with varying levels of expressiveness need to interact to achieve a final result. This paper is about HC, a framework derived from a categorical model that formalizes how two typed paradigmatic programming languages H and C can interact in a hierarchical manner: the host language H can operate on the core terms of C, but not the other way around. We recall the essential structure of the categorical model, and present a specific example of HCwhere H is capable of composing circuits that can be modeled using C. Finally, we discuss potential straightforward and natural generalizations of this instance of HC.

Keywords

Host-Core Calculi, Programming Paradigms, Computation and categorical Models, Non-Classical Computations

1. Introduction and insights

Multi-language programming concerns all those scenarios where several programming languages must interoperate. A first widespread scenario is the "Full-Stack Web Development" where for example, JavaScript is used for the frontend (user interface), Python or Ruby for the backend (business logic and database), and SQL to query a database, while HTML, CSS, TypeScript and Java are involved in various further roles. A second typical scenario occurs when we need to connect heterogeneous systems, thus we have to develop application interfaces (APIs) that allow interactions between systems (maybe involving JSON patterns). A third scenario is that of scientific analysis where it is necessary to cooperate elements of numerical calculation (MATLAB, Python) and simulation of physical phenomena (C++).

We are focused on a particular interoperability scenario, namely that of circuit programming. More precisely, we are interested in programming languages that are used to develop and describe reversible, quantum, analog, or the usual digital circuits. Digital circuits are often designed by means of Hardware Description Languages (HDLs) like VHDL and Verilog that

ICTCS'24: Italian Conference on Theoretical Computer Science, September 11–13, 2024, Torino, Italy
*Corresponding author.
†These authors contributed equally.
matteo.palazzo@unito.it (M. Palazzo); luca.paolini@unito.it (L. Paolini); luca.roversi@unito.it (L. Roversi); margherita.zorzi@univr.it (M. Zorzi)
https://www.di.unito.it/~rover (L. Roversi)
0009-0008-8455-8242 (M. Palazzo); 0000-0002-4126-0170 (L. Paolini); 0000-0002-1871-6109 (L. Roversi); 0000-0002-3285-9827 (M. Zorzi)

are endowed with suitable simulation tools (e.g. Vilinx Vivado and Quartus Prime) for the development of digital circuits. Often, HDLs programs are interfaced with software written in C/C++. Reversible circuits are a special class of digital circuits where every operation is physically reversible, so that the energy dissipation can be tamed in a precise way. Specialized languages and extensions of existing HDLs to describe reversible circuits are considered in RevKit and, sometimes, interoperates with quantum programming. The quantum programming is essentially done in terms of quantum circuits (see Quantum Azure, IBM Quantum Platform, ...) in a context controlled by a high-level language (typically Python) which sets up the desired orchestration of quantum operations. Last, a situation not dissimilar to the digital case occurs in the case of analog circuits, which typically must be interfaced with digital systems by using Verilog-AMS and VHDL-AMS (extensions of Verilog and VHDL for analog/mixed signal).

Languages distinctive traits. This paper aims to model the interaction between two languages: a core one, describing circuits and a host language that manipulates them. This interoperation is typical of any quantum development kit (e.g., see QiSKIT and QDK) that grasps an usual facet of quantum programming languages (see, for instance, [1, 2, 3, 4, 5, 6]), and reasonably and smoothly adapts to the other mentioned cases in which a classical computer must interact with circuits that perform specific tasks. For example, we are thinking about Field Programmable Gate Arrays (FPGAs) in the digital case, Field-Programmable Analog Arrays (FPAAs) in the analog case, quantum coprocessors (sometimes accessible via the cloud) in the quantum case, and energy-efficient circuits in the reversible case, all on the core side.

We notice that this work is meant to be explorative, so we will keep it as simple and intuitive as possible. We begin by identifying the distinguishing features that characterize the host-core scenario that we want to explore.

- The language for circuits we will formalize is iteration-free. It provides primitives to compose sub-circuits. The host language will allow us to manipulate sub-circuits, for example replicating them as required, possibly depending on some parameter. For example, an adder could be parameterized on the number of its input/output bits.

- The host language operating on the terms of a core language fits in the area of meta-programming, where programs manipulate other programs. We address to [7] for the readers interested to deepen the relations with meta-programming which can be summarized as follows: a program can be designed to read, generate, analyse, or transform another program.

Metodology. Our starting point is a categorical analysis as proposed in [8], in its turn inspired to [9, 10, 11], conceived as a basis to cope with a quantum programming perspective that implicitly outlines the characteristics that we here generalize, which also suggested us to adopt the terminology “host-core”.

On the categorical ground we define a typed calculus HC embodying two communicating languages H (host) and C (core), our calculus being the sound and complete internal language of the mentioned interacting categories. The language C is a linearly typed calculus sufficient to model the composition of circuits. The language H is essentially a simply typed lambda

calculus characterized by the type $\texttt{Proof}(A, B)$, where A and B are types for the terms in C. The presence of types with form $\texttt{Proof}(A, B)$ in H place H in a higher position than that of C in a straightforward hierarchy which establishes which language manipulates the terms of the other language. Specifically, host terms in H can manipulate core terms in C, namely circuits, but not vice versa.

It is worth noting that HC has some significant limitations. Firstly, for example, C will not include the fan-out wiring (see [12]), which refers to the ability to freely connect the output of a gate to multiple inputs of other gates, due to linearity constraints on its types. This limitation is quite relevant if the aim is to model digital or analog circuits. Secondly, H does not include constructs for iteration or recursion. At least apparently, this significantly restricts quantum programming languages that we might try to model by means of HC.

We here do not see the mentioned limitations as an obstacle to start the investigation that we summarize in this work. We will briefly discuss how to overcome limitations in a simple way in the conclusions.

2. A "kernel" host-core calculus HC

In this section we present the typed calculus HC. It builds upon the two components *host language* H, designed as a simply typed lambda calculus, and *embedded core language* C which we assume can be typed by a linear type discipline. By design, the host language is *privileged*, as compared to the core. Seen in the other way round, we do not allow that every of the terms we can build as part of the host language can be seen as terms at the core level. Roughly, the core language is hierarchically dependent on the host language.

HC differs from the language that Benton introduces in [9] because the communication between H and C cannot occur in both directions in HC. At the level of a categorical semantics, this means we do not require comonad or adjunction between the categories that model host and core components. Instead, we need a suitable *enriched category*.

2.1. Syntax of HC, and a type system

We strictly follow the notation in [10].

- Regarding the host language H, H-types are ranged over $X, Y, Z \ldots$, H-type contexts by Γ, Γ', H-variables by $x, y, z \ldots$, and H-terms by $s, t, v \ldots$.
- Regarding the core language C, C-types are ranged over by $A, B, C \ldots$, C-type contexts by Ω, Ω', C-variables by $a, b, c \ldots$, and C-terms by $h, f, g \ldots$.

We write $\Gamma \mid \Omega$ to identify a *mixed context* that includes both host (in Γ) and core (in Ω) type information. For every host context Γ, we denote the product of its elements by $\times_\Gamma$. Same for every core context Ω. We consider two fixed sets of *base types*: a set Σ_{C} of base C-types, ranged over by α (possibly indexed) and a set Σ_{H} of base H-types, ranged over by ι (possibly indexed).

The language $\mathscr{T}_C$ of types for the core language is generated by the grammar:

$$A, B, C \ ::= \ I \mid \alpha, \alpha_0, \alpha_1, \ldots \mid A \otimes B \tag{1}$$

with I the unit type, and $\alpha, \alpha_0, \alpha_1, \ldots$ elements of Σ_{C}, set of *base types* for C. An example of an element in Σ_{C} is **Bit** (see Section 3). The type I is the unit for the tensor $\otimes$, meaning that we can always think of $A_1 \otimes \ldots \otimes A_n$ and $I \otimes A_1 \otimes \ldots \otimes A_n$ as simply I whenever $n = 0$.

The language $\mathscr{T}_H$ of types for the host language is generated by the grammar:

$$X, Y, Z \ ::= \ 1 \mid \iota, \iota_0, \iota_1, \ldots \mid X \times Y \mid X \to Y \mid \texttt{Proof}(A, B) \tag{2}$$

with 1 the unit type, $\iota, \iota_0, \iota_1, \ldots$ elements of Σ_{H}, set of *base types* for H, and both A, B being C-types. Examples of elements in Σ_{H} are $\{\mathsf{Nat}, \mathsf{Bool}\}$.

Similarly to [4, 11], the host-type $\texttt{Proof}(A, B)$, with A, B core-types, is the formal tool to model where host and core syntax meet. Proof-theoretically $\texttt{Proof}(A, B)$ is the type of proofs from A to B in the host language. For us, core terms represent circuits that can be manipulated at the host level by means of this typing approach.

The grammars that generate the core language C and the host language H have the following mutually recursive definitions:

$$f, g ::= \bullet \mid a \mid f \otimes g \mid \mathbf{let}\ f\ \mathbf{be}\ a \otimes b\ \mathbf{in}\ h \mid \mathbf{let}\ f\ \mathbf{be}\ \bullet\ \mathbf{in}\ h \mid \mathbf{derelict}(s, f) \tag{3}$$

$$s, t ::= * \mid x \mid \langle s, t \rangle \mid \pi_1(s) \mid \pi_2(s) \mid \lambda x : X.t \mid st \mid \mathbf{promote}(a_1, \ldots, a_n.f) \tag{4}$$

respectively, where f in (4) is a core term generated by (3), and s in (3) is a host term generated by (4).

Core terms belong to a linear language[1] with let constructor and unit $\bullet$ with type I. Host terms are simply-typed lambda calculus with unit term $*$ with type 1. Finally, the functions $\mathbf{derelict}(\cdot, \cdot)$ and $\mathbf{promote}(\cdot.\cdot)$ model the *communication* between host H and core C.

The typing rules for HC are in Figure 1. The dereliction rule (der) "derelicts" a host term to the core language by applying it to a correctly typed (linear) core term; this choice is consistent with the standard presentation of this kind of rules (e.g., see [11]). Similarly, the promotion rule (prom) "promotes" a term from the core to the host by "preserving" the linear context.

Remark 1. [Promotion with empty linear contexts.] Let us assume that the closed core term f is typed by $\Gamma \mid - \vdash_{\mathsf{C}} f \ : A$. According to the promotion rule (prom) of Figure 1, we can derive $\Gamma \vdash_{\mathsf{H}} \mathbf{promote}(-.f) \ : \texttt{Proof}(I, A)$ because $A_1 \otimes \ldots \otimes A_n$ is simply the unit I. □

The following lemma can be proved by structural induction on the derivations that we can build by means of the rules in Figure 1.

Lemma 1. [Substitution]

(sub1) If $\Gamma \vdash_{\mathsf{H}} s \ : X$ and $\Gamma, x : X \mid \Omega \vdash_{\mathsf{C}} e \ : A$ then $\Gamma \mid \Omega \vdash_{\mathsf{C}} e[s/x] \ : A$.

(sub2) If $\Gamma \mid \Omega_1 \vdash_{\mathsf{C}} g \ : A$ and $\Gamma \mid \Omega_2, a : A \vdash_{\mathsf{C}} f \ : B$ then $\Gamma \mid \Omega_1, \Omega_2 \vdash_{\mathsf{C}} f[g/a] \ : B$.

(sub3) If $\Gamma \vdash_{\mathsf{H}} s \ : X$ and $\Gamma, x : X \vdash_{\mathsf{H}} t \ : Y$ then $\Gamma \vdash_{\mathsf{H}} t[s/x] \ : Y$.

[1] We use "linearity" to identify the linear use of variables (alternative linearity notions are discussed in [13, 14]).

$$\text{(av)}\quad \Gamma_1, x : X, \Gamma_2 \vdash_{\mathsf{H}} x \ : X \qquad\qquad \text{(uv)}\quad \Gamma \vdash_{\mathsf{H}} * \ : 1$$

$$\text{(pv)}\ \frac{\Gamma \vdash_{\mathsf{H}} s \ : X \qquad \Gamma \vdash_{\mathsf{H}} t \ : Y}{\Gamma \vdash_{\mathsf{H}} \langle s, t \rangle \ : X \times Y} \qquad\qquad (\pi 1 v)\ \frac{\Gamma \vdash_{\mathsf{H}} v \ : X \times Y}{\Gamma \vdash_{\mathsf{H}} \pi_1(v) \ : X}$$

$$(\pi 2 v)\ \frac{\Gamma \vdash_{\mathsf{H}} v \ : X \times Y}{\Gamma \vdash_{\mathsf{H}} \pi_2(v) \ : Y} \qquad\qquad \text{(aiv)}\ \frac{\Gamma, x : X \vdash_{\mathsf{H}} t \ : Y}{\Gamma \vdash_{\mathsf{H}} \lambda x : X.t \ : X \to Y}$$

$$\text{(aev)}\ \frac{\Gamma \vdash_{\mathsf{H}} t \ : X \to Y \qquad \Gamma \vdash_{\mathsf{H}} s \ : X}{\Gamma \vdash_{\mathsf{H}} t(s) \ : Y}$$

$$\text{(ac)}\quad \Gamma \mid a : A \vdash_{\mathsf{C}} a \ : A \qquad\qquad \text{(uc)}\quad \Gamma \mid - \vdash_{\mathsf{C}} \bullet \ : I$$

$$\text{(tc)}\ \frac{\Gamma \mid \Omega_1 \vdash_{\mathsf{C}} f \ : A \qquad \Gamma \mid \Omega_2 \vdash_{\mathsf{C}} g \ : B}{\Gamma \mid \Omega_1, \Omega_2 \vdash_{\mathsf{C}} f \otimes g \ : A \otimes B}$$

$$\text{(let1c)}\ \frac{\Gamma \mid \Omega_1 \vdash_{\mathsf{C}} f \ : A \otimes B \qquad \Gamma \mid \Omega_2, a : A, b : B \vdash_{\mathsf{C}} h \ : C}{\Gamma \mid \Omega_1, \Omega_2 \vdash_{\mathsf{C}} \mathbf{let}\ f\ \mathbf{be}\ a \otimes b\ \mathbf{in}\ h \ : C}$$

$$\text{(let2c)}\ \frac{\Gamma \mid \Omega_1 \vdash_{\mathsf{C}} f \ : I \qquad \Gamma \mid \Omega_2 \vdash_{\mathsf{C}} h \ : C}{\Gamma \mid \Omega_1, \Omega_2 \vdash_{\mathsf{C}} \mathbf{let}\ f\ \mathbf{be}\ \bullet\ \mathbf{in}\ h \ : C}$$

$$\text{(prom)}\ \frac{\Gamma \mid a_1 : A_1, \ldots, a_n : A_n \vdash_{\mathsf{C}} f \ : A}{\Gamma \vdash_{\mathsf{H}} \mathbf{promote}(a_1, \ldots, a_n.f) \ : \mathtt{Proof}(A_1 \otimes \ldots \otimes A_n, A)}$$

$$\text{(der)}\ \frac{\Gamma \vdash_{\mathsf{H}} t \ : \mathtt{Proof}(A, B) \qquad \Gamma \mid \Omega \vdash_{\mathsf{C}} f \ : A}{\Gamma \mid \Omega \vdash_{\mathsf{C}} \mathbf{derelict}(t, f) \ : B}$$

Figure 1: Typing rules

Notice that **(sub1)** allows us to substitute a host term for a variable in a core term. In designing **(sub2)**, we follow the line adopted in languages like QWire and EWire, which have some primitive notions of substitution for composing functions of $\mathtt{Proof}(\cdot, \cdot)$ type. Finally, **(sub3)** models the replacement of a term for a variable in the host language as well as in simply typed lambda calculus. These substitution lemmas are quite similar to that presented in the enriched effect calculus EEC, see [15, Prop. 2.3]

2.2. Equational theory for HC

The equational theory among the terms in H is the standard one induced by β/η-rules which equates simply typed lambda terms. Figure 2 introduces the equational theory of HC.

The equations in the rules of Figure 3 model how promotion and dereliction are each other dual, where in (der-prom) we use Ω to denote $[a_1 : A_1, \ldots, a_n : A_n]$.

$$\text{(el1)}\ \frac{\Gamma \mid \Omega_1, a : A, b : B \vdash_{\mathsf{C}} f\ : C \qquad \Gamma \mid \Omega_2 \vdash_{\mathsf{C}} g\ : A \qquad \Gamma \mid \Omega_3 \vdash_{\mathsf{C}} h\ : B}{\Gamma \mid \Omega_1, \Omega_2, \Omega_3 \vdash_{\mathsf{C}} \mathbf{let}\ g \otimes h\ \mathbf{be}\ a \otimes b\ \mathbf{in}\ f = f[g/a, h/b]\ : C}$$

$$\text{(el2)}\ \frac{\Gamma \mid \Omega_1, c : A \otimes B \vdash_{\mathsf{C}} f\ : C \qquad \Gamma \mid \Omega_2 \vdash_{\mathsf{C}} g\ : A \otimes B}{\Gamma \mid \Omega_1, \Omega_2 \vdash_{\mathsf{C}} \mathbf{let}\ g\ \mathbf{be}\ a \otimes b\ \mathbf{in}\ f[a \otimes b/c] = f[g/c]\ : C}$$

$$\text{(el3)}\ \frac{\Gamma \mid \Omega \vdash_{\mathsf{C}} f\ : A}{\Gamma \mid \Omega \vdash_{\mathsf{C}} \mathbf{let}\ \bullet\ \mathbf{be}\ \bullet\ \mathbf{in}\ f = f\ : A}$$

$$\text{(el4)}\ \frac{\Gamma \mid \Omega_1, a : I \vdash_{\mathcal{C}} f\ : A \qquad \Gamma \mid \Omega_2 \vdash_{\mathcal{C}} g\ : I}{\Gamma \mid \Omega_1, \Omega_2 \vdash_{\mathcal{C}} \mathbf{let}\ g\ \mathbf{be}\ \bullet\ \mathbf{in}\ f[\bullet/a] = f[g/a]\ : A}$$

Figure 2: Let-Evaluation rules for the core language C

$$\text{(der-prom)}\ \frac{\Gamma \mid \Omega \vdash_{\mathsf{C}} f\ : A \qquad \Gamma \mid \Omega_1 \vdash_{\mathsf{C}} g_1\ : A_1\ \ldots\ \Gamma \mid \Omega_n \vdash_{\mathsf{C}} g_n\ : A_n}{\Gamma \mid \Omega \vdash_{\mathsf{C}} \mathbf{derelict}(\mathbf{promote}(a_1, \ldots, a_n.f), g_1 \otimes \ldots \otimes g_n) = f[g_1/a_1 \ldots g_n/a_n]\ : A}$$

$$\text{(prom-der)}\ \frac{\Gamma \vdash_{\mathsf{H}} f\ : \mathtt{Proof}(A, B)}{\Gamma \vdash_{\mathsf{H}} \mathbf{promote}(a.\mathbf{derelict}(f, a)) = f\ : \mathtt{Proof}(A, B)}$$

Figure 3: Promote and Derelict duality rules

Observe that pure C-judgments with form:

$$- \mid \Omega \vdash_{\mathsf{C}} f\ : A \tag{5}$$

identify terms that we can interpret as purely linear structures that we can build by means of tensor products.

Remark. As compared to Benton LNL system, here we add the η rules to the calculus following the presentation of [10]. Observe that this is necessary to get a completeness result.

Specifically, the crucial difference between LNL original presentation and HC is that the interaction between H and C is *not* symmetric. Since we are programmatically defining HC as a minimal system, having a linear core is a natural starting choice. This could appear very limiting and difficult to overcome when writing a program. However, it is not the case.

On a first hand, to extend C with non-linear behavior, it would be enough to endow the rules for the tensor product with additive contexts and transform it into a Cartesian product. This can be easily reflected at the semantic level, restoring all the results about the correspondence between syntax and semantics [8]. Moreover, a smooth syntactical approach is proposed in the conclusion by discussing the fan-out representability: if suitable constants (gates for circuits) are assumed available in the core, then we can simulate the non-linearity.

On the other hand, the design of the system HC is motivated by the need to incorporate linearity into languages for quantum computing and multi-language frameworks, the goal being

to demonstrate how a minimal linear system like HC can be effectively integrated into a host language and extended for future applications. □

Example 1. [Composition in H of C terms] Let the following judgments:

$$\Gamma \vdash_{\mathsf{H}} s \ : \mathtt{Proof}(A, B) \qquad \Gamma \vdash_{\mathsf{H}} t \ : \mathtt{Proof}(B, C) \qquad \Gamma \mid a : A \vdash_{\mathsf{C}} a \ : A$$

be given. We can construct the term:

$$\Gamma \vdash_{\mathsf{H}} \mathbf{promote}(a.\mathbf{derelict}(t, \mathbf{derelict}(s, a))) \ : \mathtt{Proof}(A, C) \tag{6}$$

in H, denoted as $\mathbf{comp}(a.s, t)$, which becomes the *associative* composition in H of terms in C which we can think of as functions from A to B, and B to C. □

3. HC as circuit description language

We here introduce examples to show how we concretely think of using a formal framework like HC. The examples assume to equip HC with constants and functions to support basic circuit definition and manipulation, *specializing* HC to a system hosting a (toy) hardware definition language.

Since languages for circuit manipulation are particularly relevant for classical, probabilistic[16], reversible, and quantum computations[17, 18], we focus on sketching how Reversible Boolean Circuits, seen in [19] as a rewriting system instance of a 3-Category, can be built at the core level and composed at the host level.

The extensions of C and H with new terms and the specification of their operational semantics are kept at an intuitive level, in an illustrative perspective. This work is meant to show that modelling the interaction between two languages as we do is viable, leaving a full technical development to future work.

Example 2. [Reversible Boolean Circuits at the core level] We recall that any boolean reversible gate with arity k is interpreted as a self-dual boolean injective function with k input and k output.

We start by extending C to C^* to include boolean reversible gates. We need to fix a new type **Bit** and its two inhabitants $\mathbf{0}, \mathbf{1}$ such that:

$$\Gamma \mid - \vdash_{\mathsf{C}} \mathbf{0} \ : \mathbf{Bit} \qquad \Gamma \mid - \vdash_{\mathsf{C}} \mathbf{1} \ : \mathbf{Bit}$$

which we interpret obviously by the corresponding natural numbers.

The further extension of C^*, as compared to C, is by the set of boolean reversible gates "*Swap*" sw, "*Negation*" not, also known as "*Toffoli-one*", "*Toffoli-two*" cnot, and "*Toffoli-three*" ccnot whose typing rules are:

$$\frac{\Gamma \mid \Omega \vdash_{\mathsf{C}} f \ : \mathbf{Bit}}{\Gamma \mid \Omega \vdash_{\mathsf{C}} \mathsf{not}(f) \ : \mathbf{Bit}}$$

$$\frac{\Gamma \mid \Omega_f \vdash_{\mathsf{C}} f \ : \mathbf{Bit} \qquad \Gamma \mid \Omega_g \vdash_{\mathsf{C}} g \ : \mathbf{Bit}}{\Gamma \mid \Omega_f, \Omega_g \vdash_{\mathsf{C}} \mathsf{sw}(f, g) \ : \mathbf{Bit} \otimes \mathbf{Bit}}$$

$$\frac{\Gamma \mid \Omega_f \vdash_{\mathsf{C}} f \ : \mathbf{Bit} \qquad \Gamma \mid \Omega_g \vdash_{\mathsf{C}} g \ : \mathbf{Bit}}{\Gamma \mid \Omega_f, \Omega_g \vdash_{\mathsf{C}} \mathsf{cnot}(f, g) \ : \mathbf{Bit} \otimes \mathbf{Bit}}$$

$$\frac{\Gamma \mid \Omega_f \vdash_{\mathsf{C}} f \ : \mathbf{Bit} \qquad \Gamma \mid \Omega_g \vdash_{\mathsf{C}} g \ : \mathbf{Bit} \qquad \Gamma \mid \Omega_h \vdash_{\mathsf{C}} h \ : \mathbf{Bit}}{\Gamma \mid \Omega_f, \Omega_g, \Omega_h \vdash_{\mathsf{C}} \mathsf{ccnot}(f, g, h) \ : \mathbf{Bit} \otimes \mathbf{Bit} \otimes \mathbf{Bit}} \ .$$

The reversible gates can be interpreted as boolean functions with inpu/output values in $\{0, 1\}$:

$$\begin{aligned} \mathsf{sw}(x, y) &::= y \otimes x & \qquad \mathsf{not}(x) &::= 1 - x \\ \mathsf{cnot}(x, c) &::= x \oplus c & \qquad \mathsf{ccnot}(x, y, c) &::= xy \oplus c \ , \end{aligned}$$

where $\oplus$ is the exclusive or and xy the multiplication between x and y.

For cnot, modelling it as a self-dual operator, would mean to add the axiom:

$$\mathbf{let}\ \mathsf{cnot}(x, c)\ \mathbf{be}\ y \otimes c'\ \mathbf{in}\ \mathsf{cnot}(y, c') = x \otimes y$$

to C^*. The equation models that a series composition of $\mathsf{cnot}(x, c)$, and $\mathsf{cnot}(y, c')$ yields the identity on the tensor product $x \otimes y$ between the two "wires" x, and y. Analogous axioms would be required for sw, not, and ccnot.

For example, we can derive $\Gamma \mid \Omega \vdash_{\mathsf{C}} \mathsf{not}(\mathsf{not}(f)) \ : \mathbf{Bit}$ by means of the substitution rule $(sub2)$, as follows:

$$\cfrac{\cfrac{\Gamma \mid \Omega_f \vdash_{\mathsf{C}} f \ : \mathbf{Bit}}{\Gamma \mid \Omega_f \vdash_{\mathsf{C}} \mathsf{not}(f) \ : \mathbf{Bit}}}{\Gamma \mid \Omega_f \vdash_{\mathsf{C}} \mathsf{not}(\mathsf{not}(f)) \ : \mathbf{Bit}} \ .$$

Linear constraints on **Bit** forbid the duplication of variables with type **Bit**. This aspect is crucial in contexts like reversible and quantum computing where the no-cloning property on values must hold. □

Examples 3 and 4 below suggest how to accommodate the syntax of the host language to the extension in Example 2. In particular, we focus on the circuits manipulation at host level.

Example 3. [Promoting Reversible Boolean Circuits to the host level] Let our core language be C^* as in Example 2. The core constants $\mathbf{0}$ and $\mathbf{1}$ can be lifted the host level, so belonging to what we can call H^*, by means of $(prom)$:

$$\Gamma \vdash_{\mathsf{H}} \mathbf{promote}(-.\mathbf{0}) \ : \mathtt{Proof}(I, \mathbf{Bit})$$

$$\dfrac{\dfrac{\Gamma \vdash_{\mathsf{H}} f \ : \mathtt{Proof}(A,B) \qquad \Gamma \mid a : A \vdash_{\mathsf{C}} a \ : A}{\Gamma \mid a : A \vdash_{\mathsf{C}} \mathbf{derelict}(f,a) \ : B} \qquad \dfrac{\Gamma \vdash_{\mathsf{H}} g \ : \mathtt{Proof}(C,D) \qquad \Gamma \mid c : C \vdash_{\mathsf{C}} c \ : C}{\Gamma \mid c : C \vdash_{\mathsf{C}} \mathbf{derelict}(g,c) \ : D}}{\dfrac{\Gamma \mid a : A, c : C \vdash_{\mathsf{C}} \mathbf{derelict}(f,a) \otimes \mathbf{derelict}(h,c) \ : B \otimes D}{\Gamma \vdash_{\mathsf{H}} \mathbf{parallel}(a.f, c.h) \ : \mathtt{Proof}(A \otimes C, B \otimes D)}}$$

Figure 4: Parallel composition of two reversible boolean circuits lifted from C^*.

$$\Gamma \vdash_{\mathsf{H}} \mathbf{promote}(-.\mathbf{1}) \ : \mathtt{Proof}(I, \mathbf{Bit})$$

Likewise, lifting exists for gates and complex circuits built on gates. For example:

$$\begin{aligned} &\Gamma \vdash_{\mathsf{H}} \mathbf{promote}(a_1 \ldots a_n.\mathsf{not}\,(f)) \ : \mathtt{Proof}(\otimes_{\Omega}, \mathbf{Bit}) \\ &\Gamma \vdash_{\mathsf{H}} \mathbf{promote}(a_1 \ldots a_n, b_1 \ldots b_m.\mathsf{cnot}\,(f,g)) \ : \mathtt{Proof}(\otimes_{\Omega_f,\Omega_g}, \mathbf{Bit}) \ , \end{aligned}$$

where $\otimes_{\Omega_f,\Omega_g}$ denote $A_1 \otimes \ldots \otimes A_n \otimes B_1 \otimes \ldots \otimes B_m$, assuming that:

$$a_1 : A_1 \ldots a_n : A_n \vdash_{\mathsf{C}} f \ : \mathbf{Bit} \qquad\qquad b_1 : B_1 \ldots b_m : B_m \vdash_{\mathsf{C}} g \ : \mathbf{Bit} \ .$$

In fact, we can apply the lifting to H^* to more complex circuits in C^* to obtain further complex structures, like parallel composition. Let:

$$\begin{aligned} &\Gamma \vdash_{\mathsf{H}} f \ : \mathtt{Proof}(A,B) && \Gamma \vdash_{\mathsf{H}} g \ : \mathtt{Proof}(C,D) \\ &\Gamma \mid a : A \vdash_{\mathsf{C}} a \ : A && \Gamma \mid c : C \vdash_{\mathsf{C}} c \ : C \end{aligned}$$

be given, where, the terms f, g, and h have already be lifted to H^* from C^*. The parallel composition of f and g, which depend on a, and c, respectively, is:

$$\mathbf{parallel}(a.f, c.g) ::= \mathbf{promote}(a, c.(\mathbf{derelict}(f,a) \otimes \mathbf{derelict}(g,c)))$$

whose type is $\mathtt{Proof}(A \otimes C, B \otimes D)$. Fig. 4 derives it.

Extending the derivation in Fig. 4 with some more steps yields

$$\lambda f{:}\mathtt{Proof}(A,B).\, \lambda g{:}\mathtt{Proof}(C,D).\, \mathbf{parallel}(a.f, c.g) \tag{7}$$

with $a : A$ and $c : C$ open variables from C^*, not in H^*, and type

$$\mathtt{Proof}(A,B) \to \mathtt{Proof}(C,D) \to \mathtt{Proof}(A \otimes C, B \otimes D) \ .$$

A term analogous to (7), which, however, model the series composition, is:

$$\lambda h : \mathtt{Proof}(A,B).\lambda f : \mathtt{Proof}(B,C).\mathbf{comp}(a.h, f)$$

with type $\mathtt{Proof}(A,B) \to \mathtt{Proof}(B,C) \to \mathtt{Proof}(A,C)$ which applies h after f, where $\mathbf{comp}(a.h, f)$ is defined in Example 1. □

Example 4. [A circuit core language, III: Controlling Circuits, some reflections] Let us assume to call H^{++} the extension of H^* in Example 3 with boolean constants true, false, typed by the obvious rules, with:

$$\frac{\Gamma \vdash_{\mathsf{H}} t \ : \mathsf{bool} \qquad \Gamma \vdash_{\mathsf{H}} t_l \ : X \qquad \Gamma \vdash_{\mathsf{H}} t_r \ : X}{\Gamma \vdash_{\mathsf{H}} \mathtt{if}\ t\ \mathtt{then}\ t_l\ \mathtt{else}\ t_r \ : X}$$

Depending on the value of t, we can decide to operate either on t_l, or t_r which may well have been lifted from C^*. So, inside H^{++} it is possible control how to operate on circuits that come from C^*.

For example, let us assume that $a : I$, $b : I$, $c : \mathbf{Bit} \otimes \mathbf{Bit}$, and Γ be:

$$x : \mathsf{bool}, f : \mathtt{Proof}(\mathbf{Bit} \otimes \mathbf{Bit}, \mathbf{Bit} \otimes \mathbf{Bit}), g : \mathtt{Proof}(\mathbf{Bit} \otimes \mathbf{Bit}, \mathbf{Bit} \otimes \mathbf{Bit}) \ .$$

We can start to observe that the terms:

$$\mathbf{promote}(-.\mathbf{0} \otimes \mathbf{0}) \qquad\qquad \mathbf{promote}(-.\mathbf{1} \otimes \mathbf{1})$$

represent input values $\mathbf{0} \otimes \mathbf{0}$, and $\mathbf{1} \otimes \mathbf{1}$ lifted from the core level to the host one. Then, we can derive the two following judgments:

$$\begin{aligned} \Gamma \vdash_{\mathsf{H}} & \overbrace{\mathbf{comp}(a.\mathbf{promote}(-.\mathbf{0} \otimes \mathbf{0}), f)}^{F[f]} \ : \mathtt{Proof}(I, \mathbf{Bit} \otimes \mathbf{Bit}) \\ \Gamma \vdash_{\mathsf{H}} & \underbrace{\mathbf{comp}(a.\mathbf{promote}(-.\mathbf{1} \otimes \mathbf{1}), g)}_{G[g]} \ : \mathtt{Proof}(I, \mathbf{Bit} \otimes \mathbf{Bit}) \ , \end{aligned}$$

where $F[f]$ (with free variable f) stands for the application of f to the input $\mathbf{promote}(-.\mathbf{0} \otimes \mathbf{0})$, and $G[g]$ (with free variable g) has analogous meaning, or the slightly more complex judgment:

$$\Gamma \vdash_{\mathsf{H}} \overbrace{\mathbf{comp}(b.\mathbf{promote}(-.\mathbf{1} \otimes \mathbf{1}), \mathbf{comp}(c.g, f))}^{GF[g,f]} \ : \mathtt{Proof}(I, \mathbf{Bit} \otimes \mathbf{Bit})$$

in which $GF[g, f]$ (with free variables g, f) represents tha application of the promoted g to the application of the promoted f to the promoted input $\mathbf{1} \otimes \mathbf{1}$.

Building on the previous terms, we can give the type:

$$\begin{aligned} & \mathsf{bool} \to \mathtt{Proof}(\mathbf{Bit} \otimes \mathbf{Bit}, \mathbf{Bit} \otimes \mathbf{Bit}) \\ & \quad \to \mathtt{Proof}(\mathbf{Bit} \otimes \mathbf{Bit}, \mathbf{Bit} \otimes \mathbf{Bit}) \to \mathtt{Proof}(I, \mathbf{Bit} \otimes \mathbf{Bit}) \end{aligned}$$

to the term:

$$\begin{aligned} \lambda x.\lambda f.\lambda g.\mathtt{if}\ x\ \mathtt{then}\ & \overbrace{\mathbf{comp}(a.\mathbf{promote}(-.\mathbf{0} \otimes \mathbf{0}), f)}^{F[f]} \\ \mathtt{else}\ & \underbrace{\mathbf{comp}(b.\mathbf{promote}(-.\mathbf{1} \otimes \mathbf{1}), \mathbf{comp}(c.g, f))}_{GF[g,f]} \end{aligned} \tag{8}$$

where we omit the types of lambda-abstracted variables for the sake of readability. Obviously, (8) evaluates only one of the two branches depending on the eventual value of x. □

Example 4 shows a very basic form of control by the host on the core. According to further extensions of the core C, one can define more significant control operations. For example, if we move to a Turing complete host, or at least to a typed system as expressive as Gödel System $\mathcal{T}$, we can think of defining series or parallel compositions of circuits whose length or dimension depends on a numeric parameter n, used to drive some iteration. These would be quite handy to define terms at the host level ready to build well-known quantum algorithms [1], given the value of n.

4. Categorical semantics and Internal Language Theorem

In this section we recall the main ideas and some results we inherit from the technical report [8]. A soundness and a completeness theorem w.r.t. a suitable semantics, as well as an *internal language theorem*, have been proved for HC in its basic format.

One can define a model for HC in terms of an instance of an enriched category, by interpreting the host part into a category H and the core part into a suitable category C enriched in H [20].

Definition 1. A model for HC is given by a pair $(\mathcal{H}, \mathcal{C})$, where $\mathcal{H}$ is a cartesian closed category, and $\mathcal{C}$ is a $\mathcal{H}$-*enriched* symmetric monoidal category.

Structure and interpretation are defined in a standard way. A ***structure*** for the language HC in $(\mathcal{H}, \mathcal{C})$ is specified by giving an object $[\![X]\!] \in \mathbf{ob}(\mathcal{H})$ for every base H-type X, and an object $[\![A]\!] \in \mathbf{ob}(\mathcal{C})$ for every base C-type A. Given these assignments, the other types are interpreted recursively. We address to [8] for the full definitions. Here we only focus on some peculiar aspects. For example, notice that the type $\mathtt{Proof}(A, B)$ is interpreted as $[\![\mathtt{Proof}(A, B)]\!] = \mathcal{C}([\![A]\!], [\![B]\!])$, i.e. as the set of morphisms from the denotaton of A to the denotation of B.

Theorem 1. [Soundness and Completeness] Enriched symmetric, monoidal categories on a cartesian closed category are sound and complete with respect to the calculus HC.

In the proof of the previous theorem and in the following results, some notions play a crucial role. The first one is the category $\mathrm{Th}(\mathsf{HC})$ of HC-theories, whose objects are HC-theories, i.e., type systems that properly extend HC with new base types, term symbols, and equality rules. The morphisms are translations between theories, mapping types and terms to types and terms, preserving type and term judgments.

Notice that concrete instances of HC introduced in the previous section are examples of HC-theories.

The other key notion to achieve the Internal Language theorem is the ***category of models*** $\mathrm{Model}(\mathsf{HC})$. This step is the more interesting and challenging part of the proof. The objects of $\mathrm{Model}(\mathsf{HC})$ are models of HC (see Definition 1). Morphisms of $\mathrm{Model}(\mathsf{HC})$ require to introduce the formal notion of *change of base*, that explains formally how by changing the host language one can induce a change of the embedded language. Given a cartesian closed functor $F\colon \mathcal{H}_1 \longrightarrow \mathcal{H}_2$ (a morphism between two models for H_1 and H_2 resp.), the induced ***change of base*** functor $F_*\colon \mathcal{H}_1\text{-}\mathbf{Cat} \longrightarrow \mathcal{H}_2\text{-}\mathbf{Cat}$ sends a $\mathcal{H}_1$-category $\mathcal{C}$ (a category for a core C_1

embedded in H_1) to the $\mathcal{H}_2$-category $F_*(\mathcal{C})$ (a category for a core C_2 embedded in H_2). $F_*(\mathcal{C})$ has the same objects as $\mathcal{C}$ and for every A and B of $\mathcal{C}$, one defines $F_*(\mathcal{C})(A,B) = F(\mathcal{C}(A,B))$; the composition, unit, associator and unitor morphisms in $F_*(\mathcal{C})$ are the images of those of $\mathcal{C}$ composed with the structure morphisms of the cartesian closed structure on F (see [21, Proposition 3.5.10] or [20] and [8]).

Notice that the notion of change of basis suggests how effectively the embedding of a core in a host works and how two obtain new embeddings: if one provides a translation between two host languages H_1 and H_2 and C is a core language for H_1, then one can use this translation of host languages to *change the host language* for C, obtaining an embedding in H_2. See [8] for all technical details.

We know from [10] that the typed calculus HC provides an internal language of $\mathrm{Model}(\mathsf{HC})$ if we can establish an equivalence $\mathrm{Th}(\mathsf{HC}) \equiv \mathrm{Model}(\mathsf{HC})$ between the category of models and the category of theories for HC [10, 22, 23, 24, 25].

We state now the theorem showing the equivalence between the categories of models and that of theories for HC.

Theorem 2. [Internal Language Theorem] The typed calculus HC provides an internal language for $\mathrm{Model}(\mathsf{HC})$, i.e. $\mathrm{Th}(\mathsf{HC})$ is equivalent to $\mathrm{Model}(\mathsf{HC})$.

The proof relies on the construction of a pair of functors: the first functor $S\colon \mathrm{Th}(\mathsf{HC}) \longrightarrow \mathrm{Model}(\mathsf{HC})$ assigns to a theory its syntactic category, and the second functor $L\colon \mathrm{Model}(\mathsf{HC}) \longrightarrow \mathrm{Th}(\mathsf{HC})$ assigns to a pair $(\mathcal{H},\mathcal{C})$ (a model of HC, see Definition 1) its internal language, as a HC-theory (see [8]).

5. Conclusion

We conclude this work by recalling some observations we left at the end of our introduction, which are related to the expressiveness of the components H, and C of HC.

A possible way to add a fan-out to C in order to duplicate values is to adopt Toffoli's proposal in [12], where he introduces gates and ancillary wires to reversible circuits, or improvements of Toffoli's proposal in [26] where ancillae are implicitly hidden.

Similarly, we can enhance the expressiveness of H by incorporating constants with types that align with the categorical semantics, and which can be interpreted as *recursion* or *while* iterators. For instance, a *recursion* iterator could have the type $\mathsf{Nat} \to (\mathsf{Nat} \to \mathsf{Nat}) \to \mathsf{Nat} \to \mathsf{Nat}$, where Nat denotes the type of natural numbers. This type can be easily interpreted in a straightforward generalization of the categorical model from the previous section, where the tensor product $\otimes$ can be replaced by the Cartesian product $\times$.

According to a wider perspective, we see how HC can be extended to model the interaction between a classical and a reversible, or quantum language. Specifically, we plan to explore how to add the constant "projection" to HC, "projection" being present in the languages explored in [27, 2, 1, 3, 28, 29], requiring also to identify the appropriate sub-categories of the one in [8].

References

[1] L. Paolini, M. Piccolo, M. Zorzi, QPCF: higher-order languages and quantum circuits, Journal Automated Reasoning 63 (2019) 941–966. doi:10.1007/s10817-019-09518-y.

[2] L. Paolini, M. Zorzi, qpcf: A language for quantum circuit computations, in: T. Gopal, G. Jäger, S. Steila (Eds.), Theory and Applications of Models of Computation, Springer International Publishing, Cham, 2017, pp. 455–469.

[3] L. Paolini, L. Roversi, M. Zorzi, Quantum programming made easy, Electronic Proceedings in Theoretical Computer Science 292 (2019) 133–147. doi:dx.doi.org/10.4204/EPTCS.292.8.

[4] J. Paykin, R. Rand, S. Zdancewic, Qwire: A core language for quantum circuits, in: Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, ACM, New York, NY, USA, 2017, pp. 846–858. doi:10.1145/3009837.3009894.

[5] M. ZORZI, On quantum lambda calculi: a foundational perspective, Mathematical Structures in Computer Science 26 (2016) 1107–1195. doi:10.1017/S0960129514000425.

[6] M. Zorzi, Quantum calculi—from theory to language design, Applied Sciences 9 (2019). URL: https://www.mdpi.com/2076-3417/9/24/5472. doi:10.3390/app9245472.

[7] M. Amy, Sized types for low-level quantum metaprogramming, in: M. K. Thomsen, M. Soeken (Eds.), Reversible Computation, Springer International Publishing, Cham, 2019, pp. 87–107.

[8] D. Trotta, M. Zorzi, Compositional theories for embedded languages, CoRR abs/2006.10604 (2020). URL: https://arxiv.org/abs/2006.10604. arXiv:2006.10604.

[9] P. N. Benton, A mixed linear and non-linear logic: Proofs, terms and models, in: L. Pacholski, J. Tiuryn (Eds.), Computer Science Logic, Springer Berlin Heidelberg, Berlin, Heidelberg, 1995, pp. 121–135.

[10] M. E. Maietti, P. Maneggia, V. de Paiva, E. Ritter, Relating categorical semantics for intuitionistic linear logic, Appl. Categ. Structures 13 (2005) 1–36.

[11] M. Rennela, S. Staton, Classical control, quantum circuits and linear logic in enriched category theory, Log. Methods Comput. Sci. 16 (2020). URL: https://doi.org/10.23638/LMCS-16(1:30)2020. doi:10.23638/LMCS-16(1:30)2020.

[12] T. Toffoli, Reversible computing, in: J. de Bakker, J. van Leeuwen (Eds.), Automata, Languages and Programming, Springer Berlin Heidelberg, Berlin, Heidelberg, 1980, pp. 632–644.

[13] L. Paolini, M. Piccolo, Semantically linear programming languages, in: Proceedings of the 10th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, PPDP '08, Association for Computing Machinery, New York, NY, USA, 2008, p. 97–107. doi:10.1145/1389449.1389462.

[14] M. Gaboardi, L. Paolini, M. Piccolo, On the reification of semantic linearity, Mathematical Structures in Computer Science 760 (2014) 829 – 867. doi:10.1017/S0960129514000401.

[15] J. Egger, R. E. Møgelberg, A. Simpson, The enriched effect calculus: syntax and semantics, J. Log. Comput. 24 (2014) 615–654. doi:10.1093/logcom/exs025.

[16] A. Di Pierro, A type theory for probabilistic λ–calculus, in: From Lambda Calculus to Cybersecurity Through Program Analysis: Essays Dedicated to Chris Hankin on the

Occasion of His Retirement, Lecture Notes in Computer Science, Springer International Publishing, Cham, 2020, pp. 86–102. URL: https://doi.org/10.1007/978-3-030-41103-9_3. doi:10.1007/978-3-030-41103-9_3.

[17] A. Di Pierro, M. Incudini, Quantum machine learning and fraud detection, in: D. Dougherty, J. Meseguer, S. A. Mödersheim, P. Rowe (Eds.), Protocols, Strands, and Logic, Springer International Publishing, Cham, 2021, pp. 139–155.

[18] Di Pierro, Alessandra, Mancini, Stefano, Memarzadeh, Laleh, Mengoni, Riccardo, Homological analysis of multi-qubit entanglement, EPL 123 (2018) 30006. URL: https://doi.org/10.1209/0295-5075/123/30006. doi:10.1209/0295-5075/123/30006.

[19] A. Barile, S. Berardi, L. Roversi, Termination of rewriting on reversible boolean circuits as a free 3-category problem, ArXiv abs/2401.02091 (2024). URL: https://api.semanticscholar.org/CorpusID:266231594.

[20] G. Kelly, Basic concepts of enriched category theory, Theory Appl. Categ. 10 (2005).

[21] E. Riehl, Categorical homotopy theory, Cambridge University Press, 2014.

[22] M. E. Maietti, V. de Paiva, E. Ritter, Categorical models for intuitionistic and linear type theory, in: J. Tiuryn (Ed.), Foundations of Software Science and Computation Structures, Springer Berlin Heidelberg, Berlin, Heidelberg, 2000, pp. 223–237.

[23] J. Lambek, P. J. Scott, Introduction to Higher Order Categorical Logic, Cambridge Univ. Press, 1986.

[24] A. M. Pitts, Categorical logic, in: S. Abramsky, D. M. Gabbay, T. S. E. Maibaum (Eds.), Handbook of Logic in Computer Science, volume 6, Oxford Univ. Press, 1995, pp. 39–.129.

[25] P. Johnstone, Sketches of an elephant: a topos theory compendium, volume 2 of *Studies in Logic and the foundations of mathematics*, Oxford Univ. Press, 2002.

[26] L. Paolini, M. Piccolo, L. Roversi, On a class of reversible primitive recursive functions and its turing-complete extensions, New Generation Computing 36 (2018) 233–256. doi:10.1007/s00354-018-0039-1.

[27] L. Paolini, M. Piccolo, L. Roversi, A certified study of a reversible programming language, in: T. Uustalu (Ed.), 21st International Conference on Types for Proofs and Programs, TYPES 2015, May 18-21, 2015, Tallinn, Estonia, volume 69 of *LIPIcs*, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2015, pp. 7:1–7:21. doi:10.4230/LIPICS.TYPES.2015.7.

[28] A. B. Matos, L. Paolini, L. Roversi, On the expressivity of total reversible programming languages, in: Reversible Computation: 12th International Conference, RC 2020, Oslo, Norway, July 9-10, 2020, Proceedings, Springer-Verlag, Berlin, Heidelberg, 2020, p. 128–143. doi:10.1007/978-3-030-52482-1_7.

[29] L. Paolini, M. Piccolo, L. Roversi, A class of recursive permutations which is primitive recursive complete, Theor. Comput. Sci. 813 (2020) 218–233. doi:10.1016/j.tcs.2019.11.029.