A Graded Modal Type Theory for Pulse Schedules

Robin Adams
Chalmers University of Technology, Gothenburg, Sweden
University of Gothenburg, Sweden

Jean-Philippe Bernardy
Chalmers University of Technology, Gothenburg, Sweden
University of Gothenburg, Sweden

Lorenzo Perticone
Chalmers University of Technology, Gothenburg, Sweden
University of Gothenburg, Sweden

Jeremy Pope
Chalmers University of Technology, Gothenburg, Sweden
University of Gothenburg, Sweden

Abstract

The operations to be performed by a quantum computer are almost invariably given in the form of a quantum circuit. In the final stage of compilation, a quantum circuit must be translated into the input signals accepted by the quantum hardware itself. For a quantum computer based on superconducting qubits, this will be a sequence of microwave control pulses to be sent to the various input channels. A *pulse schedule* gives a full specification for which pulse should be applied to which channel at what time. There is as yet no language for these pulse schedules that is very amenable to formal semantics.

In this paper, we propose such a language called GRAMPUS (GRAded Modal type theory for PUlse Schedules). It is a *graded modal type theory*, where the grades represent timing information: a variable $x :^{50} Q_1$ will represent a state of qubit Q_1 that will exist 50 nanoseconds in the future, and a variable $y :^{-75} Q_2$ will represent a state of qubit Q_2 that existed 75 nanoseconds in the past.

We give the syntax for two type theories, one with grades (the *annotated* language) and one without (the *plain* language). We prove some metatheoretic properties, and describe the semantics in terms of category theory. We show that the input signals to a quantum chip forms a model of the annotated language. We also give a syntactic model, prove that it is initial, and hence prove soundness and completeness theorems.

2012 ACM Subject Classification Software and its engineering → Formal language definitions; Theory of computation → Type theory; Theory of computation → Categorical semantics; Theory of computation → Quantum computation theory

Keywords and phrases Quantum computing, superconducting qubits, linear type theory, graded modal type theory

Digital Object Identifier 10.4230/LIPIcs.TYPES.2025.5

Supplementary Material *Software (Source Code)*: `https://codeberg.org/radams78/grampus.git` [2] archived at `swh:1:dir:bc79e493a8f11d72e5382289e1ef8d1284024082`

Funding This research was funded by the SSF (Swedish Foundation for Strategic Research) project QuantumStack, grant number FUS21-0063.

1 Introduction

The operations to be performed by a quantum computer are almost invariably given in the form of a *quantum circuit* (see e.g. [19, Ch. 4]). This is a well-defined language with universally agreed semantics: there is no ambiguity about the unitaries and measurement operations that a quantum circuit diagram denotes.

In the final stage of compilation, a quantum circuit must be translated into the input signals accepted by the quantum hardware itself. For a quantum computer based on superconducting qubits, this will be a sequence of microwave control pulses to be sent to the various input channels [13]. A *pulse schedule* gives a full specification for which pulse should be applied to which channel at what time.

There are languages for describing these pulse schedules, such as OpenPulse [16, 3]. However these languages are not very amenable to formal semantics. If we want, for example, a formally verified quantum compiler, then we require a language for specifying pulse schedules that can be given precisely defined semantics.

We propose a type theory called GRAMPUS (GRAded Modal type theory for PUlse Schedules). It is a linear type theory augmented with *grades* [7, 1, 12] (equivalently modalities) which represent relative timing information: a variable $x :^{50} Q_1$ will represent a state of qubit Q_1 that will exist 50 nanoseconds in the future, and a variable $y :^{-75} Q_2$ will represent a state of qubit Q_2 that existed 75 nanoseconds in the past. Such a language can then be compiled to a pulse schedule. The correctness of this compiler can be captured by the following commutation square:

$$\begin{array}{ccc} \Gamma \vdash_d t : A & \xrightarrow{\quad\lfloor\cdot\rfloor\quad} & \Gamma \vdash t : A \\ \Big\downarrow{\scriptstyle [\![\cdot]\!]_p} & & \Big\downarrow{\scriptstyle [\![\cdot]\!]_u} \\ Time \to Channel \to \mathbb{R} & \xrightarrow{\text{hardware model}} & \mathbb{C}^{2^n} \multimap \mathbb{C}^{2^n} \end{array} \tag{1}$$

In the above, $\Gamma \vdash t : A$ represents a *plain program* that corresponds to a quantum circuit without measurement (and with no timing information). The judgement $\Gamma \vdash_d t : A$ represents a quantum circuit annotated with timings. Such annotations can be erased ($\lfloor\cdot\rfloor$) to obtain a plain program $\Gamma \vdash t : A$. In the semantics, a pulse schedule maps every channel and every time to a signal: $(Channel \to Time \to \mathbb{R})$. We denote the space of unitary operators as $\mathbb{C}^{2^n} \multimap \mathbb{C}^{2^n}$.

The plain program can be interpreted to a unitary operator. The hardware model maps a pulse schedule to the unitary operator that it implements. The interpretation function $[\![\cdot]\!]_u$ maps a program $\Gamma \vdash t : A$ to a unitary operator in $[\![\Gamma]\!]_u \multimap [\![A]\!]_u$ (Section 3.3). The interpretation function $[\![\cdot]\!]_p$ maps an annotated program $\Gamma \vdash_d t : A$ into a global pulse schedule, by combining the pulses associated with each gate occurring in the program (Section 3.4).

If the hardware model is correct then the diagram commutes (Theorem 25), ensuring that a pulse schedule implements the same unitary operator as its corresponding circuit.

2 Syntax of the Programming Languages

A quantum chip consists of a number of *qubits* (quantum bits) with input channels (or drive lines) and output channels (or readout lines). There are certain operations or *quantum gates* that we wish to perform on the qubits. We assume given a number of gates (usually called the *native gates* for the chip) such that we know the signal (or pulse) that must be sent down the appropriate input channel(s) in order to effect that gate.

Our languages are based on the multiplicative fragment of (first-order) intuitionistic linear logic [4]. Namely, we have products $(A \otimes B)$ with unit 1. Additionally, we have a base type q_i for each qubit q_i, representing the state of that qubit. We can also annotate a type with a (relative) timing $(d\,A)$ meaning the data will be available exactly d units of time in the future. In the annotated language, every variable in the context will be annotated with its relative availability. In what follows, the part of the syntax related to annotations is shown in blue. The erasure function $\lfloor\cdot\rfloor$ removes exactly this part.

We assume given:

- a set $\mathbf{Q}$ of *qubits* (or *qubit labels*)
- a set $\mathbf{G}$ of *gates* (or *gate constants*)
- for every gate $G \in \mathbf{G}$, a tuple $(q_1, \ldots, q_n)$ of distinct elements of $\mathbf{Q}$ which the gate acts on, and a non-negative real number d_G, the *duration* of the gate G. We say that a gate G is an *n-ary gate* iff its tuple of qubits has length n. We shall often write $(G, d_G, q_1, \ldots, q_n) \in \mathbf{G}$.

The syntax of GRAMPUS is then given by the grammar:

$$
\begin{aligned}
\text{Type } A, B \quad & ::= 1 \mid q \mid A \otimes B \mid \boxed{d}\, A \\
\text{Context } \Gamma \quad & ::= [] \mid \Gamma, x :^{d} A \\
\text{Term } s, t \quad & ::= x \mid \star \mid \text{let} \star = s \,\text{in}\, t \mid G(t_1, \ldots, t_n) \mid (s, t) \\
& \quad \mid \text{let } (x, y) = s \,\text{in}\, t \mid \text{box } d\ t \mid \text{let } \ \text{box } d\ x = s \ \text{in}\ t \mid \ \text{delay}_d(t) \\
& \quad \mid \text{let}\, x = s \,\text{in}\, u \\
\text{Judgement } \mathcal{J} \quad & ::= \Gamma \vdash t : A \mid \Gamma \vdash s = t : A
\end{aligned}
$$

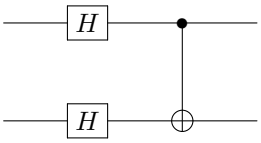
where $q \in \mathbf{Q}$, $d \in \mathbb{R}$ and $G \in \mathbb{G}$ is an n-ary gate.

▶ **Example 1.** Consider a very simple quantum chip with two qubits q_1 and q_2, where the native gates are:

- H_1, a Hadamard gate applied to q_1 with duration 100 nanoseconds;
- H_2, a Hadamard gate applied to q_2 with duration 110 nanoseconds;
- $CNOT$, a controlled NOT gate applied to q_1 and q_2 with duration 500 nanoseconds

That is, $\mathbf{Q} = \{q_1, q_2\}$ and $\mathbf{G} = \{(H_1, 100, q_1), (H_2, 110, q_2), (CNOT, 500, q_1, q_2)\}$.

We wish to run the following circuit on this chip:



In the plain language, this is represented by the term

$$x : q_1, y : q_2 \vdash CNOT(H_1(x), H_2(y)) : q_1 \otimes q_2$$

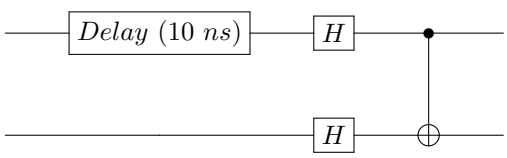
In the annotated language, we can represent this as a pulse schedule where q_1's timeline starts at time -600 and q_2's timeline starts at time -610 as follows:

$$x :^{-600} q_1, y :^{-610} q_2 \vdash CNOT(H_1(x), H_2(y)) : q_1 \otimes q_2$$

We can start the two qubits at the same time (-610) by inserting a delay of 10 ns before the Hadamard gate on q_1:

$$x :^{-610} q_1, y :^{-610} q_2 \vdash CNOT(H_1(\text{delay}_{10}(x)), H_2(y)) : q_1 \otimes q_2$$

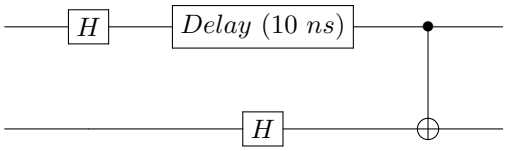
representing the following sequence of operations on the chip:



Or we can represent the pulse schedule where we apply the Hadamard gate to both qubits starting at time -610, and insert a delay of 10 nanoseconds after the Hadamard gate on qubit q_1, as follows:

$$x :^{-610} q_1, y :^{-610} q_2 \vdash CNOT(\text{delay}_{10}(H_1(x)), H_2(y)) : q_1 \otimes q_2$$

representing the following sequence of operations on the chip:



In both of these examples, we start the pulse schedule at a negative time, and then end at time zero. We can shift the second example so it starts at time 0 and ends at time 610 as follows:

$$x :^{0} q_1, y :^{0} q_2 \vdash \text{box } 610\ CNOT(H_1(\text{delay}_{10}(x)), H_2(y)) : \boxed{610}(q_1 \otimes q_2)$$

As expected, terms of the language allow manipulating products, but we can also apply an n-ary quantum gate G operating on an n-tuple of the qubits, or apply a delay of any duration to any qubit, which represents sending no input signal for that period of time. (In the ideal quantum computer, applying a delay does not change the qubit's state. In a real quantum computer, the state of an idle qubit quickly decays as it loses energy to, and gains noise from, the environment.)

The typing rules for the language are given in Figure 1. The key point is that if a gate G takes d units of time to perform its operation in hardware, then we annotate the corresponding inputs by $-d$, capturing the fact that to obtain the expected output *now*, the inputs must be provided d units of time in the past. The introduction and elimination rules for $\boxed{d}A$ are setup in a way to correctly represent a time-shift by d, following the pattern of general frameworks for grades or modalities [7, 1, 12]. It is easy to see that erasure preserves typing.

2.1 Judgemental Equality

We have rules establishing that judgemental equality is reflexive, symmetric, transitive, and a congruence for each of the primitive constructors, plus:

Beta Rules

$$\frac{\Gamma \vdash t : A}{\Gamma \vdash (\text{let } * = * \text{ in } t) = t : A}$$

$$\frac{\Gamma \vdash s : A \qquad \Delta \vdash t : B \qquad x :^{d} A, y :^{d} B, \Theta \vdash u : C}{d + \Gamma, d + \Delta, \Theta \vdash (\text{let } (x, y) = (s, t) \text{ in } u) = u[x := s, y := t] : C}$$

$$\frac{\Gamma \vdash s : A \qquad x :^{e} A, \Delta \vdash t : B}{e + \Gamma, \Delta \vdash (\text{let box } d\ x = \text{box } d\ s \text{ in } t) = t[x := s] : B}$$

$$\frac{\Gamma \vdash s : A \qquad x :^{e} A, \Delta \vdash t : B}{e + \Gamma, \Delta \vdash (\text{let } x = s \text{ in } t) = t[x := s] : B}$$

$$\frac{\Gamma, x :^{d} A, y :^{e} B, \Delta \vdash t : C}{\Gamma, y :^{e} B, x :^{d} A, \Delta \vdash t : C}\text{Swap} \qquad \frac{}{x :^{0} A \vdash x : A}\text{Var} \qquad \frac{}{\vdash * : 1}\text{Unit-I}$$

$$\frac{\Gamma \vdash s : 1 \qquad \Delta \vdash t : A}{d{+}\Gamma, \Delta \vdash \text{let} * = s \,\text{in}\, t : A}\text{Unit-E} \qquad \frac{\Gamma_1 \vdash t_1 : q_1 \qquad \cdots \qquad \Gamma_n \vdash t_n : q_n}{-d + \Gamma_1, \ldots, -d + \Gamma_n \vdash G(t_1, \ldots, t_n) : q_1 \otimes \cdots \otimes q_n}\text{Gate}$$

$$\frac{\Gamma \vdash s : A \qquad \Delta \vdash t : B}{\Gamma, \Delta \vdash (s, t) : A \otimes B}\text{Tens-I} \qquad \frac{\Gamma \vdash s : A \otimes B \qquad x :^{d} A, y :^{d} B, \Delta \vdash t : C}{d{+}\Gamma, \Delta \vdash \text{let}(x, y) = s \,\text{in}\, t : C}\text{Tens-E}$$

$$\frac{\Gamma \vdash t : A}{d + \Gamma \vdash \text{box}\ d\ t : \boxed{d}A}\text{Box-I} \qquad \frac{\Gamma \vdash s : \boxed{d}A \qquad x :^{e} A, \Delta \vdash t : B}{e - d{+}\Gamma, \Delta \vdash \text{let box}\ d\ x = s\ \text{in}\ t : B}\text{Box-E}$$

$$\frac{\Gamma \vdash t : q \qquad d \geq 0}{-d{+}\Gamma \vdash \text{delay}_d(t) : q}\text{Delay} \qquad \frac{\Gamma \vdash s : A \qquad x :^{d} A, \Delta \vdash t : C}{d{+}\Gamma, \Delta \vdash \text{let}\, x = s \,\text{in}\, t : C}\text{Let}$$

Figure 1 Typing rules for our languages. In the Gate rule, G is a gate acting on $(q_1, \ldots, q_n)$ with duration d. Timing annotations are highlighted in blue. We write $d + \Gamma$ for the result of increasing every grade in the context Γ by d.

Eta Rules

$$\frac{\Gamma \vdash t : I}{\Gamma \vdash (\text{let} * = t \ \text{in} *) = t : I}$$

$$\frac{\Gamma \vdash t : A \otimes B}{\Gamma \vdash (\text{let}\ (x, y) = t \ \text{in}\ (x, y)) = t : A \otimes B}$$

$$\frac{\Gamma \vdash t : \boxed{d}A}{\Gamma \vdash (\text{let box}\ d\ x = t \ \text{in box}\ d\ x) = t : \boxed{d}A}$$

Note we do not have $t = *$ for $t : 1$. (If t represents a process that takes time > 0 then it is not semantically equal to $*$.)

Commuting Conversions

In any linear type theory we also need judgemental equality rules for *commuting conversions*. The full list is given in Appendix 9.

3 Semantics

In this section, we describe the interpretation for our languages. We first introduce algebraic structures capturing what is needed to interpret such languages (models), and describe a general procedure to construct interpretations which we shall employ throughout this section. In the following, we assume fixed a set of qubit labels $\mathbf{Q}$ and a set of gates $\mathbf{G}$, whose elements $(G, d_G, q_1, \ldots, q_n)$ encode that gate G acts on qubits $(q_1, \ldots, q_n)$ in d_G units of time.

3.1 Categorical model of the plain language

The plain language can be interpreted in arbitrary symmetric monoidal categories, using the ideas from [7].

▶ **Definition 2** (Categorical model of the plain language)**.** A *(categorical) model* for the plain language $(\mathcal{C}^{\otimes}, X_-, f_-)$ is given by the following data:

- A symmetric monoidal category (SMC) $\mathcal{C}^{\otimes}$,
- For every qubit $q \in \mathbf{Q}$, an object $X_q : \mathcal{C}^{\otimes}$,
- For every gate $(G, d, q_1, \ldots, q_n)$ a morphism $f_G : \bigotimes_{i=1}^{n} X_{q_i} \to \bigotimes_{i=1}^{n} X_{q_i}$

We can hence interpret the plain language in any of its categorical models: the interpretation is constructed as follows.

▶ **Definition 3** (Interpretation of the plain language)**.** Given a categorical model of the plain language $(\mathcal{C}^{\otimes}, X_-, f_-)$, we define the following interpretation $[\![-]\!]_u$ operation:

- For every type A, we define an object $[\![A]\!]_u \in \mathcal{C}^{\otimes}$; we set $[\![1]\!]_u := I$, $[\![q]\!]_u := X_q$ and $[\![A \otimes B]\!]_u := [\![A]\!]_u \otimes [\![B]\!]_u$;
- For every context Γ, we define an object $[\![\Gamma]\!]_u$; we set $[\![[]]\!]_u := I$ and $[\![x : A, \Gamma]\!]_u := [\![A]\!]_u \otimes [\![\Gamma]\!]_u$;
- For every derivable judgement $\Gamma \vdash t : A$, we define a morphism $[\![\Gamma \vdash t : A]\!]_u := [\![\Gamma]\!]_u \xrightarrow{[\![t]\!]_u} [\![A]\!]_u$, where $[\![t]\!]_u$ is defined by induction on the judgement's derivation: Var and Unit-I correspond to identities, Swap is composition with $\mathcal{C}^{\otimes}$'s symmetry, Unit-E and Tens-I are the obvious tensor products, Tens-E becomes the following morphism:

$$[\![\mathrm{let}(x, y) = s \,\mathrm{in}\, t]\!]_u = [\![\Gamma]\!]_u \otimes [\![\Delta]\!]_u \xrightarrow{[\![s]\!]_u \otimes \mathbf{Id}} [\![A]\!]_u \otimes [\![B]\!]_u \otimes [\![\Delta]\!]_u \xrightarrow{[\![t]\!]_u} [\![C]\!]_u$$

and Gate becomes the following morphism:

$$[\![G(t_1, \ldots, t_n)]\!]_u = \bigotimes_{i=1}^{n} [\![\Gamma_i]\!]_u \xrightarrow{\bigotimes_{i=1}^{n} [\![t_i]\!]_u} \bigotimes_{i=1}^{n} [\![q_i]\!]_u \xrightarrow{f_G} \bigotimes_{i=1}^{n} [\![q_i]\!]_u$$

▶ **Proposition 4.** *The interpretation defined above preserves judgemental equality: if the judgment $s = t$ is admissible, then $[\![s]\!]_u = [\![t]\!]_u$.*

Proof. This is straightforward to prove by induction on the derivation of the equality judgement. ◀

3.2 Categorical model of the annotated language

In analogy to the previous subsection, we can interpret the annotated language in (appropriately structured) SMCs, which we describe here. Most of our definitions are analogous to what we described for the plain language. We additionally introduce a symmetric monoidal endofunctor $\mathsf{Shift}(x)$ for any real number $x \in \mathbb{R}$, to interpret time shifts (through their action on objects, hence on types and contexts) and the box construct (through their action on morphisms, hence on terms). Symmetry and monoidality for such endofunctors corresponds to the grades interacting appropriately with tensor products and unit types. In order to package the endofunctors $\mathsf{Shift}(x)$, we first recall the notion of a symmetric monoidal *module category* [8] (sometimes also called *actegories*).

▶ **Definition 5** (Symmetric monoidal $\mathcal{B}^{\otimes}$-module categories)**.** Given a monoidal category $\mathcal{B}^{\otimes} = (\mathcal{B}, 1, \otimes, \dots)$, a *symmetric monoidal $\mathcal{B}^{\otimes}$-module* category $\mathcal{C}^{\boxtimes,\bullet} = (\mathcal{C}^{\boxtimes}, \bullet)$ ($\mathcal{B}^{\otimes}$-SMMC in the future, or just SMMC if no emphasis is put on $\mathcal{B}^{\otimes}$) is the data of an SMC $\mathcal{C}^{\boxtimes} = (\mathcal{C}, I, \boxtimes, \dots)$ together with a strong monoidal functor $\bullet : \mathcal{B}^{\otimes} \to \mathbf{SMC}[\mathcal{C}^{\boxtimes}, \mathcal{C}^{\boxtimes}]^{\circ}$ into the monoidal (under composition) category of strong symmetric monoidal endofunctors of $\mathcal{C}^{\boxtimes}$.

▶ **Definition 6** (Categorical model of the annotated language)**.** A categorical model of the annotated language $(\mathcal{C}^{\otimes,\bullet}, X_{\_}, f_{\_}, \mathrm{delay}_{\_})$ is given by the following (where $\mathbb{R}^{+}$ denotes the discrete SMC of real numbers under addition):

- An $\mathbb{R}^{+}$-SMMC $\mathcal{C}^{\otimes,\bullet}$ (we shall write $\mathsf{Shift}(d) := (d \bullet -)$),
- For every qubit $q \in \mathbf{Q}$ an object $X_q : \mathcal{C}^{\otimes}$,
- For every gate $(G, d, q_1, \dots, q_n)$, a morphism $f_G : \mathsf{Shift}(-d)(\bigotimes_{i=1}^{n} X_{q_i}) \to \bigotimes_{i=1}^{n} X_{q_i}$,
- For every real number $x \in \mathbb{R}$ and qubit $q \in \mathbf{Q}$, a morphisms $\mathrm{delay}_x : \mathsf{Shift}(-x)(X_q) \to X_q$,

The above definition is worth unpacking: monoidality for the functor $\mathsf{Shift}(-)$ means that, for every real numbers $d_1, d_2 \in \mathbb{R}$ and object $X : \mathcal{C}^{\otimes}$, we get isomorphisms (natural in d_1, d_2, X, and appropriately coherent)

$$\mathsf{Shift}(0)(X) \simeq X \text{ and } \mathsf{Shift}(d_1 + d_2)(X) \simeq \mathsf{Shift}(d_1)(\mathsf{Shift}(d_2)(X))$$

and for a fixed $d \in \mathbb{R}$, monoidality of $\mathsf{Shift}(d)$ means that, for every $X, Y : \mathcal{C}^{\otimes}$, we get isomorphisms (natural in d, X, Y and, as before, appropriately coherent)

$$\mathsf{Shift}(d)(I) \simeq I \text{ and } \mathsf{Shift}(d)(X \otimes Y) \simeq \mathsf{Shift}(d)(X) \otimes \mathsf{Shift}(d)(Y)$$

This allows us to interpret the annotated language in any of its categorical models. The interpretation is built in a way analogous to Definition 3; the only differences relate to annotations, Box-I, Box-E and Delay (which are missing from the plain language).

▶ **Definition 7** (Interpretation for the annotated language)**.** Given a categorical model of the annotated language $(\mathcal{C}^{\otimes,\bullet}, X_{\_}, f_{\_}, \mathrm{delay}_{\_})$, we define an interpretation $[\![-]\!]_p$ as follows:

- For types we define $[\![-]\!]_p$ in the same way as $[\![-]\!]_u$, plus the clause $[\![d\, A]\!]_p := \mathsf{Shift}(d)[\![A]\!]_p$,
- For every context Γ, we define an object $[\![\Gamma]\!]_p$; we set $[\![[]]\!]_p := I$ and $[\![x :^{d} A, \Gamma]\!]_p := \mathsf{Shift}(d)[\![A]\!]_p \otimes [\![\Gamma]\!]_p$ (notice that monoidality of $\mathsf{Shift}(d)$ implies $[\![d + \Gamma]\!]_p \simeq \mathsf{Shift}(d)[\![\Gamma]\!]_p$),
- For judgements, we define $[\![-]\!]_p$ in the same way as $[\![-]\!]_u$ (after inserting coherences where appropriate); furthermore, we interpret

$$[\![\mathrm{delay}_d(t)]\!]_p = [\![-d + \Gamma]\!]_p \simeq \mathsf{Shift}(-d)[\![\Gamma]\!]_p \xrightarrow{\mathsf{Shift}(-d)[\![t]\!]_p} \mathsf{Shift}(-d)[\![q]\!]_p \xrightarrow{\mathrm{delay}_d} [\![q]\!]_p$$

$$[\![\mathrm{box}\ d\ t]\!]_p = [\![d + \Gamma]\!]_p \simeq \mathsf{Shift}(d)[\![\Gamma]\!]_p \xrightarrow{\mathsf{Shift}(d)[\![t]\!]_p} \mathsf{Shift}(d)[\![A]\!]_p$$

$$[\![\mathrm{let\ box}\ d\ x = s\ \mathrm{in}\ t]\!]_p = [\![(e - d) + \Gamma]\!]_p \otimes [\![\Delta]\!]_p \xrightarrow{\phi \otimes \mathbf{Id}} \mathsf{Shift}(e)[\![A]\!]_p \otimes [\![\Delta]\!]_p \xrightarrow{[\![t]\!]_p} [\![B]\!]_p$$

where the arrow ϕ is defined as

$$\phi = [\![(e-d)+\Gamma]\!]_p \simeq \mathsf{Shift}(e-d)([\![\Gamma]\!]_p) \xrightarrow{\mathsf{Shift}(e-d)[\![s]\!]_p} \mathsf{Shift}(e-d)(\mathsf{Shift}(d)[\![A]\!]_p) \simeq \mathsf{Shift}(e)[\![A]\!]_p$$

For instance, the term generated by Tens-I is interpreted as the following

$$\begin{aligned}
[\![\mathrm{let}(x, y) = s\,\mathrm{in}\,t]\!]_p &= [\![d + \Gamma, \Delta]\!]_p \\
&\simeq \mathsf{Shift}(d)([\![\Gamma]\!]_p) \otimes [\![\Delta]\!]_p \\
&\xrightarrow{\mathsf{Shift}(d)[\![s]\!]_p} \mathsf{Shift}(d)([\![A]\!]_p \otimes [\![B]\!]_p) \otimes [\![\Delta]\!]_p \\
&\simeq \mathsf{Shift}(d)[\![A]\!]_p \otimes \mathsf{Shift}(d)[\![B]\!]_p \otimes [\![\Delta]\!]_p \\
&\xrightarrow{[\![t]\!]_p} [\![C]\!]_p
\end{aligned}$$

and gates become

$$\begin{aligned}
[\![G(t_1,\dots,t_n)]\!]_p &= [\![-d+\Gamma_1,\dots,-d+\Gamma_n]\!]_p \\
&\simeq \mathsf{Shift}(-d)(\bigotimes_{i=1}^{n}[\![\Gamma_i]\!]_p) \\
&\xrightarrow{\mathsf{Shift}(-d)(\bigotimes_{i=1}^{n}[\![t_i]\!]_p)} \mathsf{Shift}(-d)(\bigotimes_{i=1}^{n}[\![q_i]\!]_p) \\
&\xrightarrow{f_G} \bigotimes_{i=1}^{n}[\![q_i]\!]_p
\end{aligned}$$

▶ **Proposition 8.** *The interpretation defined above preserves judgemental equality: if the judgment $s = t$ is admissible, then $[\![s]\!]_p = [\![t]\!]_p$.*

Proof. Again, this is straightforward to prove by induction on the derivation of the equality judgement. ◀

Every model of the plain language can be made into a model of the annotated language; this corresponds syntactically to erasing timing annotations (the $\lfloor - \rfloor$ operation defined in Section 1), and semantically to equipping an SMC with the trivial SMMC structure (where objects in $\mathbb{R}^+$ act identically).

▶ Remark 9. Given a model of the plain language $(\mathcal{C}^\otimes, X_-, f_-)$, we can define a model of the annotated language by defining

- Time shifts to be identity functors $\mathsf{Shift}(d) := \mathbf{Id}_{\mathcal{C}^\otimes}$ for every real number d, and
- Delays to be identity morphisms $\mathrm{delay}_d := \mathbf{Id}_{X_q}$.

3.3 Hilbert Spaces and Unitary Operators

Importantly, the category of finite-dimensional Hilbert spaces and unitary operators $\mathbf{FdHilb_U}$ is a categorical model of the plain language (Definition 2):

▶ **Proposition 10** (Categorical model of Unitary Operators)**.** *Assume that, for every n-ary gate G, we have a unitary operator $f_G : \mathbb{C}^{2^n} \to \mathbb{C}^{2^n}$. The category $\mathbf{FdHilb_U}$ is symmetric monoidal (i.e. $\mathbf{FdHilb}$ is symmetric monoidal and its identities, associators, unitors and symmetries are unitary operators). This, together with the assignments*

- *$X_q := \mathbb{C}^2$ for every qubit $q \in \mathbf{Q}$;*
- *f_G as the linear operator corresponding to the quantum gate G.*

makes it into a categorical model of the plain language.

In consequence, programs in the plain language can be interpreted in the ideal quantum computer – ignoring all aspects related to duration of execution. A program $\Gamma \vdash t : A$ hence corresponds to a unitary operator $[\![t]\!] \in [\![\Gamma]\!] \multimap [\![A]\!]$. In particular, for any gate G operating on n qubits, we assume given a corresponding operator $[\![G]\!] \in \mathbb{C}^{2^n} \multimap \mathbb{C}^{2^n}$.

3.4 Signals

In a physical quantum chip with superconducting qubits, there are a number of input channels or *drive lines* that can be used to perform operations on the qubits, and *readout lines* for receiving the outcome of measuring a qubit. (For details, see [13].)

The input to a drive line can be modelled as a function $[startTime, endTime] \to \mathbb{R}$. A complete pulse schedule consists then of such a function for each channel connected to such a drive line: $Channel \to [startTime, endTime] \to \mathbb{R}$. However, these do not form the

morphisms of an SMC; we could only form the tensor product of two pulse schedules if their start times and end times agree. To support the SMC structure, we must allow a different start or end time for each channel: $(c : Channel) \to [StartTime(c), EndTime(c)] \to \mathbb{R}$.

We can then define an SMC $\mathcal{S}$ whose morphisms are the pulse schedules themselves.

▶ **Definition 11** (SMC of Signals)**.** Given a set of channels $\mathbf{C}$, we define the SMC $\mathcal{S}$ such that:

- Objects are finite sequences of pairs $\mathbf{Ob}(\mathcal{S}) := (\mathbb{R} \times \mathbf{C})^*$ of a real number (the starting time of the signal) and a channel;
- Given $X, Y : \mathcal{S}$ with $X = [(s_1, c_1), \ldots, (s_m, c_m)]$ and $Y = [(t_1, d_1), \ldots, (t_n, d_n)]$, define the homset $\mathbf{Hom}_{\mathcal{S}}[X, Y]$ as

$$\{(\sigma, [\phi_1, \ldots, \phi_n]) \mid \sigma \in S_n \text{ such that } \forall k,\ c_k = d_{\sigma(k)} \text{ and } s_k \leq t_{\sigma(k)} \text{ and } \phi_k : [s_k, t_{\sigma(k)}) \to \mathbb{R}\}$$

  if $m = n$ and as the empty set otherwise, where S_n is the symmetric group. One could require some additional regularity conditions on the functions ϕ_k, which we simply omit;
- Identities are hence given by pairs $(e, [\emptyset, \ldots, \emptyset])$ where e is the identity permutation, and $\emptyset$ is the empty function;
- Composition is given by composing the permutations and concatenating the pulses:

$$(\sigma_1, [\phi_1, \ldots, \phi_n]) \circ (\sigma_2, [\psi_1, \ldots, \psi_n]) := (\sigma_1 \circ \sigma_2, [\gamma_1, \ldots, \gamma_n])$$

  where $\gamma_k : [r_k, t_{(\sigma_1 \circ \sigma_2)(k)}) \to \mathbb{R}$ is defined as

$$\gamma_k(t) := \begin{cases} \psi_k(t) & \text{if } t \in [r_k, s_{\sigma_2(k)}) \\ \phi_{\sigma_2(k)}(t) & \text{if } t \in [s_{\sigma_2(k)}, t_{(\sigma_1 \circ \sigma_2)(k)}) \end{cases}$$

  (intuitively, we keep track of how channels are permuted and apply the pulses sequentially)
- The tensor unit is given by the empty sequence $I := [] : \mathcal{S}$;
- Tensor product of objects is given by list concatenation;
- Tensor product of morphisms is given by the following

$$(\sigma_1, [\phi_1, \ldots, \phi_m]) \otimes (\sigma_2, [\psi_1, \ldots, \psi_n]) := (\sigma_1 \star \sigma_2, [\phi_1, \ldots, \phi_m, \psi_1, \ldots, \phi_n])$$

  where by $(- \star -) : S_m \times S_n \to S_{m+n}$ we denote the obvious group homomorphism.
- The unitors and associator are given by identities, and
- The symmetry is given by $(\sigma, [\emptyset, \ldots, \emptyset])$ where σ is the appropriate permutation and $\emptyset$ is the empty map.

We can use this category to build a model of the annotated language out of the input signals to the quantum chip, in such a way that the interpretation function $[\![\ ]\!]_p$ is exactly the compiler that translates from a GRAMPUS term to the corresponding input signals:

▶ **Proposition 12** ($\mathcal{S}$ is a categorical model of the annotated language)**.**

Assume given a correspondence between channels and qubits (in the simplest case this could be an injection $\mathbf{Q} \to \mathbf{C}$) and a pulse representation of each gate in $\mathbf{G}$. Then the SMC $\mathcal{S}$ can be extended to a categorical model of the annotated language as follows:

- *For every real number $x \in \mathbb{R}$, the functor* $\mathsf{Shift}(x)$ *is defined as follows:*

$$\mathsf{Shift}(x)[(t_1, c_1), \ldots, (t_n, c_n)] := [(t_1 + x, c_1), \ldots, (t_n + x, c_n)]$$

$$\mathsf{Shift}(x)(\sigma, [\phi_1, \ldots, \phi_n]) := (\sigma, [\phi_1(\cdot - x), \ldots, \phi_n(\cdot - x)])$$

- *For every real number $x \in \mathbb{R}$, the morphisms $delay_x$ are built from the identity permutation $e \in S_1$ and the pulse δ_0 which is constantly zero*[1]*.*

$$delay_x = (e, [\delta_0])$$

Proof. Functoriality and (symmetric) monoidality for $\mathsf{Shift}(x)$ are straightforward to verify. Furthermore, in this model, the structural natural isomorphisms making $\mathsf{Shift}(-)$ into a monoidal functor are identities, meaning it is *strict* monoidal. ◀

3.5 Syntactic Model and Completeness Theorem

We can build a model for either of our type theories out of the terms of the theory itself. It is generally considered desirable if this term model is initial in the category of models, with the interpretation function $[\![\ ]\!]$ being the unique morphism from the term model to an arbitrary model (see e.g. [24]). In this section, we prove that this is the case for both of our languages.

▶ **Definition 13** (Term category for the annotated Language)**.**
We define the model of the plain language $\mathbf{Syntax} = (\mathbf{Syntax}, X_-, f_-)$ as follows. We begin by defining a function $\mathbf{Ty}$ that sends contexts Γ to types $\mathbf{Ty}(\Gamma)$ by

$$\mathbf{Ty}([]) := 1; \quad \mathbf{Ty}(x :^d A, \Gamma) := (\boxed{d}\, A) \otimes \mathbf{Ty}(\Gamma)$$

We can then define the underlying category of **Syntax** as follows:

- Its objects are contexts,
- Its morphisms $f : \Gamma \to \Delta$ are derivable judgments $\Gamma \vdash t : \mathbf{Ty}(\Delta)$, quotiented by judgmental equality[2].
- Identities $\mathbf{id}_\Gamma : \Gamma \to \Gamma$ are defined by induction on Γ:

$$\mathbf{Id}_{[]} := *; \quad \mathbf{id}_{x:^d A,\Gamma} := (x, \mathbf{id}_\Gamma)$$

- Composition is given by inductively using the Tens-E and Unit-E rules: given $f : \Gamma \to \Delta$ and $g : \Delta \to \Xi$, if $\Delta = x_1 :^{d_1} A_1, \ldots, x_n :^{d_n} A_n$,

$$\begin{aligned} g \circ f := \text{let } (y_1, z_1) = f \quad &\text{in let box } d_1\ x_1 = y_1 \text{ in} \\ \text{let } (y_2, z_2) = z_1 \quad &\text{in let box } d_2\ x_2 = y_2 \text{ in} \\ \ldots& \\ \text{let } (y_n, z_n) = z_{n-1} \quad &\text{in let box } d_n\ x_n = y_n \text{ in} \\ \text{let } * = z_n \quad &\text{in } g \end{aligned}$$

  for distinct, fresh $y_1, \ldots, y_n, z_1, \ldots, z_n$. We can give a derivation for the judgment $\Gamma \vdash (g \circ f) : \mathbf{Ty}(\Xi)$ by generalizing the construction:

$$\frac{\Gamma \vdash t : \mathbf{Ty}(\Delta) \qquad \Delta, \Xi \vdash s : C}{\Gamma, \Xi \vdash \text{let}_\Delta\ t \text{ in } s : C}$$

and performing induction on Δ; the let_Δ statement above is defined in exactly the same way as composition, and generalizes it.

---

[1] Notice how these morphisms are not invertible: in general, due to phenomena such as decoherence, a qubit left alone will not stay the same. For a more detailed treatment of this, see section 3 of [13].

[2] In the rest of this definition, we will silently treat representatives of such equivalence classes as the classes themselves.

Verifying that the above definition indeed gives a category structure is routine: unitality and associativity of composition hinges on the usual substitution lemmas. We can further enhance the structure on **Syntax** to that of an SMC in the obvious way.

▶ **Definition 14** (Monoidal structure on **Syntax**). The category **Syntax** can be made into an SMC $(\mathbf{Syntax}, I, \otimes, \dots)$ by defining the following:

- Its unit object I is given by the unit type $I := 1$;
- Its tensor product of objects is given by context concatenation $\Gamma \otimes \Delta := \Gamma, \Delta$;
- Its tensor product of morphisms is given by inductively applying the Tens-E and Tens-I rules: if $\Gamma_1 \vdash t_1 : \mathbf{Ty}(\Delta_1)$ and $\Gamma_2 \vdash t_2 : \mathbf{Ty}(\Delta_2)$, define

$$\begin{aligned} t_1 \otimes t_2 := {} & \text{let } (x_1, y_1) = t_1 \quad \text{in} \\ & \text{let } (x_2, y_2) = y_1 \quad \text{in} \\ & \dots \\ & \text{let } (x_n, y_n) = y_{n-1} \text{ in} \\ & \text{let } * = y_n \quad \text{in} \\ & (x_1, \dots (x_n, t_2) \dots) \end{aligned}$$

- Checking that unitors and associators can just be identities is straightforward;
- The symmetry can be defined with a similar inductive strategy as the tensor product of morphisms (employing the Swap rule appropriately).

Checking the axioms of an SMC is straightforward. Since we shall evidently interpret qubits and gates as themselves (i.e. a qubit $q \in \mathbf{Q}$ is interpreted as the context $x :^0 q$, and similarly for gates), we only need to define the SMMC structure and the delay_d morphisms.

▶ **Definition 15** (SMMC structure and delay_d for **Syntax**). For any given real number $d \in \mathbb{R}$, define the symmetric monoidal endofunctor $\mathsf{Shift}(d) : \mathbf{Syntax} \to \mathbf{Syntax}$ as follows:

- On objects Γ, define $\mathsf{Shift}(d)(\Gamma) := d + \Gamma$
- On morphisms $\Gamma \vdash t : \mathbf{Ty}(\Delta)$, define

$$\begin{aligned} \mathsf{Shift}(d)(t) := {} & \text{let } (x_1, y_1) = t \quad && \text{in} \\ & \text{let } (x_2, y_2) = y_1 \quad && \text{in} \\ & \dots \\ & \text{let } (x_n, y_n) = y_{n-1} \quad && \text{in} \\ & (\text{box } d\ x_1, \dots (\text{box } d\ x_n, \text{box } d\ y_n) \dots) \end{aligned}$$

Endowing $\mathsf{Shift}(d)$ with the further structure of a symmetric monoidal functor is a simple exercise. Similarly, given a qubit $q \in \mathbf{Q}$ and a real number $d \in \mathbb{R}$, we define $\mathrm{delay}_d : [x :^{-d} q] \to [x :^0 q]$ as (the obvious derivation for) the judgment

$$[x :^{-d} q] \vdash (\text{box } 0\ \mathrm{delay}_d(x), *) : q \otimes 1$$

The same constructions as above (Definitions 13 and 14) can be repeated for the plain language, producing an SMC $\mathbf{Syntax}_u$. By construction, **Syntax** (resp. $\mathbf{Syntax}_u$) is a model of the annotated (resp. plain) language.

▶ Remark 16. Every categorical model of the plain (resp. annotated) language $(\mathcal{C}^{\otimes}, X_{-}, f_{-})$ (resp. $(\mathcal{C}^{\otimes,\bullet}, X_{-}, f_{-}, \mathsf{Shift}(-), \mathrm{delay}_{-})$) admits a symmetric monoidal functor $\mathbf{Syntax}_u \to \mathcal{C}^{\otimes}$ preserving qubits and gates (resp. symmetric monoidal $\mathbb{R}^+$-*linear* functor[3] $\mathbf{Syntax} \to \mathcal{C}^{\otimes}$, also appropriately preserving delay_{-}). This functor is unique up to natural isomorphism that preserves qubits and gates, implying initiality for **Syntax**.

Proof. Such a functor can be constructed by adapting Definition 3 (resp. Definition 7). ◀

▶ **Theorem 17** (Completeness). *Suppose* $\Gamma \vdash_d s : A$ *and* $\Gamma \vdash_d t : A$. *If* $[\![s]\!]_p = [\![t]\!]_p$ *in every model, then* $\Gamma \vdash_d s = t : A$.

Proof. If the equality holds in every model then it holds in **Syntax**, which means precisely that $\Gamma \vdash_d s = t : A$. ◀

A similar Completeness Theorem can be proved for the plain language.

3.6 The Category Circ

We can give an alternative characterization of the initial model **Syntax** as follows.

▶ **Definition 18.** The $\mathbb{R}^+$-SMMC Circ is defined in the following way:

- Its objects are finite sequences of pairs (d, q) of a real number and a qubit: $\mathbf{Ob}(\mathsf{Circ}) := (\mathbb{R} \times \mathbf{Q})^*$, hence are exactly contexts whose types are restricted to just qubits,
- Its unit object is the empty sequence,
- Its tensor product is given, on objects, by concatenation,
- Its morphisms are generated (under composition and tensor products) by permutations, delays for every qubit q and real number x

  $$\mathrm{delay}_d : [(x, q)] \to [(x + d, q)]$$

  and gates for every $(G, d, q_1, \ldots, q_n) \in \mathbf{G}$

  $$\phi_G : [(x, q_1), \ldots, (x, q_n)] \to [(x + d, q_1), \ldots, (x + d, q_n)]$$

- Unitors and associators are just identities,
- Symmetries are the appropriate permutations,
- For every real number $d \in \mathbb{R}$, the endofunctor $\bullet(d)$ acts
  - On objects, by adding d to the first component of every element of the sequence,

    $$\bullet(d)([(x_1, q_1), \ldots, (x_n, q_n)]) := [(x_1 + d, q_1), \ldots, (x_n + d, q_n)]$$

  - On morphisms, it is the identity on permutations, maps delays to delays appropriately, and similarly for gates.

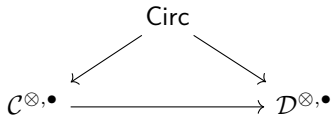
Let us call this category circuits (Circ) and write its homsets $\Gamma \multimap_G \Delta$. A similar definition gives Circ_u, where we omit the delay morphisms and the timing data from the objects.

It is important to notice that Circ (resp. Circ_u) is *not* the category of contexts over the annotated (plain) theory: it is the free *strict* SMMC (SMC) on the data encoded by $\mathbf{Q}$ and $\mathbf{G}$, while the category of contexts would be the free SMMC on such data. The equivalence between the two is a consequence of a general strictification theorem[4].

[3] I.e. a strong linear functor [8, definition 3.3.2] such that the underlying functor and the lineator are strong symmetric monoidal

[4] While the case for the unannotated language is well-known ([14] and [15, chapter XI]), in the general case it is easier to prove the equivalence directly.

▶ Remark 19. A strong symmetric monoidal $\mathbb{R}^+$-linear functor $\mathcal{C}^{\otimes,\bullet} \to \mathcal{D}^{\otimes,\bullet}$ between models preserves the choices of qubits, gates and delays precisely when it makes the following diagram commute



Hence the category of models is equivalent to the coslice category $\mathsf{Circ} \downarrow \mathbb{R}$-**SMMC**.

In complete analogy, the category of models for the unannotated language is equivalent to the coslice $\mathsf{Circ}_u \downarrow$ **SMC**.

4 Normal forms

In this section we describe a *normal* representation for GRAMPUS; i.e. one that makes semantically equivalent terms syntactically α-equivalent. At the end of the section, we will deal with the representation for Circ morphisms themselves. But we first discuss how such morphisms should be embedded into the terms of a programming language. We construct such normal forms in three stages:

1. Decomposition of the inputs using eliminators ($\Gamma \vdash_e C$);
2. Application of circuit gates and delays ($\Gamma \multimap_G \Delta$, see Section 3.6);
3. Construction of the output of the desired type by using constructors ($\Gamma \vdash_n C$).

These judgements are defined by the following rules.

$$\frac{\Theta, \Delta \vdash_e t : A}{\Theta, x :^d 1, \Delta \vdash_e \mathrm{let} * = x \,\mathrm{in}\, t : A}\text{Unit-E} \qquad \frac{}{\vdash_n * : 1}\text{Unit-I}$$

$$\frac{\Theta, x :^d A, y :^d B, \Delta \vdash_e t : C}{\Theta, z :^d (A \otimes B), \Delta \vdash_e \mathrm{let}(x,y) = z \,\mathrm{in}\, t : C}\text{Tens-E} \qquad \frac{\Gamma \vdash_n s : A \qquad \Delta \vdash_n t : B}{\Gamma, \Delta \vdash_n (s,t) : A \otimes B}\text{Tens-I}$$

$$\frac{\Theta, x :^{d+e} A, \Delta \vdash_e t : B}{\Theta, z :^d \boxed{e} A, \Delta \vdash_e \mathrm{let\ box}\ e\ x = z\ \mathrm{in}\ t : B}\text{Box-E} \qquad \frac{\Gamma \vdash_n t : A}{d + \Gamma \vdash_n \mathrm{box}\ d\ t : \boxed{d} A}\text{Box-I}$$

$$\frac{\Delta \vdash_n t : B \qquad c : \Theta \multimap_G \Delta}{\Theta \vdash_e t\ \mathsf{after}\ c : B}\text{Circ} \qquad \frac{}{x :^0 q \vdash_n x : q}\text{Var}$$

In the above, the eliminators only work on the first non-qubit variable in the context (Θ stands for a context with only qubits). Because those are only variables, no further reduction can happen.

▶ Remark 20. Because gates have the same domain and codomain up to permutation and later durations, so does any circuit applied in the Circ rule, and thus variable names can remain the same before and after applying the circuit (even if they can be permuted).

With the exception of the Circ rule, the rules are a subset of GRAMPUS, and thus are given the same semantics. With this in place, we can show that the effect that the semantics of normal-form introduction and elimination rules is trivial.

▶ **Lemma 21.** *If* $\Gamma \vdash_n t : A$, *then* $[\![\Gamma]\!]_p = [\![A]\!]_p$.

Proof. The proof is a simple induction on the structure of t, relying on the fact that the interpretation of constructors do not use gates or delays. Additionally $id \otimes id = id$ and $\mathsf{Shift}(d)\ id = id$. ◀

To state a similar result about eliminators, we want to generalise over them. We can do so by defining the relation $\Gamma \multimap_e \Delta$, which is any context transformation performed by an eliminator in the form allowed by the $\vdash_e$ judgement.

▶ **Definition 22** (Relation $\Gamma \multimap_e \Delta$)**.**

$$* = x \ : \ (x :^d 1 \multimap_e []) \qquad (x, y) = z \ : \ (z :^d (A \otimes B) \multimap_e x :^d A, y :^d B)$$

$$\mathrm{box}\ d\ x = z \ : \ (z :^d \boxed{e} A \multimap_e x :^{d+e} A)$$

Then all normalised eliminators become instances of the following rule:

$$\frac{e : \Gamma \multimap_e \Delta \qquad \Theta, \Delta, \Xi \vdash_e t : A}{\Theta, \Gamma, \Xi \vdash_e \mathrm{let}\, e \,\mathrm{in}\, t : A}$$

and we can state the desired lemma as follows.

▶ **Lemma 23.** *For any $e : \Gamma \multimap_e \Delta$, we have $[\![\mathrm{let}\, e \,\mathrm{in}\, t : A]\!]_p = [\![t]\!]_p$.*

Proof. By case analysis. For example the case for split is: $[\![\mathrm{let}(x, y) = z \,\mathrm{in}\, t]\!]_p = [\![t]\!]_p \circ \mathsf{Shift}(d)[\![z]\!] = [\![t]\!]_p$ ◀

▶ **Theorem 24** (Normalization)**.** *Given a term $\Gamma \vdash t : A$, there is a unique u such that $\Gamma \vdash_e u : A$ and $[\![t]\!]_{\mathsf{Circ}} = [\![u]\!]_{\mathsf{Circ}}$.*

Proof.

- Recall that in Circ, contexts evaluate to contexts. More precisely $[\![\Gamma]\!]_{\mathsf{Circ}}$ is a list of (delayed) qubits from Γ, in the same order.
- The circuit to embed (in the exactly single occurrence of rule *Circ*) is given by the semantics into Circ; $[\![t]\!]_{\mathsf{Circ}} : [\![\Gamma]\!]_{\mathsf{Circ}} \multimap_G [\![A]\!]_{\mathsf{Circ}}$.
- The matching constructor $[\![A]\!]_{\mathsf{Circ}} \vdash_n s : A$ is constructed by induction on the type. Furthermore constructor rules are purely type-directed, and exactly one rule applies at any point.
- The elimination sequence which matches Γ to $[\![\Gamma]\!]_{\mathsf{Circ}}$ is done by induction on the suffix of Γ which starts with a non-elementary qubit type. As for constructors, there is a single rule which can apply at any point, because the first non-elementary type dictates which rule applies. When there are only qubits in the context, rule Circ applies on $[\![t]\!]_{\mathsf{Circ}}$, and this is always applicable because the eliminators previously applied did not change the context semantics (Lemma 23). Hence there is always exactly one eliminator to use.

The fact that this normal form respects semantics is a consequence of Lemmata 21 and 23. ◀

Discussion

We have so far established the existence of unique normal forms, but the computation itself is grouped in a single morphism of the SMC Circ. Finally we can consider what method should be used to best represent morphisms in this category. There are many possible choices, because SMCs feature a number of laws relating objects and morphisms.

One route is to map them onto the GRAMPUS syntax, such that for any morphism $h : \Gamma \multimap_G \Delta$, give a term $\Delta, \Xi \vdash t : C$ we can construct a term of type C in context Γ, Ξ. The structural morphisms correspond to context shuffling. The gates and delays must be

embedded in the following way. Let us consider the case of a gate shifted by time e, ie. mapping context $e + \Gamma$ to $e + \Delta$. The corresponding term is:

$$\text{let box}\, e\, z = \text{box}\, e\, G(\Gamma) \,\text{in let}_{e+\Delta} = z \,\text{in}\, t.$$

To represent composition, one can always apply one such transformation and then the other.

Another possible choice is to represent Circ morphisms is Einstein notation, commonly used to represent tensor expressions in physics. It is clear that plain tensors are representable this way, but pulse schedules can be represented using Einstein notation just as well, because every free SMC can be represented this way. In Einstein notation, every input *and* output variable is made explicit. Every morphism is annotated with its inputs as subscripts and outputs as superscript. So a circuit composed of a single gate $H : (x : q_1, y : q_2 \multimap_G z :^d q_1, w :^d q_2)$ is written H^{zw}_{xy}. The composition $f \circ g$ is written as a multiplication $F^{intermediate}_{inputs} G^{outputs}_{intermediate}$. Unitors, associators, and exchange morphisms are represented by the appropriate Kronecker deltas. Then the laws of the SMCs correspond to simplifications of such Kronecker deltas. After such simplifications, testing the equality of morphisms in Einstein notation is a simple syntactic test, up to renaming of intermediate variable names. We direct the reader to [5] for details on conversions back and forth between Einstein notation and morphisms in an arbitrary SMC.

In the interests of implementation and formal verification, however, a variable-free representation is used for scheduling. This representation is described in Section 6.3.

5 Hardware Model and Correctness

A *hardware model* specifies the action of a complete pulse schedule (given as a function $Time \to Channel \to \mathbb{R}$) on the state of a quantum chip. If the chip has n qubits, then the state of the chip is given by a unit vector in $\mathbb{C}^{2^n}$, and inputting a pulse schedule will perform a certain unitary operator $U : \mathbb{C}^{2^n} \multimap \mathbb{C}^{2^n}$ on the state.

In principle, we can calculate this unitary from the Hamiltonian of the chip, which is determined by the chip's circuit design. If there are k input channels $V_1(t), \ldots, V_k(t)$ (where $V_i(t)$ is the voltage applied on input channel i at time t), then the Hamiltonian can be given in the form of a linear operator $H(V_1(t), \ldots, V_k(t)) : \mathbb{C}^{2^n} \to \mathbb{C}^{2^n}$.

The evolution of the state of the quantum chip is then given by $|\phi(t)\rangle = U(t)|\phi(0)\rangle$, where $|\phi(t)\rangle \in \mathbb{C}^{2^n}$ is the state at time t and $U(t)$ is the unitary

$$U(t) = \exp\left(-\frac{i}{\hbar}\int_0^t H(V_1(t'), \ldots, V_k(t'))dt'\right) \quad .$$

Given a quantum chip whose Hamiltonian is given by the above, we say that the pulse schedule given by $V_1(t'), \ldots, V_k(t')$, with start time 0 and end time t, *implements* the unitary $U(t)$.

▶ **Theorem 25** (Correctness). *Assume that we have a model of the plain language in Hilbert spaces in which every gate G is assigned the unitary U_G; and a model of the annotated language in $\mathcal{S}$ such that, for every gate G, the morphism f_G is a pulse schedule that implements U_G. Then the diagram (1) commutes.*

Proof. The proof is by induction on the derivation of $\Gamma \vdash_d t : A$ and relies on the following observations: If pulse schedule p implements U and q implements V then $q \circ p$ implements $V \circ U$. If p implements U and q implements V then $p \otimes q$ implements $U \otimes V$. We always have p and $\text{Shift}(d)(p)$ implement the same unitary. ◀

Therefore, if a scheduling algorithm (a function from the plain language to the annotated language) is a right inverse to erasure, then the pulse schedule generated implements the same unitary as the original circuit.

6 Intermediate Representations

Intermediate representations in the dependently-typed programming language Agda [20] are introduced in order to facilitate the implementation of scheduling; they are isomorphic respectively to the plain and annotated surface languages ($\Gamma \vdash t : A$ and $\Gamma \vdash_d t : A$) up to normalisation, but avoid redexes while being more normalised than a free SMC generated by qubits and gates. The source code is available online;[5] listings in this section make minor cosmetic changes to improve readability.

6.1 Plain circuits

The plain language is represented through the type family `Circuit`, indexed by two lists of qubits indicating the types of the circuit's input and output.

Listing 1 Circuit.

```
data Circuit : Qubits → Qubits → Set where
  wire : Circuit a a
  gate : Gate a → Circuit a a
  serial : Circuit a b → Circuit b c → Circuit a c
  parallel :
    par-qubits a1 a2 a3 →
    par-qubits b1 b2 b3 →
    Circuit a1 b1 →
    Circuit a2 b2 →
    Circuit a3 b3
```

The type `par-qubits x y z` used in the `parallel` constructor maps each qubit to a bijection between its occurrences in `x` or `y` and its occurrences in `z`.

It is defined as follows.

```
par-qubits : Qubits → Qubits → Qubits → Set
par-qubits x y z = (q : Qubit) → Bij ((q ∈ x) ⊎ (q ∈ y)) (q ∈ z)
```

This is equivalent to permutations from the concatenation of x and y to z, meaning that the `parallel` constructor captures association, unitors, and exchange.

As is the case for the plain language, the `Circuit` datatype does not preclude multiple occurrences of the same qubit, but is well-behaved in that it treats occurrences as distinct individuals (informally, they are never confused with one another, copied, or destroyed; the definition of `par-qubits` plays a significant role in this). As a result, issues of duplication can be safely confined to the observation that an element of type `Circuit a b` is physically realizable exactly when `a` (or equivalently, `b`) is free of duplicates.

6.2 Annotated circuits

In order to describe the annotated language, the notion of a *timing function* is introduced. A timing function on a list x of qubits is a function that takes a qubit q and an element of $q \in x$, i.e. the index of an occurrence of q in x, and produces a time value:

[5] Repository: `https://codeberg.org/radams78/grampus.git`

Listing 2 Timing.

```
Timing : Qubits → Set
Timing qs = (q : Qubit) → q ∈ qs → Time
```

The annotated language can then be represented in a similar way to the plain language, but with as additional type indices two timing functions f and g; f sends each input qubit to the time the annotated circuit needs it, and g sends each output qubit to the time the annotated circuit is done with it.

If an annotated circuit were depicted graphically as a schedule of gates, with qubits on the y-axis and time on the x-axis, the two timing functions would represent the schedule's left and right boundaries. Accordingly, they are referred to respectively as the annotated circuit's *left-timing* and *right-timing*. With respect to a typing judgement $\Gamma \vdash_d t : A$, f and g capture annotations in Γ as well as boxes in Γ and A.

The datatype – a possible representation of the category Circ– is then defined as follows:

Listing 3 AC.

```
data AC : (a b : Qubits) → Timing a → Timing b → Set where
  delay : f ≤ g → AC a a f g
  gate :
    (gt : Gate a) →
    constantly t f →
    constantly (t + duration gt) g →
    AC a a f g
  serial : AC a b f g → AC b c g h → AC a c f h
  parallel :
    (pa : par-qubits a1 a2 a3) →
    (pb : par-qubits b1 b2 b3) →
    (par-timing pa f1 f2 f3) →
    (par-timing pb g1 g2 g3) →
    AC a1 b1 f1 g1 →
    AC a2 b2 f2 g2 →
    AC a3 b3 f3 g3
```

The type `constantly t f` asserts that `f` gives time `t` for each qubit in its domain, and the relation $\leq$ on timing functions is defined pointwise. For the `gate` constructor it is noted that the uses of `constant` mean the gate uses all its qubits over the same time interval; graphically, its schedule is a rectangle, that is to say its left and right edges are flat rather than ragged.

The type `par-timing` ensures that timing functions agree for each qubit occurrence across the bijection specified by `par-qubits`, motivated both practically by the encoding of timing functions, and also by the interest in distinguishing occurrences of qubits as described above in relation to the definition of `par-qubits`.

Finally, annotations can be erased: each constructor of `AC` is simply replaced by its counterpart in `Circuit` (with `delay` becoming `wire`). This is carried out by the function `erase-timing`, with the following type:

Listing 4 erase.

```
erase : AC a b f g → Circuit a b
```

6.3 As-late-as-possible scheduling

The *as-late-as-possible* scheduling algorithm is implemented through the function `alap`:

Listing 5 As Late As Possible.

```
alap : (g : Timing b) → Circuit a b → Σ (Timing a) (λ f → AC a b f g)
```

It takes a right-timing g and plain circuit as arguments, and returns both an annotated circuit and its left-timing f. It is implemented by structural induction on the circuit, packing the schedule as far to the right as possible and propagating timings from right to left.

The non-trivial case is that of a `gate`: gates operate on all of their qubits over the same time interval, hence their constant left and right timing functions in `AC`, but the provided right timing g is arbitrary (graphically: the gate is a rectangle, while g might be jagged). This is solved by first "flattening" g by inserting a delay with right-timing g and constant left-timing $h = \mathrm{const}(\min(g))$. The gate is then scheduled left of this delay, i.e. with right-timing h. It is observed that the `delay` will have duration zero for at least one qubit occurrence (where g attains its minimum), so we could not have hoped to schedule the gate later, and `alap` is indeed performing as-late-as-possible scheduling as claimed.

While the full definitions of equivalence on `Circuit` and `AC` are not yet complete, a rudimentary notion of equivalence, written `_≈_`, is defined on `Circuit`: it is the smallest relation that is reflexive, respected by `serial` and `parallel`, and equates `serial ckt wire` with `ckt`. This form of equivalence is sufficient to show that `alap` (with some decoration) is a valid scheduling algorithm, which is to say a right inverse to erasure:

Listing 6 erase-alap.

```
erase-alap : (g : Timing b) (ckt : Circuit a b) →
  (erase ∘ proj₂ ∘ alap g) ckt ≈ ckt
```

From the result at the end of Section 5 we then know that a pulse schedule generated from the annotated output of `alap` implements the same unitary as the original circuit.

7 Related Work

There are a growing number of quantum programming languages in existence now, both imperative languages such as Q# [17], and functional languages such as LIQUi|> [25], Quipper [11] and Proto-Quipper [23]. All of these languages have as their intended semantics *quantum circuits*. They provide the user with high-level tools for describing the construction of quantum algorithms. They thus correspond to what we called the *plain language* in this paper.

There have been a few languages that use type theory in their design, including Proto-Quipper [10], Qwire [21] and Silq [6]. The survey by Nguyen [18] provides a review of the applications of type theory to quantum programming language design. Again, these languages describe quantum circuits without timing information, and so correspond to our plain language.

The pulse schedules are sent in the form of an extremely low-level language such as Q1ASM [22] to the wavefunction generator that creates the signals passed down the input channels of the quantum hardware itself.

There are very few languages that represent a quantum circuit with timing information and explicit delays – the intermediate stage between a quantum circuit (without timing information) and a language such as Q1ASM. The only such language that the authors are aware of is OpenPulse [16, 3], until recently a subset of OpenQASM [9].

We therefore see GRAMPUS as filling a need of a language that represents quantum circuits with timing information, in a form that is suitable for formal semantics and for building a verified quantum compiler.

The design of GRAMPUS was heavily inspired by Granule [12], an implementation of graded modal type theory in Haskell.

8 Conclusion

We have given two linear type theories that can represent quantum circuits with timing information. We have given a model in terms of the input signals to a quantum chip, in such a way that the interpretation function is the algorithm that perform compilation. We have also proved that the term models are the initial models, giving confidence that the syntax and semantics are in close correspondence.

For future work, we want to expand our language GRAMPUS with the other features used in quantum algorithms, in particular measurement, and the ability to perform hybrid algorithms where the result of a measurement affects which gates are subsequently performed.

References

1 Andreas Abel and Jean-Philippe Bernardy. A unified view of modalities in type systems. *Proceedings of the ACM on Programming Languages*, 4(ICFP):90:1–90:28, 2020. doi:10.1145/3408972.

2 Robin Adams, Jean-Philippe Bernardy, Lorenzo Perticone, and Jeremy Pope. Graded Modal Type Theory for Pulse Schedules. Software, This research was funded by the SSF (Swedish Foundation for Strategic Research) project QuantumStack, grant number FUS21-0063., swhId: swh:1:dir:bc79e493a8f11d72e5382289e1ef8d1284024082 (visited on 2026-07-23). URL: https://codeberg.org/radams78/grampus.git, doi:10.4230/artifacts.27329.

3 Thomas Alexander, Naoki Kanazawa, Daniel J. Egger, Lauren Capelluto, Christopher J. Wood, Ali Javadi-Abhari, and David C. McKay. Qiskit Pulse: Programming quantum computers through the cloud with pulses. *Quantum Science and Technology*, 5(4):044006, August 2020. doi:10.1088/2058-9565/aba404.

4 Nick Benton, Gavin Bierman, Valeria De Paiva, and Martin Hyland. A term calculus for intuitionistic linear logic. In Marc Bezem and Jan Friso Groote, editors, *International Conference on Typed Lambda Calculi and Applications*, volume 664 of *Lecture Notes in Computer Science*, pages 75–90. Springer, 1993. doi:10.1007/BFb0037099.

5 Jean-Philippe Bernardy and Patrik Jansson. Domain-specific tensor languages. *Journal of Functional Programming*, 35:e9, 2025. doi:10.1017/S0956796825000048.

6 Benjamin Bichsel, Maximilian Baader, Timon Gehr, and Martin Vechev. Silq: A high-level quantum language with safe uncomputation and intuitive semantics. In Alastair F. Donaldson and Emina Torlak, editors, *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, PLDI 2020, pages 286–300, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3385412.3386007.

7 Aloïs Brunel, Marco Gaboardi, Damiano Mazza, and Steve Zdancewic. A core quantitative coeffect calculus. In Zhong Shao, editor, *Programming Languages and Systems - 23rd European Symposium on Programming, ESOP 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5–13, 2014, Proceedings*, volume 8410 of *Lecture Notes in Computer Science*, pages 351–370. Springer, 2014. doi:10.1007/978-3-642-54833-8_19.

8 Matteo Capucci and Bruno Gavranović. Actegories for the working amthematician, 2024. arXiv:2203.16351.

9 Andrew Cross, Ali Javadi-Abhari, Thomas Alexander, Niel De Beaudrap, Lev S. Bishop, Steven Heidel, Colm A. Ryan, Prasahnt Sivarajah, John Smolin, Jay M. Gambetta, and Blake R. Johnson. OpenQASM 3: A broader and deeper quantum assembly language. *ACM Transactions on Quantum Computing*, 3(3):12:1–12:50, September 2022. doi:10.1145/3505636.

10 Peng Fu, Kohei Kishida, Neil J. Ross, and Peter Selinger. A tutorial introduction to quantum circuit programming in dependently typed Proto-Quipper. In Ivan Lanese and Mariusz Rawski, editors, *Reversible Computation - 12th International Conference, RC 2020, Oslo, Norway, July 9-10, 2020, Proceedings*, volume 12227 of *Lecture Notes in Computer Science*, pages 153–168, Cham, 2020. Springer International Publishing. doi:10.1007/978-3-030-52482-1_9.

11 Alexander S. Green, Peter LeFanu Lumsdaine, Neil J. Ross, Peter Selinger, and Benoît Valiron. Quipper: A scalable quantum programming language. In Hans-Juergen Boehm and Cormac Flanagan, editors, *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 333–342, New York, NY, USA, 2013. Association for Computing Machinery. doi:10.1145/2491956.2462177.

12 Jack Hughes and Dominic Orchard. Program synthesis from graded types. In Stephanie Weirich, editor, *Programming Languages and Systems - 33rd European Symposium on Programming, ESOP 2024, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2024, Luxembourg City, Luxembourg, April 6–11, 2024, Proceedings, Part I*, volume 14576 of *Lecture Notes in Computer Science*, pages 83–112, Cham, 2024. Springer Nature Switzerland. doi:10.1007/978-3-031-57262-3_4.

13 P. Krantz, M. Kjaergaard, F. Yan, T. P. Orlando, S. Gustavsson, and W. D. Oliver. A quantum engineer's guide to superconducting qubits. *Applied Physics Reviews*, 6(2):021318, June 2019. doi:10.1063/1.5089550.

14 Saunders Mac Lane. Natural associativity and commutativity. *Rice Institute Pamphlet - Rice University Studies*, 49:28–46, 1963.

15 Saunders Mac Lane. *Categories for the Working Mathematician*. Number 5 in Graduate texts in mathematics. Springer-Verlag, New York, NY, 1971.

16 David C. McKay, Thomas Alexander, Luciano Bello, Michael J. Biercuk, Lev Bishop, Jiayin Chen, Jerry M. Chow, Antonio D. Córcoles, Daniel Egger, Stefan Filipp, Juan Gomez, Michael Hush, Ali Javadi-Abhari, Diego Moreda, Paul Nation, Brent Paulovicks, Erick Winston, Christopher J. Wood, James Wootton, and Jay M. Gambetta. Qiskit backend specifications for openqasm and openpulse experiments, 2018. arXiv:1809.03452.

17 Microsoft. *Q# Language Specification*, 2020. URL: https://github.com/microsoft/qsharp-language/tree/main/Specifications/Language#q-language.

18 Van Han Nguyen. A survey of quantum type theory: From linearity to formal verification. *EAI Endorsed Transactions on Context-aware Systems and Applications*, 10, July 2025. doi:10.4108/eetcasa.9669.

19 Michael A. Nielsen and Isaac L. Chuang. *Quantum Computation and Quantum Information*. Cambridge University Press, 2000. URL: https://www.cambridge.org/de/academic/subjects/physics/quantum-physics-quantum-information-and-quantum-computation/quantum-computation-and-quantum-information-10th-anniversary-edition?format=HB.

20 Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Chalmers University of Technology, 2007.

21 Jennifer Paykin, Robert Rand, and Steve Zdancewic. QWIRE: A core language for quantum circuits. In Giuseppe Castagna and Andrew D. Gordon, editors, *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, POPL '17, pages 846–858, New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3009837.3009894.

22 QBLOX BV. Q1ASM user guide. https://docs.qblox.com/en/v2026.04.0/products/qblox_instruments/q1/, 2026.
23 Neil J. Ross. Algebraic and logical methods in quantum computation, 2017. arXiv:1510.02198.
24 Taichi Uemura. A general framework for the semantics of type theory. *Mathematical Structures in Computer Science*, 33(3):134–179, March 2023. doi:10.1017/s0960129523000208.
25 Dave Wecker and Krysta M. Svore. LIQUi|>: A software design architecture and domain-specific language for quantum computing, 2014. arXiv:1402.4467.

9 Appendix – Commuting Conversions

The full list of commuting conversions is given below. Each one should be given as a rule of deduction with premises that ensure the left-hand side is well-typed (which ensures the right-hand side is well-typed too). We omit these to save space here.

$$(\text{let } * = s \text{ in let } * = t \text{ in } u) = (\text{let } * = \text{let } * = s \text{ in } t \text{ in } u)$$
$$(\text{let } * = s \text{ in let } * = t \text{ in } u) = (\text{let } * = t \text{ in let } * = s \text{ in } u)$$
$$(\text{let } * = s \text{ in } G(t_1, \dots, t_n)) = G(t_1, \dots, t_{i-1}, \text{let } * = s \text{ in } t_i, t_{i+1}, \dots, t_n)$$
$$(\text{let } * = s \text{ in } (t, u)) = (\text{let } * = s \text{ in } t, u)$$
$$(\text{let } * = s \text{ in } (t, u)) = (t, \text{let } * = s \text{ in } u)$$
$$(\text{let } * = s \text{ in let } (x, y) = t \text{ in } u) = (\text{let } (x, y) = \text{let } * = s \text{ in } t \text{ in } u)$$

$$(\text{let } * = s \text{ in let } (x, y) = t \text{ in } u) = (\text{let } (x, y) = t \text{ in let } * = s \text{ in } u)$$
$$(\text{let } * = s \text{ in box } d\ t) = \text{box } d(\text{let } * = s \text{ in } t)$$
$$(\text{let } * = s \text{ in let box } d\ x = t \text{ in } u) = (\text{let box } d\ x = \text{let } * = s \text{ in } t \text{ in } u)$$
$$(\text{let } * = s \text{ in let box } d\ x = t \text{ in } u) = (\text{let box } d\ x = t \text{ in let } * = s \text{ in } u)$$
$$(\text{let } (x, y) = s \text{ in let } * = t \text{ in } u) = (\text{let } * = \text{let } (x, y) = s \text{ in } t \text{ in } u)$$
$$(\text{let } (x, y) = s \text{ in let } * = t \text{ in } u) = (\text{let } * = t \text{ in let } (x, y) = s \text{ in } u)$$
$$(\text{let } (x, y) = s \text{ in } G(t_1, \dots, t_n)) = G(t_1, \dots, t_{i-1}, \text{let } (x, y) = s \text{ in } t_i, t_{i+1}, \dots, t_n)$$

$$(\text{let } (x, y) = s \text{ in } (t, u)) = (\text{let } (x, y) = s \text{ in } t, u)$$
$$(\text{let } (x, y) = s \text{ in } (t, u)) = (t, \text{let } (x, y) = s \text{ in } u)$$
$$(\text{let } (x, y) = s \text{ in let } (z, w) = t \text{ in } u) = (\text{let } (z, w) = \text{let } (x, y) = s \text{ in } t \text{ in } u)$$
$$(\text{let } (x, y) = s \text{ in let } (z, w) = t \text{ in } u) = (\text{let } (z, w) = t \text{ in let } (x, y) = s \text{ in } u)$$
$$(\text{let } (x, y) = s \text{ in box } d\ t) = \text{box } d(\text{let } (x, y) = s \text{ in } t)$$
$$(\text{let } (x, y) = s \text{ in let box } d\ z = t \text{ in } u) = (\text{let box } d\ z = \text{let } (x, y) = s \text{ in } t \text{ in } u)$$
$$(\text{let } (x, y) = s \text{ in let box } d\ z = t \text{ in } u) = (\text{let box } d\ z = t \text{ in let } (x, y) = s \text{ in } u)$$
$$(\text{let box } d\ z = s \text{ in let } * = t \text{ in } u) = (\text{let } * = \text{let box } d\ z = s \text{ in } t \text{ in } u)$$

$$
\begin{aligned}
(\text{let box } d\ z = s \text{ in let } * = t \text{ in } u) &= (\text{let } * = t \text{ in let box } d\ z = s \text{ in } u) \\
(\text{let box } d\ z = s \text{ in } G(t_1, \ldots, t_n)) &= G(t_1, \ldots, t_{i-1}, \\
&\qquad \text{let box } d\ z = s \text{ in } t_i, \\
&\qquad t_{i+1}, \ldots, t_n) \\
(\text{let box } d\ z = s \text{ in } (t, u)) &= (\text{let box } d\ z = s \text{ in } t, u) \\
(\text{let box } d\ z = s \text{ in } (t, u)) &= (t, \text{let box } d\ z = s \text{ in } u) \\
(\text{let box } d\ z = s \text{ in let } (x, y) = t \text{ in } u) &= (\text{let } (x, y) = \text{let box } d\ z = s \text{ in } t \text{ in } u) \\
(\text{let box } d\ z = s \text{ in let } (x, y) = t \text{ in } u) &= (\text{let } (x, y) = t \text{ in let box } d\ z = s \text{ in } u) \\
(\text{let box } d\ z = s \text{ in box } d'\ t) &= \text{box } d'\ (\text{let box } d\ z = s \text{ in } t) \\
(\text{let box } d'\ z = s \text{ in let box } d\ x = t \text{ in } u) &= (\text{let box } d\ x = t \text{ in let box } d'\ z = s \text{ in } u)
\end{aligned}
$$